Image File System

Reference Manual

Version 5.1

September 4, 1991

Acknowledgements

Based on suggestions by Wesley Snyder of North Carolina State University and Joe Erkes of General Electric's Corporate Research and Development Center, the staff of Communication Unlimited, Inc. designed and coded IFS version 1 during the fall on 1986. Mark Lanzo designed and implemented most of versions 1-4, although other members of the CUI staff contributed heavily, including Katie Boone, Mark Yarboro, Gary McCauley, Bennett Groshong, and Paul Hemler. Katie Boone is responsible for version 5.

Contents

List of Figures

Chapter 1

IFS Reference Manual

IFS (**I**mage **F**ile **S**ystem) is a set of routines used to manipulate images, from within *C* programs. An *image* just refers to any array of data. The term came into use because IFS was originally written to manipulate 2-dimensional pictures, such as ones obtained from a standard camera. However, IFS is not restricted to 2-dimensional images, and is capable of handling arrays of arbitrary dimensionality. In the current release of IFS, most of the image manipulating routines are designed specifically for 2-dimensional data. Later releases of IFS will have enhanced multidimensional routines.

IFS is a simple system to use, and hides from the user the implementation details of basic data manipulation functions, such as allocating space for data arrays, performing I/O, and manipulating images with different data formats. IFS is intended as a development tool for program writing, and is not designed to for high execution speed. Use of the standard IFS image access functions (such as ifsigp) is in fact quite slow in some operations. It is efficient when the aim is to write and test programs quickly and easily. IFS does provide, however, access to pointers and data types which allow very efficient software to be written while retaining the IFS image structure.

This document gives some very brief instructions on the use of IFS, and provides documentation for the individual IFS functions. Any questions not answered here need to be directed to the author(s). Questions should be sent to:


```
Rosalyn Snyder
Communication Unlimited, Inc.
3603 Octavia St.
Raleigh, NC 27606
```

1.0.1 New with Version 5

- Support for X-windows. IFS images may now be displayed on any X-windows device. The graphics support automatically determines the dynamic range of the graphics device (many X-windows devices are binary, for example), and either thresholds or dithers, at the user's command.

- CPU independent code. Various computers use different conventions for storage of bytes within a word, necessitating byte-swapping when one machine reads a file written on another. Furthermore, depending on the computer, byte swapping may be required on 16 bit or 32 bit boundaries, or both. Finally, there are at least two different conventions for floating point data, DEC and IEEE (which must still be corrected after byte swapping) IFS Version 5 automatically determines what type of machine it is running on and determines what type of machine wrote the input file. Should the input file be incompatible, the IFS file read routine automatically performs all data conversions.

1.1 Using IFS

1.1.1 Program Compilation and Linking

In order to use IFS, the user needs to link his programs to the IFS library at compilation time. To specify these libraries, one would use the switches *-lifs* on the *cc* or *ld* command. For example, a typical command to compile a program would look like:

```
cc -g test.c -o test -lifs
```

To actually make use of the IFS functions, the user's program also needs to *#include* a header file or two to define various structures used by the IFS routines. All programs which use IFS should include the files *¡stdio.h¿* and $< ifs.h >$. Most IFS routines will return error codes through an external (global) variable called ifserr; if the user plans to examine these error codes he should also include the header file $< ifserr.h >$. This file defines a set of symbolic constants which one may use rather than using actual values for codes. It is not wise to use actual values in place of these constants when writing programs as the definitions for the constants may change from one release of IFS to the next.

1.2 Referencing IFS images

All IFS *images* include a *header* which contains various items of information about the image, such as the number of points in the image, the number of dimensions for the image, the data format, and so on. Also associated with the image is the actual *data* for the image. The image header includes a pointer to the image data. The user manipulates an image by calling some function in the IFS library; one of the arguments to the function will be the address of the header for the image. The functions will automatically figure out where the data is and how to access it from the information in the header. In addition to handling the work of accessing data in images, the IFS routines automatically take care of allocating space in memory to store data and headers. Everything is totally dynamic in operation; there are no fixed-dimension arrays needed. This relieves the user of the difficulties involved with accessing data in arrays (when using *C*), when the arrays are not of some fixed size.

The header structure for an image is defined in the file $< ifs.h >$, and is known by the name IFSHDR. To manipulate an image, the user merely needs to declare a pointer to an image header structure (as IFSHDR * your_image; or IFSIMG your_image;). Then, the user simply calls some IFS function to create a new image, and sets the pointer to the value returned from that function. Some typical programs are given in the examples in Figures 1.1 through 1.5.

```
#include <stdio.h>
#include <ifs.h>
main(  )
{
IFSIMG img1, img2; /* Declare pointers to headers     */
int len[3]; /* len is an array of dimensions, used by ifscreate */
int threshold; /* threshold is an int here     */
int row,col; /* counters     */
int v;
img1 = ifspin("infile.ifs"); /* read in file by this name     */
len[0] = 2; /* image to be created is two dimensional     */
len[1] = 128; /* image has 128 columns     */
len[2] = 128; /* image has 128 rows     */
img2 = ifscreate("u8bit",len,IFS_CR_ALL,0);  /* image is unsigned 8 bit
*/
threshold = 55; /* set some value to threshold     */
for( row = 0; row < 128; row++)
for( col = 0; col < 128; col++)
    {
    v = ifsigp(img1,row,col); /* read a pixel as an int */
    if (v > threshold)
ifsipp(img2,row,col,255);
    else
ifsipp(img2,row,col,0);
    }
ifspot(img2, "img2.ifs");        /* write image 2 to disk   */
}
```

Figure 1.1: Example IFS program to threshold an image using fixed values of dimensions and defined data type

1.2.1 Naming conventions in IFS

All of the IFS functions have names which begin with the letters ifs, so users should have no problems avoiding conflicts when naming their own functions. Also, all external variables or *#define*'d constants also begin with the letters "ifs" or "IFS". Originally, all IFS routines had names which were limited to 6 characters in an effort to improve compatibility between different compilers. Unfortunately, with three of the letters already being fixed (as "ifs"), this doesn't leave much left to create meaningful function names with. Hence, many IFS functions have rather cryptic names. Later versions of IFS have relaxed this restriction, so that newer functions have longer and more descriptive names.

Starting with release 3.0 of IFS, all of the IFS functions also have version numbers built into them. These 'version numbers' are actually printable strings which are globally accessible. These strings usually contain the function's name, a version number, and the date of the last modification to the function. Other items of information may occasionally also be contained in the string. If an IFS function has a name *ifsXXXX*, where *XXXX* is just some stem naming the function, then the string which gives its version number will have the name *ifsv_XXXX*. For instance, if a user wanted to know what version of the function *ifscreate* was in his IFS library, he could include the statements

```
extern char * ifsv_create;

printf("%s\n",ifsv_create);
```

somewhere within his program. Also, the function *ifsversion* can be used to print out the version strings of all functions in the IFS library.

1.3 Error handling in IFS

IFS provides various levels of error checking. When an error occurs, an IFS function usually returns some sort of error flag. IFS also has two external (global) variables which relate to error handling. The first one is known as *ifserr*, and is set to error code(s) which the user may examine to help determine what went wrong. The second one is IFSSLV (for "IFS Severity Level"), which affects the action IFS takes upon detecting an error. Both of these variables are declared as "extern int" variables in the header file $< ifs.h >$, so it is not necessary for the user to declare them.

The various error codes which may be returned are defined in the header file $< ifserr.h >$, which the user should make sure to include in his program if he plans on using *ifserr*. These error codes are described in detail in Section 1.5. Scrutinizing the file $< ifserr.h >$ may also prove useful. The error codes are

```
#include <stdio.h>
#include <ifs.h>
main(  )
{
IFSIMG img1, img2; /* Declare pointers to headers     */
int *len; /* len is an array of dimensions, used by ifscreate */
int frame,row,col; /* counters     */
float threshold,v; /* threshold is a float here     */
img1 = ifspin("infile.ifs"); /* read in file by this name     */
len = ifssiz(img1); /* get dimensions of input image     */
/* ifssiz returns a pointer to an array of dimensions*/

img2 = ifscreate(img1->ifsdt,len,IFS_CR_ALL,0);
/* output image is to be   */
 /* same type as input     */
threshold = 55; /* set some value to threshold     */

/* check for one, two or three dimensions */
switch(len[0]){
    case 1: /* 1d signal */
    for( col = 0; col < len[1]; col++)
{
v = ifsfgp(img1,0,col); /* read a pixel as a float */
if (v > threshold)
ifsfpp(img2,0,col,255.0); /* write a float  */
    /* if img2 not float, will be converted*/
else
ifsfpp(img2,0,col,0.0);
}
break;

    case 2: /* 2d picture */
    for( row = 0; row < len[2]; row++)
    for( col = 0; col < len[1]; col++)
{
v = ifsfgp(img1,row,col);   /* read a pixel as a float */
if (v > threshold)
    ifsfpp(img2,row,col,255.0);/* store a float */
else
    ifsfpp(img2,row,col,0.0);
}
    break;
```

Figure 1.2: First portion of example, see Figure 1.3 for remainder

```
    case 3: /* 3d volume */
    for(frame = 0; frame < len[3];frame++)
    for( row = 0; row  < len[2]; row++)
    for( col = 0; col  < len[1]; col++)
{
v = ifsfgp3d(img1,frame,row,col);   /* read a pixel as a float */
if (v > threshold)
    ifsfpp3d(img2,frame,row,col,255.0);
else
    ifsfpp3d(img2,frame,row,col,0.0);
}
    break;
    default: printf("Sorry I cant do 4 or more dimensions\n");
    }
ifspot(img2, "img2.ifs");        /* write image 2 to disk   */
}
```

Figure 1.3: Example IFS program to threshold an image using number of dimensions, size of dimensions, and data type determined by the input image

```
#include <stdio.h>
#include <ifs.h>
main(  )
{
IFSIMG  img1, img2; /* Declare pointers to headers */
int len[3]; /* len is an array of dimensions, used by ifscreate */
int size; /* number of bytes in image */
int threshold; /* threshold is an int here */
register int count; /* number of pixels in image */
register unsigned char *ptri,*ptro;
img1 = ifspin(""); /* read in file; prompt user for name  */
len[0] = 2; /* image to be created is two dimensional */
len[1] = ifsdimen(img1,0); /* get  columns of input  */
len[2] = ifsdimen(img1,1); /* get  rows    of input  */
img2 = ifscreate("u8bit",len,IFS_CR_ALL,0);  /* image is unsigned 8 bit */
threshold = 55; /* set some value to threshold */
ptri = (unsigned char *)img1->ifsptr; /* get address of input data */
ptro = (unsigned char *)img2->ifsptr; /* get address of output data */
size = len[1] * len[2]; /* compute number of pixels */
for( count = 0; count  < size; count++)
    {
    if( *ptri++ > threshold)
*ptro = 255;
    else
*ptro = 0;
    ptro++;
    }
ifspot(img2, "");        /* write image 2 to disk, prompt user for   */
/* file name      */
}
```

Figure 1.4: Example IFS program to threshold an image using two dimensions, size of dimensions determined by input file, and defined data type of unsigned char on both files. Pointers are used for speed.

```
#include <stdio.h>
#include <ifs.h>
#include <ifserr.h>   /* optional */
...
main(  )
{
IFSHDR * img1, * img2; /* Declare pointers to headers */
       .
       .
       .
img1 = ifsmkh(128, 128, "char");  /* make a 128*128 2-d image   */
  /* Space for data & header
     are automatically allocated     */

  .... manipulate image 1 .....

img2 = ifs_exwin(img1,10,10,100,75);  /* extract a sub-image    */
 /* of the original image (img1) and call it img2 */

ifspot(img1, "img1.ifs");        /* write image 1 to disk   */
```

Figure 1.5: Example IFS program to extract a sub-image

indicated by individual bits in *ifserr*, so it is actually possible for several error flags to be set simultaneously. Also, some error codes are actually combinations of other codes. For instance, the codes IFSE_NO_OPEN and IFSE_NOT_IMAGE are two possible errors which may occur when trying to read or write IFS images. If the user checks for the condition IFSE_IO_ERR, he has automatically tested for both of the errors IFSE_NO_OPEN and IFSE_NOT_IMAGE. The way to test for such error codes is with the bitwise logical AND operator, rather than with a comparison. I.e.:

```
if (ifserr & IFSE_IO_ERR) ...
```

is preferable to:

```
if (ifserr == IFSE_IO_ERR) ...
```

because in this way, more than one bit may be tested, or just a single bit.

The second global variable, IFSSLV, allows the user to specify what action to take when an error occurs. Currently, there are three possible courses of action to take upon an error; these are chosen by setting IFSSLV to some "severity level" code. The three severity levels are represented by the constants IFS_QUIET, IFS_WARN, and IFS_FATAL (which are defined in $< ifs.h >$). These affect the action taken upon the occurrence of an error as follows:

IFS_QUIET Do not print out any error messages to the user. The function just returns an error code to the calling routine. The user must make sure to watch out for this code, and act accordingly. If the error is not handled, the program will probably crash.

IFS_WARN If an error occurs, print out some message describing the error (to *stderr*). The routine also returns the appropriate error code. This allows the user to know what is going on, but still allows the program to trap errors. IFS_WARN is probably the recommended severity level for most applications, and is the default value for IFSSLV.

IFS_FATAL If an error occurs, print out an error message, and abort the program. This is not an exceedingly user-friendly option, but is probably better than the perennial Unix favorite "bus error: core dumped".

1.3.1 Image validation in IFS

Most IFS functions will double-check the header of an image before attempting to perform some operation on the image. This is done to verify that the argument the user passed to the function legitimately points to an IFS image, and does not just represent some random value. The most likely source for such an error would be insufficient error checking in a user's program, when the severity level variable IFSSLV was set to some value other than IFS_FATAL. For instance, a section of code such as

```
img = ifsmkh(nrows, ncols, "ubyte");
ifsipp(img, 10, 20, 255);
```

(which attempts to create an image and set the pixel at location 10,20 to a value of 255) could be a potential source for an error, if *ifsmkh* had been incapable of creating the image as requested. It would then have returned the value NULL, which would be passed to the function *ifsipp*. If *ifsipp* did not check the header, it would blindly attempt to use NULL as a pointer to an image header, which would probably crash the user's program.

The problem with this error checking is that it takes time to perform. If an image was 100 by 100 pixels in size, and the routine *ifsipp* was used to set the value of each pixel in the image, then the header would end up being checked 10000 times! For a program which accesses an image(s) heavily, this header checking overhead takes a significant amount of time. Timing analyses on sample programs have shown that it is possible for 30% of the CPU time used by a program to be spent in the header checking operation.

The user may disable the header checking operation in some IFS routines. This, of course, places upon the programmer the responsibility to perform more extensive error checking operations, if robust code is desired. If the user sets the external integer variable IFSCHK to zero, then certain routines will cease to check image headers. Header checking can be re-enabled by setting IFSCHK to any non-zero value. Note that not all of the IFS routines are affected by IFSCHK. Generally, only those routines which are called with high frequency, and for which the header checking represents a significant fraction of the execution time for that function, will be affected by this variable. Incidentally, it is not necessary for the user's program to declare IFSCHK. As with IFSSLV and ifserr, this is declared in the header file $< ifs.h >$.

1.3.2 Coordinate systems and array storage in IFS

IFS stores arrays in the same manner that *C* normally does. As with *C*, the indices for arrays start with zero rather than one. For example, if you create an image with 30 rows and 20 columns, then valid row indices for that function range from 0 to 29, and column indices may go from 0 to 19.

One common source of confusion is the usage of the terms *row* and *column* to denote array subscripts when working with 2-dimensional arrays. It is quite typical for a user's program to view the coordinate system in terms of an x and a y axis. The intent in IFS is that the *column* axis represents the horizontal axis, and the *row* axis is the vertical. Hence the width of the image is equivalent to the number of columns in the image, and height is the number of rows. It is common usage that the x axis is the horizontal axis, hence a *column* coordinate is synonymous with an x coordinate. If this is the coordinate system you normally use, beware of the

temptation to write code of the form:

```
int x, y;
...
for(x=0;x<width;x++)
   for(y=0;y<height;y++)
      ifsipp(img, x, y, value);
...
```

The correct code in this case should be:

```
int x, y;
...
for(x=0;x<width;x++)
   for(y=0;y<height;y++)
      ifsipp(img, y, x, value);
...
```

Unfortunately, people have a tendency to write coordinate pairs as (x ,y) or (*row*,*column*), but these two are not synonymous if you interpret x , y , *row*, and *column* in the manner described above.

A second problem occurs when displaying images on graphics output devices. There is no set standard as to where the origin of the coordinate system is among graphics displays. It is probably most common that the origin is in the upper left corner of the display, and moving in the positive direction along the *column* axis moves you to the right, and moving in the positive direction along the *row* axis moves you downwards. Some devices place the origin in the lower left corner of the screen, and moving in the positive *row* direction moves you towards the top of the display. The positive *column* direction still usually is to the right. This also corresponds to the way most people label axes when hand-drawing a graph. The net affect here is that images displayed in this coordinate system will be upside-down as compared to the first type of system. To further confound the issue, many programs which plot on printers reverse the meaning of the x and y axes, so that images plotted in this manner are rotated by 90 degrees in one direction or another.

The point of these warnings about display coordinate systems is that IFS knows nothing about the nature of the user's display mechanism. There is no specific "up", "down", "left", or "right". The user should not be too surprised if an image appears flipped or rotated from what was expected.

1.4 IFS FUNCTION LISTING

This section lists all of the functions in the IFS library, in alphabetical order. The convention used to describe the syntax for the function is:

```
return_value = function_name(arg1, arg2, ....);
typeof return_value;
typeof arg1;
typeof arg2; ....
```

where "typeof" denotes a **C** variable type (such as "int", "float", "char *", or "IFSHDR *", or other TYPEDEF's or STRUCT's). For instance, the sample description

```
p = ifsalc(numbytes);

char * p;
int numbytes;
```

indicates that the function *ifsalc* returns a pointer to a character, and that it takes one argument, which is an integer.

1.4.1 ifsalc

ifsalc – allocate storage (memory)

```
cptr = ifsalc(NumBytes);
char * cptr;
int NumBytes;
```

Ifalc is an IFS function used to allocate storage in main memory – such as for storing arrays and image headers. The storage will be initialized to all zeroes. It is essentially just a call to the system function *calloc*; the only difference being that **ifsalc** performs a small amount of error checking. If the system can not allocate the requested amount of storage, then **ifsalc** will return the value NULL, and the external variable *ifserr* will be set to the value IFSE_NO_MEM. If the external variable IFSSLV is not set to the value IFS_QUIET, then **ifsalc** will write an error message to *stderr* if it can't allocate the requested space. If IFSSLV is set to IFS_FATAL, then **ifsalc** will also abort your program upon an error.

1.4.2 ifscigp

ifscigp – get pixel value from a 2-d (possibly complex) image
Usage:

```
val = int ifscigp(ptri,row,col)]{}

    IFSHDR *ptri;  /*pointer to image header structure
    int row,col,val;   /*coordinates (in pixels) of pixel to examine.*/
```

Ifscigp returns (as an int) the value of the pixel at a specified coordinate in a 2-d image. If image is "complex" format, returns the imaginary portion of the number.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the pixel value won't fit in an "int" (for example, a large number in a "float" or "complex" image), then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.3 ifscigp3d

ifscigp3d – get pixel value from a 3-d image

Usage:

```
val = int ifscigp3d(ptri,frame,row,col);

IFSHDR *ptri;  /* pointer to image header structure*/

int frame,row,col; /*coordinates (in pixels) of pixel
                                    to examine.*/
```

Ifscigp3d returns (as an integer) the value of the pixel at a specified coordinate in a 3-d image. If image is "complex" format, returns the imaginary portion of the number (assuming it can be converted to an int).

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the pixel value won't fit in an "int" (for example, a large number in a "float" or "complex" image), then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.4 ifscipp

ifscipp – set pixel value in a 2-d image
Usage:

```
status =  ifscipp(ptri,x,y, val);

IFSHDR *ptri;   /*pointer to image header structure*/

int x,y;  /*coordinates (in pixels) of pixel to examine.*/
int val;
int status; /*return status*/
```

Returns:

IFS_SUCCESS or IFS_FAILURE

Ifscipp sets the value of the pixel at a specified coordinate in a 2-d image, where the input is a int. If image is "complex" format, stuffs the imaginary portion of the number, and DOES NOT set the real part to zero.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the value stuffed won't fit in the output image datatype, then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.5 ifscipp3d

ifscipp3d – set pixel in a 3-d image

Usage:

```
status =  ifscipp3d(ptri,frame,row,col,val);
IFSHDR *ptri;

int frame,row,col; /*coordinates (in pixels) of pixel
to examine.*/
int status; /*return status*/
```

Returns:

IFS_SUCCESS or IFS_FAILURE

Ifscipp3d sets the value of the pixel at a specified coordinate in a 3-d image, where the input is integer (byte, ubyte, etc). If image is "complex" format, stuffs the imaginary portion of the number, and sets the real part to zero.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the value stuffed won't fit in the output image datatype, then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.6 ifscfgp

ifscfgp – get value of a pixel in a 2-d image.

```
\item [{\em Usage:}]{}

    val = (double) ifscfgp(ptri,row,col);
    IFSHDR *ptri;

    int row,col; /*coordinates (in pixels) of pixel
    to examine.*/
```

Ifscfgp returns (as a float) the value of the pixel at a specified coordinate in a 2-d image. If image is "complex" format, returns the imaginary portion of the number.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the pixel value won't fit in a "double" results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error. There could be possible round off errors.

1.4.7 ifscfgp3d

ifscfgp3d – get value of a pixel in a 3-d image
Usage:

```
val = (double) ifscfgp3d(ptri,frame,row,col);
    IFSHDR *ptri;
    int frame,row,col; /*coordinates (in pixels) of pixel
    to examine.*/
```

ifscfgp3d returns (as a double) the value of the pixel at a specified coordinate in a 3-d image. If image is "complex" format, returns the imaginary portion of the number.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the pixel value won't fit in a "double" results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error. There could be possible round off errors.

1.4.8 ifscfpp

ifscfpp – set value of a pixel in a 2-d image.

Usage:

```
status = ifscfpp(ptri,x,y, val);
IFSHDR *ptri;
int x,y; /* coordinates (in pixels) of pixel to examine.*/

double val; /*the value to stuff.*/
int status;
```

Returns:

IFS_SUCCESS or IFS_FAILURE

Ifscfpp sets the value of the pixel at a specified coordinate in a 2-d image, where the input is a float. If image is "complex" format, stuffs the imaginary portion of the number, and sets the real part to zero.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the value stuffed won't fit in the output image datatype, then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.9 ifscfpp3d

ifscfpp3d – sets the value of a pixel in a 3-d image (This is a completely new version of ifsfpp which handles 3-d images.)

Usage:

```
status =  ifscfpp3d(ptri,frame,row,col, val);
IFSHDR *ptri;

int frame,row,col  /* coordinates (in pixels) of pixel
                 to examine.*/

double val; /*the value to stuff.*/
int status;
```

Returns:

IFS_SUCCESS or IFS_FAILURE

Ifscfpp3d sets the value of the pixel at a specified coordinate in a 3-d image, where the input is a float. If image is "complex" format, stuffs the imaginary portion of the number, and sets the real part to zero.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the value stuffed won't fit in the output image datatype, then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.10 Ifsclose

ifsclose – close an open file

```
rc = ifsClose(File);
    FILE * File;
    int rc;
```

Ifsclose is identical to the standard I/O library function *fclose*, except that it will avoid closing *File* if *File* corresponds to *stdin, stdout,* or *stderr.* If *File* is NULL, **ifsclose** returns -1, else it just returns whatever value *fclose* would return.

Ifsclose is supplied as a complement to *ifsOpen* since the latter function may return *stdin* or *stdout* in some circumstances, and the user typically does not want to close these files.

1.4.11 ifscreate

ifscreate – create an IFS image

```
img = ifscreate(type,len,flags,structsize);
    IFSHDR * img;
    char * type;
    int len[];
    int flags;
    int structsize;
```

Ifscreate is used to create a new IFS image or image header. Space for the header is automatically allocated, and a pointer to the header is returned. Various fields in the header structure will be set to default values. Space for the actual data may also be allocated, depending on the value of the *flags* variable. If space for the data array is allocated, it will be filled with zeros. If the image can not be created, **ifscreate** returns the value NULL, and the external variable *ifserr* will be set to some error code, as given in the *#include* file $< ifserr.h >$. The image as created will not have any "tail" structure associated with it.

The arguments to **ifscreate** are:

type The data format for individual pixels, such as "byte" or "Ddouble". The valid data types are listed in a later section of this manual. If the data type is not recognized by IFS, then **ifscreate** will return NULL, and *ifserr* will be set to the code IFSE_BAD_DTYPE.

len An *n+1*-length integer array – the first element (*len[0]*) gives the number of dimensions for the image, the remaining elements give the length for each dimension of the image being created. This is in exactly the same format as the arrays returned by the function *ifssiz*. The lengths are given in terms of ascending *rank* for the image. Images are stored in standard *C* storage order: the *column* or *x* index changes most rapidly when scanning through memory, hence this dimension has rank 1. The *row* or *y* index has rank 2, the *frame* or *z* index has rank 3, and so on. I.e., the second element of the array (*len[1]*) gives the number of columns of the image, *len[2]* is the number of rows, etc.

flags The various bits of this argument determine precisely what is and is not allocated when generating the image. If *flags* = IFS_CR_ALL or IFS_CR_DATA, then storage space for the image is allocated, as well as storage for the image header. In this case the field img→ifsptr points to the data storage. If *flags* = IFS_CR_HDR then only space for the image header is allocated. The field img→ifsptr will be set to NULL. The user must supply an array to store the image in, and set img→ifsptr to point to this array. *Note:* version 4.0

and 5.0 of **ifscreate** will ALWAYS allocate space for the image header; the flag IFS_CR_HDR is not really examined, and is only intended for possible future expansion. All that is really checked is the IFS_CR_DATA bit. The flag IFS_CR_ALL is the combination of IFS_CR_DATA and IFS_CR_HDR and is probably the best flag to use if one wants data space allocated.

structsize This argument is only needed if *type* is "struct", in which case it gives the size of a single data element (structure) in bytes. If *type* is not "struct" this argument may be omitted or set to 0.

Example:

```
/* Create a 2-d image with 20 rows and 30 columns */
/* and a 1-d array of 10 structures. */
#include <ifs.h>
main()
{
IFSHDR * img, * strimg;
int lengths[3];
IFSHDR * ifscreate();
typedef struct { int red; int green; int blue; } RGB;
 .
 .
 .
/* create 2D byte array */
lengths[0] = 2;    /* Image will be 2D */
lengths[1] = 30;   /* Number of columns (width; x-dimension) */
lengths[2] = 20;   /* Number of rows (height; y-dimension)   */
img = ifscreate("ubyte", lengths, IFS_CR_ALL);
if (img == NULL) { /* error processing code */ }
 .
 .
 .
/* create 1D structure array */
lengths[0] = 1;
lengths[1] = 10;
strimg = ifscreate("struct",lengths,IFS_CR_ALL,sizeof(RGB));
 .
 .
 .
}
```

1.4.12 ifsdimen

ifsdimen – get size of dimension or image data

```
len = ifsdimen(image, n);
int len;
IFSHDR * image;
int n;
```

Ifsdimen returns the length (number of elements) of the *n*th dimension of *image*. It also may be used to get the total number of elements or bytes required by the data section of an image. The argument *n* is the rank of the dimension being queried, i.e., *ifsdimen(img,0)* is the number of columns, *ifsdimen(img,1)* is the number of rows, and so on.

If *n* is specified as -1, **ifsdimen** returns the total number of elements in the image (the product of all the individual dimension lengths). If *n* is specified as -2, **ifsdimen** returns the total number of bytes occupied by the image data, i.e., the total number of elements times the size in bytes for a single element.

If there is some error, **ifsdimen** returns zero and sets the external variable *ifserr* appropriately. Possible error conditions are IFSE_BAD_HEADER or IFSE_NULL_HEADER for invalid images, or IFSE_WRONG_NDIM if *n* is invalid (such as asking for the number of frames for a 2D image).

1.4.13 ifsexwin

ifsexwin – Extract a window from an image

```
#include <ifs.h>
new = ifsexwin(old, r1,c1, r2, c2);

    IFSHDR * new, * old;
    int r1, c1, r2, c2;
```

Ifsexwin is used to create a new image which is a subimage of some old image. The old image must be a two-dimensional image. The arguments *r1,c1* and *r2,c2* give the row and column positions of the corners of a box which defines the region to be extracted. These corners must be on opposite ends of a diagonal for the window. It does not matter which corners are chosen for each point, as long as as they are on opposite ends of a box diagonal. The region extracted includes the area of the bounding box itself, ie, is inclusive of the rows *r1, r2* and columns *c1, c2*.

Ifsexwin returns a pointer to the newly created image, or NULL if some error occurred. In the latter case, the external variable *ifserr* will be set to indicate the nature of the error. Possibilities are:

IFSE_BAD_HDR If the pointer *old* does not point to a valid IFS image.

IFSE_NO_MEM If space couldn't be allocated for the new image.

IFSE_WRONG_NDIM If the original image is not two-dimensional.

IFSE_BAD_POS If either of the box coordinates is outside the image dimensions.

The dimensionality of windowed image is consistent. That is, a 1-d/2-d slice (a 3-d image one voxel thick in one or more dimensions) returns with a header consistent with the actual dimensionality.

1.4.14 ifsexwin3d

ifsexwin3d – Extract a window from an image
ifsexwin3d – extract window from 3-d image

Usage:

new_img = ifsexwin3d(old_img, f1, r1, c1, f2, r2, c2)

where f1,r1,c1 and f2,r2,c2 are the coordinates (frame,row,col) of one corner of the box and the opposite (diagonal) corner. It doesn't matter which corners are chosen.The box which is extracted includes the bordering surface (i.e, coordinates are f1,r1,c1 to f2,r2,c2 INCLUSIVE).

Returns:

This function returns NULL if an error occurs, and returns an error code thru the external variable 'ifserr'.

External variables:

ifserr, IFSSLV

Ifsexwin3d extracts a piece (window) out of a 3-d IFS image, to make a new IFS image. The data type of the new image is identical to that of the old one. The dimensionality of windowed image is consistent. That is, a 1-d/2-d slice (a 3-d image one voxel thick in one or more dimensions) returns with a header consistent with the actual dimensionality.

1.4.15 ifsfgp

ifsfgp – get pixel from a 2-D image

```
value = ifsfgp(img,row,col);

    double value;
    int row, col;
    IFSHDR * img;
```

Ifsfgp is used to get the value of some pixel in a 2-dimensional image. The value returned is of type *double*, regardless of what the data format of the image is. Otherwise, **ifsfgp** is identical to the function *ifsigp*, in all respects. See the documentation for *ifsigp* for more details.

1.4.16 ifsfgp3d

ifsfgp3d – gets the value of a pixel in a 3-d image (A generic multidimensional fgp can be attempted thru variable parameter passing, but that would make the code unportable.)

Usage:

```
val = ifsfgp3d(ptri,frame,row,col);

    IFSHDR *ptri; /* pointer to image header structure*/

    int frame,row,col;/* coordinates (in pixels)
    of pixel to examine.*/
    double val;
```

Ifsfgp3d returns (as a double) the value of the pixel at a specified coordinate in a 3-d image. If image is "complex" format, returns the real portion of the number.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the pixel value won't fit in a "double" then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error. Round off error can occur in conversions. e.g. int to double typecasting.

ifsfpp3d – set the value of a pixel in a 3-d image
Usage:

```
status =  ifsfpp3d(ptri,frame,row,col, val);
    IFSHDR *ptri  /* pointer to image header structure*/

    int frame,row,col; /*coordinates (in pixels) of pixel
     to examine.*/

    double val; /*the value to stuff*/
    int status;
```

Returns:

IFS_SUCCESS or IFS_FAILURE

Ifsfpp3d sets the value of the pixel at a specified coordinate in a 3-d image, where the input is a float. If image is "complex" format, stuffs the real portion of the number, and sets the imaginary part to zero.

Known bugs, special notes:

@ This routine does not check to see if the specified coordinates actually are in bounds.

@ If the value stuffed won't fit in the output image datatype, then results are undefined. Maybe you get garbage, maybe your program aborts on an overflow type of error.

1.4.17 ifsfpp

ifsfpp – put pixel value into a 2-D image

```
status = ifsfpp(img,row,col,value);

    int status;
    double value;
    int row, col;
    IFSHDR * img;
```

Ifsfpp is used to set the value of some pixel in a 2-dimensional image. The argument *value* is automatically converted from a floating point number (*float* or *double*) into whatever data format the image is in. In all other respects, **ifsfpp** is identical to the function *ifsipp*. See the documentation for *ifsipp* for more details.

1.4.18 ifsfree

ifsfree – delete (deallocate) an IFS image

```
img = ifsfree(img, flags);

    IFSHDR * img;
    int flags;
```

Ifsfree is used to get rid of an IFS image which is no longer in use. The space for the header and/or data is deallocated, and returned to the operating system for other use. Basically, **ifsfree** just consists of several calls to the system function *cfree*.

The arguments to **ifsfree** are:

img A pointer to the image header structure.

flags A set of flags which indicates exactly what is to be deallocated.

Possibilities for the flags are:

IFS_FR_DATA If this flag is set, then **ifsfree** will deallocate the space allocated for the storage of the actual image data (if there is any), and the header field img→ifsptr will be set to NULL to show that the header no longer has any data associated with it. If there is no data associated with the header, then this flag has no effect. This will not cause any errors.

IFS_FR_HDR If this flag is set, then **ifsfree** will deallocate the space allotted for the image header. The data space is left intact. This is usually only going to be used if the user supplied his own data area for the image (perhaps a static array or somesuch).

IFS_FR_ALL If this flag is specified, then **ifsfree** will free up everything – image header and data. IFS_FR_ALL is just the combination of IFS_FR_DATA and IFS_FR_HDR. This is probably the normal flag to set when calling **ifsfree** .

Ifsfree returns a pointer to the image header as it should be AFTER the desired things have been deallocated. If only the IFS_FR_DATA flag was specified, then **ifsfree** returns the original pointer value *img*, with the field img→ifsptr now set to NULL to show that the data array has been deleted. If the header structure was freed, then **ifsfree** returns NULL to indicate that the pointer is no longer valid. Hence it is good practice to assign the return value from **ifsfree** back to the original pointer value *img*. It is not an error to simply say ifsfree(img,IFS_FR_ALL) rather than img = ifsfree(img,IFS_FR_ALL) to get rid of an image, but the latter usage is preferable in that it will make it more obvious to any subsequent routines called

(erroneously) with the argument *img*. **Ifsfree** will also return the value NULL if an error occurred. In this case, the external variable *ifserr* will be set to the appropriate error code. Possible error conditions are:

IFSE_NULL_HDR This indicates that you passed the pointer NULL for the argument *img*.

IFSE_BAD_HDR This indicates that the pointer *img* does not reference a valid IFS image structure. Note that the error IFSE_NULL_HDR is actually a subclass of the error IFSE_BAD_HDR, so if you test the value "(ifserr & IFSE_BAD_HDR)", you will automatically also pick up errors of the type IFSE_NULL_HDR.

BUGS:

Trying to deallocate something which was not originally obtained using some standard system memory-allocation function (e.g, *calloc*, or the IFS routines *ifsalc* or *ifscreate*) will cause grave errors – usually a program crash. This is a problem of the system allocate/deallocate routines and not **ifsfree**.

1.4.19 ifsGetFN

ifsGetFN – read in a filename and expand it

```
FileName = ifsGetFN(Prompt, Input);

    char * FileName;
    char * Prompt;
    FILE * Input;
```

IfsGetFN will read in a string from the file *Input* (typically *stdin*), and expand it using the function *ifsPrsFN*. It returns a pointer to the name it read, or NULL if it failed. Space for the filename is dynamically allocated and may be freed (using *cfree*) when the user is through with it.

If *Input* is a terminal, and *Prompt* is not NULL, then *Prompt* will be printed before the filename is read in.

IfsGetFN normally delimits filenames with any control character or whitespace character and strips off any leading whitespace characters supplied in the name. Any character (including whitespace characters) may be put in a filename by prefixing it with a backslash \. This applies to leading whitespace characters as well as whitespace characters in the middle or end of the name.

1.4.20 ifsGetImg

ifsGetImg – open a file and read an IFS image from it

```
img = ifsGetImg(FileName, Prompt, ReadTail);

    IFSHDR * img;
    char * FileName;
    char * Prompt;
    int ReadTail;
```

IfsGetImg reads an IFS image from some file *FileName*. If *ReadTail* is false (zero), then any tail information associated with the image will not be read in. It returns a pointer to the new image header, or NULL if it failed, in which case the external integer variables *ifserr* and *column* can be examined to determine the nature of the error. Space for the header, data, and tail is allocated dynamically, each may be freed (using *cfree* or *ifsfree*) when the user is through with it.

If *FileName* is NULL, then the input image is read from *stdin*. If *FileName* is a null string (that is, *FileName* = ""), then a filename will be read in from *stdin* using the routine *ifsGetFN*. In this case, if *Prompt* isn't NULL and *stdin* corresponds to a terminal, then the string *Prompt* will be printed on the terminal (actually, to *stderr*) before reading a name. If *stdin* is not attached to a terminal, such as when input is being piped in from another program, then the printing of the prompt string is suppressed. If a filename is read in from *stdin*, and it is "-" (a single dash), then the image itself will be read from *stdin*. Filenames are expanded using *ifsPrsFN*, so they may contain such things as environment variable names and "~login_id" constructs.

IfsGetImg works by opening the specified file and then calling *ifsRdImg* to do the actual work of getting the image. It then closes the file when it's done (unless it read from *stdin*). Note that calling **ifsGetImg** with *Filename* = NULL is essentially the same as calling *ifsRdImg* directly.

The complimentary routine to **ifsGetImg** is *ifsPutImg*.

1.4.21 ifsigp

ifsigp – get pixel from a 2-D image

```
value = ifsigp(img,row,col);
    int value;
    int row, col;
    IFSHDR * img;
```

Ifsigp is used to get the value of some pixel in a 2-dimensional image. The value returned is of type *int*, regardless of what the data format of the image is. **Ifsigp** performs all necessary type conversions. If the value of the pixel in the image will not fit into an *int* data type, then the value that is returned will be meaningless. If the image data format is one of the *complex* forms, then **ifsigp** returns the real part of the specified data point. If some sort of error occurs, then **ifsigp** will return zero, and the external variable *ifserr* will be set to indicate the nature of the error.

The arguments to **ifsigp** are:

img A pointer to the image header structure. This should refer to a 2-dimensional image. If the image has 3 or more dimensions, then **ifsigp** will access the first frame of the data (ie, all indices besides the first two will simply be treated as zero).

row,col The coordinates of the point to be examined. Note that *row, col* may also be regarded as a *y, x* pair. Beware that *row* corresponds to the *y* index, not the *x* index.

The following error codes (defined in the #include file $< ifserr.h >$) may be returned by **ifsigp** :

IFSE_BAD_HEADER: The pointer *img* does not point to an actual IFS image.

IFSE_BAD_DTYPE: The image is of some data type that **ifsigp** does not recognize. Usually this indicates that your header has been damaged, and the field $img \rightarrow ifsdt$ is mangled, or the image data type is "struct". It could also occur if someone added a new data type to that understood by IFS, and forgot to modify **ifsigp** accordingly.

BUGS:

- **Ifsigp** does not verify that the image passed to it corresponds to a 2-dimensional image.
- The indices *row, col* are not checked to verify that they lie inside the image dimensions.

- **Ifsigp** does not check to make sure that the data pointer *img-¿ifsptr* is not NULL. Previous versions of IFS did not allow this data pointer to be NULL, so it was not previously necessary to check for this.

- Results when numeric overflow occur (as is possible when converting a floating point number into an integer) are undefined.

Any of the above bugs could cause an abrupt and unpleasant termination of your program, generally with the infamous "bus error: core dumped" message under UNIX systems. Of course, such a crash would be indicative of some prior error in the user's program not having been caught.

1.4.22 ifsipp

ifsipp – put pixel value into a 2-D image

```
status = ifsipp(img,row,col,value);

    int status;
    int value;
    int row, col;
    IFSHDR * img;
```

Ifsipp is used to set the value of some pixel in a 2-dimensional image. The argument *value* is automatically converted from an integer into whatever data format the image is in. If the image is of type *complex*, then **ifsipp** sets the real part of the datum to *value*, and the imaginary portion to zero. **Ifsipp** returns the value IFS_SUCCESS if it succeeded, otherwise it returns the value IFS_FAILURE and sets the external variable *ifserr* to the appropriate error code.

The arguments to **ifsipp** are:

img A pointer to the image header structure. This should refer to a 2-dimensional image. If the image has 3 or more dimensions, then **ifsipp** will access the first frame of the data (ie, all indices besides the first two will simply be treated as zero).

row,col The coordinates of the point to be examined. Note that *row, col* may also be regarded as a *y, x* pair. Beware that *row* corresponds to the *y* index, not the *x* index.

value The actual data value to be put into the image. Note that if the datum represents some value that can not be represented in the data format of the image itself (such as trying to place the value 500 into a *ubyte* image), a meaningless value will end up being put into the image.

The following error codes (defined in the *#include* file $< ifserr.h >$) may be set by **ifsipp** (in the variable *ifserr*):

IFSE_BAD_HEADER: The pointer *img* does not point to an actual IFS image.

IFSE_BAD_DTYPE: The image is of some data type that **ifsipp** does not recognize. Usually this indicates that your header has been damaged, and the field $img \rightarrow ifsdt$ is mangled, or that the image data type is "struct". It could also occur if someone added a new data type to that understood by IFS, and forgot to modify **ifsipp** accordingly.

BUGS:

- **Ifsipp** does not verify that the image passed to it corresponds to a 2-dimensional image.
- The indices *row, col* are not checked to verify that they lie inside the image dimensions.
- **Ifsipp** does not check to make sure that the data pointer *img-¿ifsptr* is not NULL. Previous versions of IFS did not allow this data pointer to be NULL, so it was not previously necessary to check for this.

Any of the above bugs could cause an abrupt and unpleasant termination of your program, generally with the infamous "bus error: core dumped" message under UNIX systems. These problems will not occur however, unless the user's program contains some other sort of error.

1.4.23 ifsmkh

ifsmkh – Create a two-dimensional IFS image **Usage:**

```
#include $<ifs.h>$
imageptr = ifsmkh(nrows, ncols, dataformat);

    IFSHDR * imageptr;
    int nrows, ncols;
    char * dataformat;}
```

THIS FUNCTION IS OBSOLETE STARTING WITH RELEASE 3.0 OF IFS
THE FUNCTION 'ifscreate' SHOULD BE USED INSTEAD

Ifsmkh is used to create a two-dimensional image. Space to store the image and its header is automatically allocated, and the image is initialized to all zeros. Various fields in the image header are filled in with default values. The dimensions of the image that is created are given by the arguments nrows (number of rows) and ncols (number of columns). The string dataformat sets what the format for each pixel in the image will be. Valid data types are listed in one of the appendices to this manual.

Ifsmkh returns a pointer to the header structure for the newly created image. If an error occurs (usually meaning that an argument was invalid, or that it couldn't allocate enough memory to make a new image), then the value *NULL* is returned. In this event, the global variable *ifserr* will be set to indicate the nature of the error. Possible values for *ifserr* are IFSE_NO_MEM and ISFE_BAD_DTYPE.

BUGS/NOTES: The data format string **is** case-sensitive.

1.4.24 ifsopen

ifsopen – open a file for reading or writing.

```
File = ifsOpen(FileName, Mode, Prompt, NumRetries);

    FILE * File;
    char * FileName;
    char * Mode;
    char * Prompt;
    int NumRetries;
```

Ifsopen opens up a file *FileName* for reading or writing, and returns a pointer to the open file descriptor (*stream* in Unix terminology). If the file can not be opened or some other error occurs, then **ifsopen** will return NULL. The argument *Mode* is the same as the mode argument to the standard i/o library function *fopen*, i.e. "r" or "w" for read or write access. If *FileName* is NULL, then **ifsopen** just returns *stdin* or *stdout* as is appropriate for the specified *Mode*.

If *FileName* is a null string (*FileName = ""*), then **ifsopen** will read the name of the file to be opened from *stdin*. If *stdin* is attached to a terminal, then the string *Prompt* will be printed before getting the filename (unless *Prompt* is NULL). *FileName* is read using the function *ifsGetFN*, and expanded using *ifsPrsFN*, so it may contain the names of environment variables or constructs of the form "~login_id" to represent some user's home directory name. If the name read in is "-" (a single dash), then **ifsopen** will return *stdin* or *stdout*, according to the argument *Mode*.

If a filename is being read interactively (when *FileName = ""*, *stdin* is connected to a terminal, and *Prompt* is not NULL), then the user is allowed *NumRetries* mistakes before **ifsopen** will give up and return NULL. For instance, if **ifsopen** tries to open a non-existent file for reading, it will print a message to the user and ask for a new name. After several failures it will give up. This is to prevent such things as runaway shell scripts from sitting in a perpetual error loop.

1.4.25 ifspin

ifspin – read in an image from disk

```
img = ifspin(filename);

    IFSHDR * img;
    char * filename;
```

Ifspin is used to read an IFS image from the specified file *filename*. All necessary storage space for the image and its data is automatically allocated. The "tail" information for the file is not read in. If the user wants the tail information read in, he should use the newer function *ifsGetImg*. If *filename* points to a null string, then **ifspin** will prompt the user to specify some filename. Any filename (whether or not read interactively) will be translated using the function *ifsPrsFN*, which will substitute for environment variables and names of users' home directories specified in the C-shell shorthand form of "~user/filenam". If *filename* is NULL, then input will be read from *stdin*. Also, if a user is prompted for a filename, if he specifies a name of "-", the input will be read from *stdin*. The printing of a prompt string will be suppressed if *stdin* is not attached to a terminal.

Ifspin returns a pointer to the new image, or NULL if some sort of error occurs. In the latter case, the external variable *ifserr* will be set to indicate the nature of the error. Possibilities are:

IFSE_NO_OPEN – if the specified file can't be opened (usually meaning that it doesn't exist).

IFSE_IO_ERR – if some sort of I/O error occurred (usually meaning the file does not contain a valid IFS image). The standard system I/O library variable *errno* may contain additional information about the nature of the error. Note that IFSE_NO_OPEN is a subclass of the IFSE_IO_ERR error, so one can check for both automatically by using a construct of the form "if (ifserr & IFSE_IO_ERR) action_to_take();".

IFSE_NO_MEM – if it isn't possible to allocate storage to put the image into.

IFSE_BAD_NAME – if some error occurred when translating the file name.

BUGS/NOTES:

Ifspin is an obsolete function. Under version 4 of IFS, this just remaps its arguments and calls *ifsGetImg*.

1.4.26 ifspot

ifspot – write an image to disk

```
status = ifspot(img, filename);

    int status;
    IFSHDR * img;
    char * filename;
```

Ifspot is used to write an IFS image to the specified file *filename*. If *filename* points to a null string, then **ifspot** will prompt the user to specify some filename, and read a filename from *stdin*. If *filename* is NULL, then **ifspot** will write the image to *stdout*. Also, if **ifspot** reads a filename from *stdin*, and the filename is "-", then **ifspot** will write to *stdout*. If *stdin* is not connected to a terminal (e.g, input is being piped in from another program), then the printing of a prompt will be suppressed.

The filename is translated using *ifsPrsFN*, so it may contain environment variables (beginning with a leading "$") and the names of users' home directories specified in the C-shell shorthand form of "~user/filename".

Ifspot returns the value IFS_SUCCESS if it succeeded, or IFS_FAILURE if some sort of error occurred. In the latter case, the external variable *ifserr* will be set to indicate the nature of the error. Possibilities are:

IFSE_BAD_HEADER – if *img* doesn't point to a valid image.

IFSE_NOT_IMAGE – if there is no data associated with the header, i.e., the field *img→ifsptr* is set to NULL.

IFSE_NO_OPEN – if the specified file can't be opened (usually meaning that the name is invalid or that the user doesn't have write permission in the directory in which he is trying to put the image).

IFSE_IO_ERR – if some sort of I/O error occurred. The standard system I/O library variable *errno* may contain additional information about the nature of the error. Note that IFSE_NO_OPEN is a subclass of the IFSE_IO_ERR error, so one can check for both automatically by using a construct of the form "if (ifserr & IFSE_IO_ERR) action_to_take();".

IFSE_BAD_NAME – if some error occurred while translating the name.

BUGS/NOTES:

- The function of **ifspot** has been superceded by the newer function *ifsPutImg*. Starting with version 4 of IFS, **ifspot** is just a dummy routine which remaps its arguments and calls *ifsPutImg*.

- **Ifspot** does not write out any "tail" information associated with the image.

1.4.27 ifsPrsFN

ifsPrsFN – expand a filename

```
NewName = ifsPrsFN(Name,rc);

    char * NewName;
    char * Name;
    int  * rc;
```

IfsPrsFN scans a string *Name* looking for references to environment variables or abbreviations for a user's home directory of the form "~user" such as is provided by the Unix C-shell. It returns a pointer to the expanded name, or NULL if it failed. The space for the expanded name is allocated using *calloc*, so it may be *cfree*'ed when the user is through with it. A status code is returned through the pointer *rc*. This code will be 0 if it was successful, 1 if the expansion failed (such as by reference to an unset environment variable), or 2 if the routine had an internal error (such as a failure in a call to *calloc*).

Environment variables are specified by prefixing the name with a dollar sign "$". The name of the environment variable may contain any alphanumeric character, and is terminated by the first non-alphanumeric character found. The name may be enclosed in braces to isolate it from other characters, such as when the user desires the first character after the environment variable name to be an alphanumeric. Also, if the name is enclosed in braces, almost any printable character can be part of the variable name rather than just alphanumerics. Environment variable substitution is done on a strict left to right basis.

A reference to some user's home directory may be specified in the same manner as that allowed by the Unix C-shell. If the first character in a filename begins with a tilde '~' character, then the word immediately following the tilde (where 'word' is terminated by the first character which is not alphanumeric or underscore) is taken to be the name of some user's login id; the name of the user's home directory is substituted for the "~login_id" construct.

Examples

Assume the following environment variables and login id's:

$i	"ifs"
$file	"output"
$J	"~john"
~	"/usr/users/myhome"
~john	"/usr/users/alpha"

Then the following names expand as:

NAME	EXPANSION
myfile$i	myfileifs
myfile.$i	myfile.ifs
~/myfile	/usr/users/myhome/myfile
~john/$file.$i	/usr/users/alpha/output.ifs
$J/$file.$i	/usr/users/alpha/output.ifs
$ibase	no expansion unless environment variable "ibase" set
$ibase	ifsbase (braces isolate "i" from "base")

1.4.28 ifsPutImg

ifsPutImg – open a file and write an IFS image to it

```
rc = ifsPutImg(Image, FileName, Prompt, WriteTail);

    int rc;
    IFSHDR * Image;
    char * FileName;
    char * Prompt;
    int WriteTail;
```

IfsPutImg writes an IFS image to some file *FileName*. If *WriteTail* is false (zero), then any tail information associated with the image will not be written to the new file. **IfsPutImg** returns IFS_SUCCESS if all went well, or IFS_FAILURE if something went wrong, in which case the external integer variables *ifserr* and *column* can be examined to determine the nature of the error.

If *FileName* is NULL, then the image is written to *stdout*. If *FileName* is a null string (that is, *FileName = ""*), then a filename will be read in from *stdin* using the routine *ifsGetFN*. In this case, if *Prompt* isn't NULL and *stdin* corresponds to a terminal, then the string *Prompt* will be printed on the terminal (actually, to *stderr*) before reading a name. If *stdin* is not attached to a terminal, such as when input is being piped in from another program, then the printing of the prompt string is suppressed. If a filename is read in from *stdin*, and it is "-" (a single dash), then the image itself will be written to *stdout*. Filenames are expanded using *ifsPrsFN*, so they may contain such things as environment variable names and "~login_id" constructs.

IfsPutImg works by opening the specified file and then calling *ifsWrImg* to do the actual work of storing the image. It then closes the file when it's done (unless it wrote to *stdout*). Note that calling **ifsPutImg** with *Filename* = NULL is essentially the same as calling *ifsWrImg* directly.

The complimentary routine to **ifsPutImg** is *ifsGetImg*.

1.4.29 ifsRdHdr

ifsRdHdr – read an IFS image header from an open file

```
hdr = ifsRdHdr(file);

    IFSHDR * hdr;
    FILE * file;
```

IfsRdHdr reads an image header from a previously opened file. It does not read in any data or tail information for the file. It returns a pointer to the new image header, or NULL if it failed, in which case the external integer variables *ifserr* and *column* can be examined to determine the nature of the error. Space for the header is allocated dynamically, and may be freed (using *cfree*) when the user is through with it.

After the header is read, the file pointer is positioned so that the next character read from the file will the first byte of the data stored in the file. Hence, **ifsRdHdr** does scan past any padding at the end of the header.

There is no complimentary routine for writing headers to open files in this version of IFS. Writing a header to a file without writing any data would not make sense. Accordingly, there is a function *ifsWrImg*, but not an *ifsWrHdr*.

1.4.30 ifsRdImg

ifsRdImg – read an IFS image from an open file

```
img = ifsRdImg(File, ReadTail);

    IFSHDR * img;
    FILE * File;
    int ReadTail;
```

IfsRdImg reads an image from a previously opened file. If *ReadTail* is false (zero), then any tail information associated with the image will not be read in. It returns a pointer to the new image header, or NULL if it failed, in which case the external integer variables *ifserr* and *column* can be examined to determine the nature of the error. Space for the header, data, and tail is allocated dynamically, each may be freed (using *cfree* or *ifsfree*) when the user is through with it.

IfsRdImg will always read the entirety of an image file (including tail information and any padding after it), discarding the tail if it is not wanted, and the file read position will be set so that the next read request will start with the first byte after the end of the image. If *File* corresponds to a disk file, this just means the read pointer will point to the end-of-file (unless some garbage has been concatenated to to the of the image file). If *File* does not correspond to a disk file, such as when piping is being used and *File* is *stdin*, this means the file read pointer is positioned so that subsequent read requests (including *read*, *scanf*, *getchar*, another call to **ifsRdImg**, etc.) will properly read new data rather than reading padding characters left over from the end of the first image file.

The complementary routine to **ifsRdImg** is *ifsWrImg*.

1.4.31 ifssiz

ifssiz – Get size (lengths of all dimensions) of an IFS image

```
#include $<ifs.h>$
dlength = ifssiz(image);
    int * dlength;
    IFSHDR * image;
```

Ifssiz is used to determine the lengths of each dimension of an IFS image. It returns a pointer to an integer array, the various elements of which indicate the lengths of each dimension of the image, and also how many dimensions the array is defined as. The array will have *N+1* elements, where *N* is the number of dimensions of the image. The first element of the array (element number zero) gives the number of dimensions for the image. Subsequent elements of the array give the length of each dimension, where the dimensions are in order of ascending rank; i.e., element one gives the number of pixels per line (number of columns) for the image, element two gives the number of lines (rows), element three is the number of frames, and so forth.

The space for the array returned by **ifssiz** is automatically allocated using standard system calls (e.g., *calloc*), and as such may be released back to the system with the appropriate calls (*free, cfree*) when the user is through with the array.

If there is some error in **ifssiz** , then the external variable *ifserr* will be set to some error code as defined in the file $< ifserr.h >$ – most likely IFSE_BAD_HEADER or IFSE_NO_MEM.

Example usage:

```
int nrows, ncols, ndims, * dimlength;
IFSHDR * image2d;

... make or read in image pointed to by image2d ...

dimlength = ifssiz(image2d);
ndims = dimlength[0];
if (ndims != 2)  { /* Exit with nasty error messages ... */ }
ncols = dimlength[1];
nrows = dimlength[2];
cfree( (char *) dimlength );
```

1.4.32 ifsslice

ifsslice – take a complete slice of a two-d or three-d image

Usage:

new_img = ifsslice(old_img,string,value)

where string is a char pointer pointing to a string. The following are legitimate strings

"frame", "f", "row", "r", "column", "col", "c".

Passing any one of these strings will inform the function that the slice should be taken with that particular dimension (row,col or frame) held constant at the integer parameter value. i.e, if the string is "frame" and the value = 10, then a 2-d slice of the 3-d image at frame=10 is returned. Similarly for row and col slices. This is a generic slice program for 2-d and 3-d images. Using ifsslice on 1-d images will return with a copy of the image pointer and a warning.Similarly,a text string of "frame" on a 2-d image returns a copy of the image pointer and a warning.

Returns:

This function returns NULL if an error occurs, and returns an error code thru the external variable 'ifserr'.

External variables:

ifserr, IFSSLV

Special routines used:

ifscfree, ifsdie, ifswarn, ifsexwin, ifsexwin3d, ifssiz

Ifsslice extracts a complete slice of a two-d or three-d image from the constituent image. The datatype of the sliced new image is exactly that of the old image. Note that the slice is complete in all dimensions except in one dimension.

1.4.33 ifsversion

ifsversion – display version numbers

```
ifsversion(file);
    FILE * file;
```

Ifsversion will write the version numbers of all the IFS functions it knows about to the specified *file*. Typically, *file* will be *stdout* or *stderr*. For each function **ifsversion** knows about, it will print the name of the function, its version number, and the date it was last modified. In rare cases there may be some additional information printed.

1.4.34 ifsWrImg

ifsWrImg – write an IFS image to an open file

```
rc = ifsWrImg(Image, File, WriteTail);
    int rc;
    IFSHDR * Image;
    FILE * File;
    int WriteTail;
```

IfsWrImg writes an IFS image to some opened file *File*. If *WriteTail* is false (zero), then any tail information associated with the image will not be written to the new file. **IfsWrImg** returns IFS_SUCCESS if all went well, or IFS_FAILURE if something went wrong, in which case the external integer variables *ifserr* and *column* can be examined to determine the nature of the error.

The complimentary routine to **ifsWrImg** is *ifsRdImg*.

1.5 IFS Error Codes

This section describes the various error flags which may be set (in the global variable *ifserr*) when an error occurs in an IFS routine. These are defined in the *#include* file $< ifserr.h >$. Each error is represented by a bit or set of bits in *ifserr*; hence it is best to test for specific bits rather than using a standard comparison ("=="). Note that all of the IFS error codes have names which are of the form IFSE_xxxxxx, where xxxxxx is the actual name for the error.

IFSE_ERROR This is a combination of all possible errors. It is defined to be -1, i.e., all bits of the variable *ifserr* are set. Hence, all other error codes are subclasses of this code. IFS routines do not generally return this code. It generally indicates that either (a) an error was too complex for IFS to figure out, or (b) it was such a rare error that it was not considered important enough to define a separate code for the error.

IFSE_BAD_HEADER **or** IFSE_BAD_HDR The pointer you passed to a function does not correspond to a valid header for an IFS image. Most IFS routines double-check image headers before doing anything, and will exit with IFSE_BAD_HEADER set if the header is not valid.

IFSE_NULL_HEADER **or** IFSE_NULL_HDR The value NULL was passed to some IFS routine where you should have passed a pointer to an image header. The most likely cause of this is calling a routine to get or put a pixel in an image, when you haven't yet created (or read in) the image. This error is a subclass of the error IFSE_BAD_HEADER.

IFSE_NO_OPEN A file could not be opened for I/O activities. Usually this indicates that the file does not exist (on reads), or that the user does not have access permission for the specified file or directory. IFSE_NO_OPEN is a subclass of the error IFSE_IO_ERR.

IFSE_NOT_IMAGE An error occurred when attempting to read an image header from a file. This usually means the file is too small to possibly be an IFS image. An image header alone occupies at least 1 block, where a block is normally defined to be 512 bytes. IFSE_NOT_IMAGE is a subclass of the error IFSE_IO_ERR.

IFSE_BAD_NAME A filename is considered invalid. This is normally set within routines such as *ifsPrsFN* when a name expansion fails (such as by reference to a file "~fred/file.ifs" when user "fred" doesn't exit). IFSE_BAD_NAME is a subclass of the error IFSE_IO_ERR.

IFSE_IO_ERR Some sort of error occurred while performing I/O. The system global variable *errno* may contain additional information about the error. Common causes are (a) encountering an unexpected EOF, (b) inability to write output due to a full disk or user's disk quotas exceeded, (c) inability to open a file.

IFSE_BAD_DTYPE The datatype (*short, int, float,* etc.) is invalid or unrecognized by a particular routine. Usually this will only occur if you pass an invalid argument to an image creation routine (e.g., *ifsmkh* or *ifscreate*). It might also occur on routines which read or write data in images if the image header has been corrupted, or if the function is not capable of working on an image of a particular data type (for instance, it would make little sense to pass a *complex* format image to a histogram routine).

IFSE_BAD_POS Some coordinate (array index) is illegal for the specified operation, such as trying to access a pixel in column 30 of an image which is only 20 columns wide. Note that the routines which read or write single pixels currently do NOT check to see if coordinates are within bounds. This is a flaw with IFS which will probably be fixed at a later time.

IFSE_WRONG_NDIM The routine called does not work with images of the dimensionality of the image being used. An example would be trying to extract a window (2D subimage) from a 1-dimensional array.

IFSE_NOT_SUPPORTED The specified function is not currently allowed. Usually this indicates a function which is not yet implemented, but which is intended to be implemented. In rare cases it may indicate that a function is obsolete (a separate error code may later be defined for this).

1.6 IFS Data Types

This section describes the various data types that IFS version 5 understands. IFS has a certain basic set of names it recognizes for data types, and which it actually puts into image headers; in addition, it recognizes a number of synonyms for data types which it automatically remaps into the real data type. Some of these synonyms are machine dependent, for instance, a type of "int" may map to "16bit" on one machine and "32bit" on another machine. The final authority on data types and synonyms is the header file "ifstypes.h" which contains a table relating synonyms to the proper data type. Note that the data types ARE case sensitive. The possible data types are:

8bit Signed byte. Synonyms are byte, char, i1, and I1.

u8bit Unsigned byte. Synonyms are ubyte, uchar, u1, and U1.

16bit Signed 16 bit integer. Synonyms are short, i2, and I2.

u16bit Unsigned 16 bit integer. Synonyms are ushort, u2, and U2.

32bit Signed 32 bit integer. Synonyms are int, long, i4, and I4.

u32bit Unsigned 32 bit integer. Synonyms are uint, ulong, u4, and U4.

32flt 32 bit floating point number. Synonyms are float, real, real*4, r4, and R4.

64flt 64 bit floating point number. Synonyms are double, real*8, r8, and R8.

32cmp Complex number consisting of two 32flt numbers (real and imaginary parts). Synonyms are complex, complex*4, c4, and C4.

64cmp Complex number consisting of two 64flt numbers (real and imaginary parts). Synonyms are complex*8, c8, and C8.

struct Arbitrary user defined structure. Although IFS will read and write such images, it supplies no intrinsic routines to manipulate such images.

1.7 The structure of an IFS image

An IFS image, whether it is in a disk file or in program memory, is stored as a set of three distinct pieces. When written to disk, each piece will begin on a block boundary, where the size of a block is given by the constant BLOCKSIZE, which is defined the *#include* file $< ifs.h >$. Hence, there may be garbage bytes between one section and the next.

The first piece is a *header* for the image. This header contains all sorts of information relevant to the processing of the image, along with information intended solely for the user's benefit. Sample items in the header include the number of dimensions the image has, how long each dimension is, who the creator of the image is, and so on.

The second entity in an image is the actual image data. The data is just stored in one long linear array, in exactly the same way that any C program stores arrays. The user can directly access this data if he/she so desires, although the usual way to to get at the data is to use various IFS routines such as *ifsigp* and *ifsipp*.

The third part of the image is the *tail*. The tail is just a block of data at the end of the file which IFS places no particular interpretation upon. It is up to the users' programs to manipulate and understand the contents of the tail. An sample usage for the tail might be to store the text of a spoken message for which the data block was the digitized message.

In most cases, it is not necessary for the user to directly alter any of the information in the image header as the IFS routines themselves will fill in the header with all the information needed to process the image, and all of the user information fields will be set to default values which are fine for most applications. However, at times it is desirable to alter fields in the header, which requires that the user know what the fields in the header are, and how they are used.

The header actually consists of several C structures. These structures are defined in the *#include* file $< ifs.h >$. The header actually consists of two types of structures. The first structure is the main image header structure, and contains most of the relevant information about the image, such as the number of dimensions, the format of the data, etc. This structure is the so called IFSHDR structure which one refers to when one declares an image pointer variable in a program (e.g: IFSHDR * img1, * img2;).

Along with the IFSHDR structure exists a variable number of *dimension sub-headers*. There is one of these sub-headers for each dimension of the image, e.g., a 2-d image would have two sub-headers. The main piece of information in these subheaders is how long each dimension is. This structure goes by the name IFSDIM. The IFSDIM structures come directly after the the main header structure, both in the in-core images and the disk images. Hence, given a pointer to the main header structure, and the sizes of the headers, one can easily generate pointers to any of the dimension headers. The macro *ifsgetdim* (defined in $< ifs.h >$) may be used for this:

```
IFSDIM * dim;
IFSHDR * img;
dim = ifsgetdim(img,2);
```

will return a pointer to the third dimension sub-header (the first sub-header has number of zero).

1.7.1 The image header fields

char ifsmgc[4] This is the "magic number" field in the header. This field is used by the various routines as a way of verifying the validity of the header passed to them. If this field does not contain a special "magic number" (really, a character string rather than a number), then the IFS routines will assume that an invalid pointer was sent to them. The user should never alter this field.

int ifsbb This is the number of bytes in a physical block, when images are stored on disk. This value is set to the constant BLOCKSIZE, which is defined in "ifs.h". For all systems to date, the blocksize is 512. When images are written to disk files, the header always starts at block 0, and the data always begins at the start of the next block after the header, i.e, there may be a small amount of wasted space between the end of the header and the start of the data, if the header does not completely fill the last block it occupies.

int ifssoh This is the block number of the first block of the header. This is always set to zero, at least for the time being.

int ifssod This is the number of the block at which the data starts. The user can position directly to the start of the data array by using *fseek* to position an I/O pointer to the *ifssod*ifsbb* byte of the file. Of course, this only works for disk files.

int ifssot This is the block number of the start of the *tail* for the file. If this field is negative, it indicates that there is no tail present; taking the absolute value of it would give the block number at which the tail would be if it existed.

char * ifstail This is a pointer to the image tail, for an in-core image. If there is no tail, this is set to NULL.

int ifstsz This gives the size of the tail in bytes. If there is no tail, this is just zero.

char ifsfc[8] This is the *file class* field. This is not used by UNIX installationss of IFS, and is intended for systems running operating systems other than UNIX.

char ifsct[8] This is the file *class type* field. This also is not used and is for non-UNIX systems.

char ifsunm[32] This field is used to store the name of the owner of the file, as a null terminated character string. Note that since one byte must be reserved for the terminating null, that the effective username length is 31 characters. The user can put anything here he wants. When a user creates a new image, this field is filled in with his/her login id.

char ifscdt[32] This is a character string giving the time and date at which the image was created. This is automatically filled in when a user creates a new image, but the user can change it if he so desires. As with the name field, there can be up to 31 characters, plus the terminating null character.

char ifscpg[32] This is a character string giving the name of the program which created the image. When a user first creates an image, this field is normally filled with the name of the subroutine which actually created the image (e.g.: "ifsmkh").

char ifsver[8] This is a character string giving the version of the program which created the image. E.g., "V 1.00" or "Ver 1A" or something in that vein. Certain routines such as *ifsmkh* will stuff their version number in here.

char ifsrs1[40] This is just space reserved for future expansion.

char ifsdts[16] This is a character string giving the units of data for the pixels in the image, e.g., for an intensity image, this field might contain "lumens". One must make sure not to use names for units which exceed 15 characters. The default for this field is just "pixels".

float ifsdsc This field gives a scaling factor for the data in the image. This can be used along with the data offset (defined below) to convert values in the image array to some other scale. This might be used for example, if an image is taken and digitized using some measuring instrument, and later it is found that the instrument was "offcenter" (a data offset) or suffered from some sort of compression (scaling) problem. The default for this field is 1.0.

float ifsdof This field gives an offset which should be applied to the data in an image. I.e., the real value for a point in the image array should be calculated as

$$real - value = stored - value * ifsdsc + ifsdof$$

Note that the routines which get values from the image array (such as *ifsfgp*) do NOT apply the scaling factors. The default for this field is 0.0.

char ifsdt[16] This is a character string which tells what number format the pixels in the image are stored in, such as *u8bit* or *32flt*.

int dtype This is a numeric encoding of the *ifsdt* field which has been added to the header structure with version four of IFS. This has been added to increase the speed at which certain routines work.

int ifsbpd This is the number of bytes which are needed to store a single pixel value, i.e., it's the "sizeof" whatever data type is used for the image. Of course, this field can be deduced from the *ifsdt* field.

int ifsdims This gives the number of dimensions for the image. This refers to the number of indices needed to get at values in the image array, i.e, the pixels themselves don't count as a dimension. For instance, an image which has 10 rows and 20 columns is a 2-d image. Some other nomenclatures might refer to this as a 3-d image, where the third axis is the pixel measurement axis (range, brightness, or whatever).

char * ifsptr This is a pointer which gives the address of the of the first data element in the data array, for in-memory images. When files are written to disk, the value NULL is written for this field. This is normally automatically set to the correct value when an image is read in, although the user can alter it to point to some other array.

int * ifsdln This is a pointer to an array which is used when calculating the address of any arbitrary point of the image. This array has *ifsdims* elements. The first element is just set to 1, the next element is the number of columns in the image, the next element is the number of rows times the number of columns, the fourth element is numcols * numrows * numframes, and so on. If the user has an N dimensional image, and the N-length vector V gives the coordinate of some point in the image (i.e., V = [col, row, frame, cube...]), then the dot-product of V and *ifsdln* will give you an offset which may be added to the starting address of the image to find the desired element, assuming the "starting address" is an appropriately declared pointer. If the "starting address" is declared as a "char *" (such as with the header field *ifsptr*) then the offset must be scaled by the data size (*ifsbpd*). This may sound confusing, but really just represents the usual way that a set of indices are converted to absolute memory addresses for an array, whether by IFS or the C language itself. Note: the array itself is not written to disk when an image is stored. It is created when an image is read in (such as by *ifsRdImg*, *ifspin*, etc.) using information in the dimension sub-headers.

char * userptr This field is not used at present.

char ifsrs3[4] More reserved space.

1.7.2 The dimension sub-header fields

For each dimension of the image, there will be a structure of the following form tacked on after the end of the main header structure. The user can obtain a pointer

to one of these structures using the *ifsgetdim* macro, or can calculate their positions manually using the size of the main header and subheaders.

The dimension sub-header fields are:

int ifslen The length (number of elements) of this dimension.

int ifsrnk The *rank* of this dimension. The rank of the dimension defines the order in which the dimensions are actually stored in memory. The dimension with the lowest rank is the dimension which changes most rapidly. Hence, the dimension with rank 1 is equivalent to "columns", the rank 2 dimension is "rows", the rank 3 dimension is "frames", and so on. Note that images are stored in row-major order (as with all C arrays), which is contrary to the way some languages store arrays – Fortran for instance stores in column-major form. Also note that the first dimension subheader after the main header is not necessarily the header for the lowest rank (columns) although the IFS routines do by convention store the dimension subheaders in order of ascending rank, this is not a requirement.

char ifsdir[8] The *direction* of this dimension. This is for images for which lines are not always stored in a top to bottom, left to right form. For instance, some camera systems scan from left to right on one line, then go from right to left on the next line, and store the data in the same form. This would be known as "forward-backward alter" storage. Other possibilities include "forward" (normal), "backward", and "backward-forward alter". Currently, IFS does NOT recognize this field, and treats all images as being stored in "forward" format. This is only for possible future expansion. The string "fwd" is placed in this field.

char ifsxun[8] This is a character string which gives the units for this dimension e.g., "inches" or "mils". Make sure not to use names exceeding 7 characters. The default for this field is "pixels".

float ifsxsc The scaling factor to apply to this dimension, analogous to the scaling factor which exists in the main header.

float ifsxof The scaling offset for this dimension.

char rs4[32] Reserved space.

Chapter 2

Image Processing Subroutines

In this chapter, A number of subroutines are presented which are of general applicability. Most have been written using pointers and sophisticated code in order to optimize speed.

2.1 Subroutine descriptions

The following subroutines are available in the library /usr/local/lib/libiptools.a

2.1.1 ifsadd

ifsadd– add two ifs images, point by point

$$out(i,j) = in1(i,j) + in2(i,j)$$

```
int  ifsadd (in1,in2,out)
IFSHDR *in1,*in2,*out;
```

RETURNS 0 if successful,
-1 if all three arguments do not have same dimensions
-2 if data type unsupported (complex double)
-3 if one input has type complex and output is real
-4 if both inputs are real and output is complex

CAUTION if output is type char, values greater than 255 will be clipped to lie between 0 and 255

NOTES if one image is real and the other complex, the output must be complex and the real parts of the images will be added

2.1.2 ifscfft2d

ifscfft2d– perform in-place 2D fast Fourier transform

```
len = cfft2d(img_ptr,type)
IFSHDR * imgptr;
int type;
```

Ifscfft2dperforms an in-place 2-D fast Fourier transform on a complex ifs image. The transform is performed in place on 8BYTE-PER-PIXEL (complex float) data only! Note that fft's only work on images of dimension $2^n \times 2^n$.

The second argument is an indicator for forward or inverse fft. -1 for forward, +1 for inverse

If there is some error, the subroutine exits to the user with an error message. Possible errors are:

- Image dimensions are not a power of two
- Image data type is not complex float

2.1.3 ifsc2imag

ifsc2imag– extract imaginary part of a complex ifs image, point by point

```
val = ifsc2imag (in1,out)
   int val;
   IMSHDR *in1,*out;
```

RETURNS 0 if successful,
-1 if both arguments do not have
same dimensions -2 if data type unsupported (complex double)

CAUTION: if output is type char, values greater than 255 will be clipped to lie between 0 and 255

2.1.4 ifsc2mag

ifsc2mag– return magnitude of a complex ifs image, point by point

```
val = ifsc2mag (in1,out)
   int val;
   IMSHDR *in1,*out;
```

RETURNS 0 if successful,
-1 if both arguments do not have
same dimensions -2 if data type unsupported (complex double)

CAUTION: if output is type char, values greater than 255 will be truncated to 255

2.1.5 ifsc2phase

ifsc2phase– return phase of a complex ifs image, point by point

```
val = ifsc2phase (in1,out)
   int val;
   IMSHDR *in1,*out;
```

RETURNS 0 if successful,
-1 if both arguments do not have
same dimensions -2 if data type unsupported (complex double)

CAUTION: if output is type char, values greater than 255 will be truncated to 255

2.1.6 ifsc2real

ifsc2real– return real part of a complex ifs image, point by point

```
val = ifsc2real (in1,out)
   int val;
   IMSHDR *in1,*out;
```

RETURNS 0 if successful,
-1 if both arguments do not have
same dimensions -2 if data type unsupported (complex double)

CAUTION: if output is type char, values greater than 255 will be truncated to 255

2.1.7 ifsmult

ifsmult– multiply two ifs images, point by point

```
int  ifsmult (in1,in2,out)
IFSHDR *in1,*in2,*out;
```

RETURNS 0 if successful,
-1 if all three arguments do not have same dimensions
-2 if data type unsupported (complex double)
-3 if one input has type complex and output is real
-4 if both inputs are real and output is complex

CAUTION if output is type char, values greater than 255 will be truncated to 255

NOTES if one image is real and the other complex, the output must be complex and the real parts of the images will be added

2.1.8 ifsrecip

ifsrecip– take reciprocal of an ifs image, point by point

```
int  ifsrecip (in1,out)
IFSHDR *in1,*out;
```

RETURNS 0 if successful,
-1 if both arguments do not have same dimensions
-2 if data type unsupported (complex double)
-3 if one input has type complex and output is real
-4 if both inputs are real and output is complex

CAUTION if output is type char, values greater than 255 will be truncated to 255

NOTES if one image is real and the other complex, the output must be complex and the real parts of the images will be added

2.1.9 ifssub

ifssub– subtracts two ifs images, point by point. The second argument is subtracted from first.

```
int  ifssub (in1,in2,out)
IFSHDR *in1,*in2,*out;
```

RETURNS 0 if successful,
-1 if all three arguments do not have same dimensions
-2 if data type unsupported (complex double)
-3 if one input has type complex and output is real
-4 if both inputs are real and output is complex

CAUTION if output is type char, values greater than 255 will be truncated to 255

NOTES if one image is real and the other complex, the output must be complex and the real parts of the images will be added

Chapter 3

Image Synthesis Programs

3.1 qsyn-synthesize range images

Qsyn generates synthetic altitude images of objects which are composed of quadric surfaces or pieces of quadric surfaces.

Usage:

% qsyn formatfile.q

Qsyn is a program which generates synthetic altitude images of objects which are composed of quadric surfaces or pieces of quadric surfaces. The image data is in an unsigned byte format, although a few minor changes could be made to **Qsyn** to allow for some other output data type. Image manipulation is done using the **IFS** image manipulation routines in use at Communication Unlimited.

Qsyn generates *altitude images*, *i.e.*, two dimensional images which contain three dimensional information, where the coordinates of a pixel (its row and column index) correspond directly to the x and y values for the pixel, and the pixel value itself (the datum) corresponds directly to the *altitude* or z value for the point. Hence, a point in three-space at position *[x,y,z]* corresponds to some pixel in an image I:

$$[x, y, z] \rightarrow I[r, c] \tag{3.1}$$

where *I[r,c]* is the value of the pixel at row r, column c of the image. *R*, *c*, and *I[r,c]* are linearly related to *y*, *x*, and z, respectively. In **Qsyn**, the linear relationship is simply taken to be $r = y$, $c = x$, and $I[r, c] = z$. Note that an *altitude* image is not the same as a *range* image, which is also commonly used to represent three-dimensional images. In a range image, the pixel value represents the distance from some point in three space to a fixed reference position (*i.e.*, the viewpoint), whereas an altitude image is based on the distance to some reference plane, hence a range image is actually a perspective projection of an altitude image.

Qsyn generates images composed of quadric surfaces. An image may contain any number of surfaces; in addition, each surface may also have constraints placed on it. These constraints are also quadric surfaces (quadric inequalities). A quadric surface is a 3-dimensional surface which may be described by a general quadric equation:

$$q(x,y,z) = Ax^2 + By^2 + Cz^2 + Fyz + \\ Gxz + Hxy + Px + Qy + Rz + K = 0. \tag{3.2}$$

This includes common shapes such as spheres, cones, and planes. **Qsyn** works by generating a quadric surface which is bounded by a set of quadric constraints. For example, an image of an open-ended can may be produced by synthesizing an image of a cylinder which is constrained by two planes perpendicular to the cylinder. The constraints would specify those points on the cylindrical surface which lie above the lower plane and below the upper plane. As a matter of terminology, I will use the term *quadric section* to refer to a quadric surface along with a set of constraints on that surface.

In order to use **Qsyn**, you must understand the coordinate systems it uses to orient surfaces and sections. Theoretically, you could place surfaces wherever you wanted by specifying the appropriate coefficients in the quadric equation 3.2. In practice, this is a pain since the coefficients are a function of the objects position and orientation as well as its shape. E.g.,

$$x^2 + y^2 + z^2 = r^2 \tag{3.3}$$

describes a sphere of radius r centered at the origin. The quadric coefficients are $A = B = C = 1, K = -(r^2)$. Moving this sphere so that its center is at location $(x,y,z) = (10,5,0)$ gives:

$$(x-10)^2 + (y-5)^2 + z^2 = r^2 \tag{3.4}$$

which when put into the form of equation 3.2 looks like

$$x^2 + y^2 + z^2 - 20x - 10y + (125 - r^2) = 0. \tag{3.5}$$

Rotating an object affects the quadric coefficients in a still more complicated way.

Qsyn allows you to specify a surface using any coordinate system you desire; you may then translate or rotate the object to move it to a different coordinate system. **Qsyn** uses several different coordinate systems to ease the task of creating images which are composed of multiple surfaces. Figure 3.1 shows the various coordinate systems used, and are described here in the text.

Each surface (including constraints) is defined in terms of its own *local* coordinate system. Typically you would choose the coordinate system in which it was

Figure 3.1: Coordinate systems used by QSYN

easiest to describe the shape you want. Each quadric section has its own coordinate system, known as a *section* or *object* coordinate system. The latter term is perhaps misleading in that what you think of as an object may actually consist of more than one section.

By specifying the relationship between the *local* coordinate system for each surface in a section to the *section's* coordinate system, you specify how the various parts of the section fit together. This essentially is used to relate the constraints to the actual surface being synthesized, with the coordinate system of the surface itself (its *local* system) typically being coincident with the *section* coordinate system. This is not a requirement though; the surface and its constraints are all placed relative to a common *section* system, rather than the constraints being placed relative to the *local* system of the surface.

The next higher level coordinate system is the *reference* or *base* coordinate system. This is the base coordinate system for the entire image to be synthesized. Its relationship to the *object* coordinate systems is the same as that of the *object* to *local* coordinate systems. By specifying the relative locations of each *object* coordinate system in the *base* system, you define how the various quadric sections fit together, and define what the overall image will look like.

The highest level coordinate system is the *viewpoint* coordinate system. This corresponds to the coordinate system for the image array, and hence, the display equipment. The x and y axes correspond to the horizontal and vertical axes of the display, and the z axis would be the actual pixel value, *i.e.*, the brightness or color of the pixel indicates the altitude. The relation between the view coordinate system and the base coordinate system specifies the position which you are actually seeing the image from. In many cases the two coordinate systems may be coincident, or one merely a translation of the other. Note that the term *viewpoint system* is somewhat misleading in that this is not a range image; this transform really defines a plane of projection for the image (which is fully described at the level of the *base* system). The projection plane itself is the xy plane of the *viewpoint* coordinate system. The image array itself is just a finite piece of this infinite projection plane. Specifically, the origin $[x, y] = [0, 0]$ corresponds with the pixel at location $[col, row] = [0, 0]$ in the image array (which is in one corner of the image). Hence, if you define the objects in your image to lie around the origin (in the *base* coordinate system), when you display the image you will probably find that all of the objects will lie in one corner of the image, unless you have displaced the *base* system relative to the *viewpoint* system. Put simply, the origin of the *base* system will be in one corner of the image you synthesize unless you make sure to move it – and you may end up not seeing parts of your objects since they will be clipped at the image borders.

In **Qsyn**, the relations between coordinate systems is expressed in terms of six basic motions: translations along the three coordinate axes, and rotations about the axes. Motions along the axes go by the names of *movex*, *movey*, and *movez*. Rotations about the axes go by the names of *rotx, roty,* and *rotz*, or alternatively

as *yaw*, *pitch*, and *roll*, respectively. When specifying the relationship between two coordinate systems, the motions are given in terms of the higher coordinate system, *i.e.*, the higher level system is the base system. For example, to specify that the *local* coordinate system for a constraint has its origin at location $(x, y, z) = (10, 5, 2)$ in the *section* coordinate system, you would specify the motion as *(movex 10, movey 5, movez 2)*. Note that you do not specify the motion as *(movex -10, movey -5, movez -2)*, as this would be specifying the origin of the higher system (the *section* system) in terms of the lower (the *local* system). This is easy to see for motions which are pure translations, but may provide a source of confusion when rotations are involved.

The best way to regard the motions is as being *object oriented*. Although you are specifying the relationship between two coordinate systems, the lower coordinate system can be regarded as an object in the higher coordinate system (imagine that the lower level system has a cube sitting at its origin, and you are moving the cube around in the higher level system). Initially, the two coordinate systems start out with coincident axes, *i.e.*, the lower system is an object sitting at location (0,0,0) in the higher system. If you specify a motion of *movex 10*, then all 'objects' in the higher level system are displaced 10 units down the x axis. If you specify a motion of *roll 20*, then all objects swivel around the z axis of the higher system. This is not the same as simply rotating the lower coordinate system's coordinate system by 20 degrees! If the lower system's origin coincides with the higher system's origin, then the effects *are* the same; however, if the origin points do not coincide, then the lower system's origin will be seen to swing on an arc around the higher system's axis. Hence, the rotation will also cause a translation in two of the axes of the higher system. Figures 3.2 illustrate this, and also show how the order of motions is important.

Qsyn works by reading a file which describes the image to be synthesized. This file contains the quadric coefficients for each surface and constraint (in their own local coordinate system), and movement commands which specify how the coordinate systems are located. **Qsyn** reads this specification file and generates the image; when it is done, it prompts for the name of a file to write the image to. The output file is a 2-d IFS format image. The data type for the image will be unsigned byte, with the value of a pixel indicating its height. The nearer a pixel is, the higher its pixel value.

The format for the image specification file is relatively simple. presented below is a sample specification file; the format is described below. It is a text file composed of several blocks. These blocks may contain other blocks within them.

At the beginning of the file is a header block which specifies the size (number of rows and columns) of the image to be generated, the number of quadric sections to synthesize, and a set of motions which will translate the reference coordinate system to the viewpoint coordinate system.

The motions which specify the relationship between the two coordinate systems

Figure 3.2: Order of motions

```
##############################################################################
#                                                                            #
# This is a QSYN description file which will generate a simple altitude image #
# of a clipped lead sticking through a hole in the underside of a printed    #
# circuit board.  The lead will have a "clinch angle" (angle that lead is    #
# bent away from pointing straight up) of 80 degrees, and a "lead angle"     #
# (angle in the xy plane -- the plane representing the PC board) of 30       #
# degrees.                                                                   #
#                                                                            #
# The image is composed of 4 pieces (quadric sections):                      #
#                                                                            #
#     1.  A cylindrical piece which is the body of the lead.                 #
#     2.  A plane with a hole in it which represents the circuit board.      #
#         The cylinder in (1) goes through the hole.                         #
#     3.  A sphere at one end of the cylinder in (1) [same radius as the     #
#         cylinder] which terminates the lower end of the lead.              #
#     4.  A second spherical section which caps the other end of the lead.   #
#         This second sphere has a larger radius than the sphere, so that    #
#         the higher end of the cylinder is clipped almost in a plane.       #
#         Note that if this cap wasn't here, you'd be able to see inside the #
#         cylinder in (1)!                                                   #
#                                                                            #
# NOTE:  This file is not directly suitable to be passed as input to QSYN.   #
#        QSYN does not understand comments in a file (comments going from    #
#        the '#' symbol to the end of the line).  However, a little          #
#        cleverness under Unix systems will make it suitable:               #
#                                                                            #
#                 sed 's/#.*$//' example.q | qsyn                            #
#                                                                            #
#        The strange looking "sed" command will edit out the comments before #
#        piping the file into qsyn.                                          #
#        Eventually, I will fix QSYN so that it will remove the comments     #
#        itself.                                                             #
#                                                                            #
##############################################################################
```

Figure 3.3: QSYN example, page 1

```
#------------------------ Header section -----------------------------------

128 128 # Dimensions of image
4 # Number of quadric sections in image

# This is the 'base' to 'viewpoint' coordinate system transform block:
    roll 030 # Rotate 30 degrees about the Z axis
# This rotates EVERYTHING by 30 degrees, and is
# what gives the lead a "lead angle" of 30 degrees.
# Note that I'm really rotating the board too, but
# since the board is an infinite xy-plane, you can't
# tell it.  Hence, a more technically accurate, but
# slightly more cumbersome way to do this is to have
# the "roll 30" in the 'section' to 'base' transforms
# of each of the actual pieces making up the lead.
    movex 64 # Now I'll do a translation in X and Y so that
    movey 64 # the object is centered in the image rather
    movez 75 # than in the lower left corner.
    end # movez 75 is just 'cause I want board at Z = 75.
#
#---------------------- End of header section ---------------------------------


# This is the first quadric section.  It describes the plane which represents
# the printed circuit board.

# SECTION to BASE coordinate system transform block.
# All of the surfaces in this sections (that is, including the constraint
# surfaces) will be moved by this transform.
    end # A Null block (i.e., the two systems here
                               # are coincident).

###### Define quadric surface:
# The quadric surface coefficients:
    0 0 0
    0 0 0
    0 0 1   0  # The plane "Z = 0"
#
# and the transform relating the LOCAL system to the SECTION system:
    end # Null block.  systems are coincident.
###### end of definition for the quadric surface.
```

Figure 3.4: QSYN example, page 2

```
#
#  Now: define the constraints on the above surface:
#
1  # this is the number of constraints.
#
1 1 1  # constraint quadric coefficients
        0 0 0  # X*X + Y*Y + Z*Z = 100, ie, a sphere of
        0 0 0 -100  # radius 10.
#
# LOCAL to SECTION transform for this constraint:
end  # Once again, no transform.
# And lastly, a "<" or ">" symbol which indicates on which side of the
# constraint surface the object must lie:
    > # The ">" specifies that my constraint
# equation actually is
# X*X + Y*Y + Z*Z >= 100
# Hence, only points on the plane Z = 0 which
# are OUTSIDE this sphere are valid.
# This puts a hole at the center of my plane.



# This is the second quadric section.  It's the cylinder which makes up the
# body of the lead.  I originally define the cylinder as lying on the Z axis
# (which also makes it easy to specify constraints on it).  Then I use the
# SECTION to BASE transform to tip the cylinder (and its constraints) over
# to give the appearance of a clinched lead.

# The SECTION to BASE transform:  lead is clinched 80 degrees.  The "movez 6"
# displaces the cylinder upwards 6 units, which is needed because otherwise
# the lead will be embedded in the board plane, rather than lying just above it
    pitch 080 movez 6 end

# Define my cylinder:  cylinder on Z axis with radius 6:
    1 1 0
    0 0 0
    0 0 0
    -36
end # No transform.
```

Figure 3.5: QSYN example, page 3

```
#
# Now, specify my constraints.  Remember that the cylinder is defined as
# lying on the Z axis (even in the SECTION system, since the above transform
# was null), so the constraints clip the cylinder at right angles to its
# axis.  Now, the SECTION to BASE transform applies to both the cylinder
# and its constraints, so the net effect is to pitch the CLIPPED cylinder
# by 80 degrees.
#
    2 # there will be two constraints.

0 0 0   0 0 0   0 0 1  0   end > # Constraint 1: Z >= 0
0 0 0   0 0 0   0 0 1  -34 end < # Constraint 2: Z <= 34
#
# So, now I have a cylindrical piece lying between Z = 0 and Z = 34.
# Note that the cylinder is NOT capped at the ends.


# Section 3.  This is a sphere which will cap off the lower end of the lead.

movez 6 end # The "movez 6" is same as in section 2.
    1 1 1
    0 0 0 # Sphere of radius 6 (at the origin in the local
    0 0 0 # system) ...
    -36
end # ... and at the origin in the SECTION system ...
# but centered at x,y,z = 0,0,6 in the BASE system.

0 # Zero constraints.
```

Figure 3.6: QSYN example, page 4

```
# Lastly, section 4, the sphere capping off the upper end of the lead.
# You need to pay careful attention to the movements for this piece to
# observe how the spherical patch does indeed end up capping off the
# cylinder described in section 2.  Note that the movements given here
# certainly do not describe the ONLY valid way to get the patch; any of
# a variety of movements would do the trick.

pitch 080 movez 6 end
    1 1 1 # sphere of radius 8.
    0 0 0
    0 0 0
    -64
movez 28.7085 end # That goofy number you have to work out from
# the geometry of the situation.  The sphere
# is placed so that it will sit right at the
# end of the clipped cylinder.  The sphere
# will intersect the cylinder at a height of
# Z = 34 (or Z = 40 after the "movez 6" is
# applied).

1 # 1 constraint:
    0 0 0   0 0 0   0 0 1 -34 end > # only want that part of the
# sphere which will cap off
# the cylinder; I don't want
# to put a big ball at the
# end of the cylinder.



###### And now, the name of the file to write the image to:
pcb_lead.ifs
```

Figure 3.7: Sample QSYN input file, page 5

are actually an instance of a type of sub-block known as a *coordinate transform block*, or just *transform*. These transforms occur in several places, and always have the same format. There are six basic motions which may be specified in a transform block. There are also several complex motions which are simply composites of the basic motions. All of the motions are specified by some keyword describing the motion to perform, followed by the parameters appropriate for that keyword. The six basic motions are those mentioned earlier: *movex, movey, movez, rotz (roll), roty (pitch),* and *rotx (yaw)*. The *movex, movey,* and *movez* commands can also be shortened to *x, y,* and *z*. Each motion takes a single argument which is the amount to move or the angle to rotate (in degrees). To specify a complete transform block, you merely specify an arbitrary number of basic motions (in arbitrary order), followed by the keyword *end*. The basic motions are performed in the order specified. A sample transform block might look like:

movex 20 movey 10 pitch 30 end

This will shift an object 20 units along the x axis, 10 units in y, and swing the object 30 degrees around the y axis.

The composite motions are just shortcuts for specifying certain common sets of motions. The combination *'movex 10 movey 20 movez 30'* can be specified more rapidly as *'movexyz 10 20 30'* or just *'xyz 10 20 30'*. Similarly, *'rpy 10 20 30'* is short for *'roll 10 pitch 20 yaw 30'*. All six basic motions can be expressed using the commands '*rpyt* $\theta_z\theta_y\theta_x\Delta_x\Delta_y\Delta_z$' and '*trpy* $\theta_z\theta_y\theta_x\Delta_x\Delta_y\Delta_z$'. *Rpyt* does the rotations first, then the translations; *trpy* performs the translations first. Note that the syntax for *trpy* is inconsistent in that although the translations are performed first, they are the last arguments specified for the command.

After the header block there comes a set of *section* blocks, one block for each quadric section to synthesize. Each block describes a surface to generate, and all the constraints for the surface. The quadric section blocks are in turn composed of smaller blocks. The first sub-block is a transform block which converts from the section coordinate system to the base coordinate system. This is followed by a *surface* block. A surface block contains a set of quadric coefficients to describe a single quadric surface, and also contains a transform block which translates the surface's local coordinate system to the section coordinate system. After the surface block comes the number of constraints, followed by a set of constraint blocks. A constraint block is identical to a surface block except that it contains a flag indicating which way the constraint inequality goes (e.g., choose all points in the surface *below* some plane).

3.2 3dsyn-synthesize density images

3dsyn generates synthetic three dimensional density images of objects which are

composed of quadric surfaces or pieces of quadric surfaces.

Usage:

% 3dsyn formatfile.q

3dsyn is a program which generates synthetic three dimensional density images of objects which are composed of quadric surfaces or pieces of quadric surfaces. The image data is in an unsigned byte format, although a few minor changes could be made to **3dsyn** to allow for some other output data type. Image manipulation is done using the **IFS** image manipulation routines in use at Communication Unlimited.

3dsyn generates *density images*, *i.e.*, three dimensional images which contain three dimensional information, where the coordinates of a voxel (its row and column and frame index) correspond directly to the x, y, and z values for the voxel, and the voxel value itself (the datum) corresponds directly to the *density* for the point. Hence, a point in three-space at position *[x,y,z]* corresponds to some voxel in an image I:

$$[x, y, z] -> \ I[f, r, c] \tag{3.6}$$

where *I[f,r,c]* is the value of the voxel at frame f, row r, column c of the image. F, r, c, and *I[f,r,c]* are linearly related to z, y, and x, respectively. In **3dsyn**, the linear relationship is simply taken to be $f = z$, $r = y$, and $c = x$, and $I[f, r, c] = density$.

3dsyn generates images composed of quadric surfaces. An image may contain any number of surfaces; in addition, each surface may also have constraints placed on it. These constraints are also quadric surfaces (quadric inequalities). A quadric surface is a 3-dimensional surface which may be described by a general quadric equation:

$$\begin{aligned} q(x, y, z) = {} & Ax^2 + By^2 + Cz^2 + Fyz + \\ & Gxz + Hxy + Px + Qy + Rz + K = 0. \end{aligned} \tag{3.7}$$

This includes common shapes such as spheres, cones, and planes. **3dsyn** works by generating a quadric surface which is bounded by a set of quadric constraints. For example, an image of an open-ended can may be produced by synthesizing an image of a cylinder which is constrained by two planes perpendicular to the cylinder. The constraints would specify those points on the cylindrical surface which lie above the lower plane and below the upper plane. As a matter of terminology, I will use the term *quadric section* to refer to a quadric surface along with a set of constraints on that surface.

In order to use **3dsyn**, you must understand the coordinate systems it uses to orient surfaces and sections. Theoretically, you could place surfaces wherever you wanted by specifying the appropriate coefficients in the quadric equation 3.7. In practice, this is a pain since the coefficients are a function of the objects position and orientation as well as its shape. E.g.,

$$x^2 + y^2 + z^2 = r^2 \tag{3.8}$$

describes a sphere of radius r centered at the origin. The quadric coefficients are $A = B = C = 1, K = -(r^2)$. Moving this sphere so that its center is at location $(x, y, z) = (10, 5, 0)$ gives:

$$(x - 10)^2 + (y - 5)^2 + z^2 = r^2 \tag{3.9}$$

which when put into the form of equation 3.7 looks like

$$x^2 + y^2 + z^2 - 20x - 10y + (125 - r^2) = 0. \tag{3.10}$$

Rotating an object affects the quadric coefficients in a still more complicated way.

3dsyn allows you to specify a surface using any coordinate system you desire; you may then translate or rotate the object to move it to a different coordinate system. **3dsyn** uses several different coordinate systems to ease the task of creating images which are composed of multiple surfaces. Figure 3.1 show the various coordinate systems used, and are described here in the text.

Each surface (including constraints) is defined in terms of its own *local* coordinate system. Typically you would choose the coordinate system in which it was easiest to describe the shape you want. Each quadric section has its own coordinate system, known as a *section* or *object* coordinate system. The latter term is perhaps misleading in that what you think of as an object may actually consist of more than one section.

By specifying the relationship between the *local* coordinate system for each surface in a section to the *section's* coordinate system, you specify how the various parts of the section fit together. This essentially is used to relate the constraints to the actual surface being synthesized, with the coordinate system of the surface itself (its *local* system) typically being coincident with the *section* coordinate system. This is not a requirement though; the surface and its constraints are all placed relative to a common *section* system, rather than the constraints being placed relative to the *local* system of the surface.

The next higher level coordinate system is the *reference* or *base* coordinate system. This is the base coordinate system for the entire image to be synthesized. Its relationship to the *object* coordinate systems is the same as that of the *object* to *local* coordinate systems. By specifying the relative locations of each *object* coordinate system in the *base* system, you define how the various quadric sections fit together, and define what the overall image will look like.

The highest level coordinate system is the *viewpoint* coordinate system. This corresponds to the coordinate system for the image array, and hence, the display equipment. The relation between the view coordinate system and the base coordinate system specifies the position which you are actually seeing the image from. In many cases the two coordinate systems may be coincident, or one merely a translation of the other. Note that the term *viewpoint system* is somewhat misleading in that this is not a range image; this transform really defines a plane of projection for

the image (which is fully described at the level of the *base* system). The projection plane itself is the xy plane of the *viewpoint* coordinate system. The image array itself is just a finite piece of this infinite projection plane. Specifically, the origin $[x, y, z] = [0, 0, 0]$ corresponds with the voxel at location $[frame, col, row] = [0, 0, 0]$ in the image array (which is in one corner of the image). Hence, if you define the objects in your image to lie around the origin (in the *base* coordinate system), when you display the image you will probably find that all of the objects will lie in one corner of the image, unless you have displaced the *base* system relative to the *viewpoint* system. Put simply, the origin of the *base* system will be in one corner of the image you synthesize unless you make sure to move it – and you may end up not seeing parts of your objects since they will be clipped at the image borders.

In **3dsyn**, the relations between coordinate systems is expressed in terms of six basic motions: translations along the three coordinate axes, and rotations about the axes. Motions along the axes go by the names of *movex*, *movey*, and *movez*. Rotations about the axes go by the names of *rotx, roty,* and *rotz*, or alternatively as *yaw*, *pitch*, and *roll*, respectively. When specifying the relationship between two coordinate systems, the motions are given in terms of the higher coordinate system, *i.e.*, the higher level system is the base system. For example, to specify that the *local* coordinate system for a constraint has its origin at location $(x, y, z) = (10, 5, 2)$ in the *section* coordinate system, you would specify the motion as *(movex 10, movey 5, movez 2)*. Note that you do not specify the motion as *(movex -10, movey -5, movez -2)*, as this would be specifying the origin of the higher system (the *section* system) in terms of the lower (the *local* system). This is easy to see for motions which are pure translations, but may provide a source of confusion when rotations are involved.

The best way to regard the motions is as being *object oriented*. Although you are specifying the relationship between two coordinate systems, the lower coordinate system can be regarded as an object in the higher coordinate system (imagine that the lower level system has a cube sitting at its origin, and you are moving the cube around in the higher level system). Initially, the two coordinate systems start out with coincident axes, *i.e.*, the lower system is an object sitting at location (0,0,0) in the higher system. If you specify a motion of *movex 10*, then all 'objects' in the higher level system are displaced 10 units down the x axis. If you specify a motion of *roll 20*, then all objects swivel around the z axis of the higher system. This is not the same as simply rotating the lower coordinate system's coordinate system by 20 degrees! If the lower system's origin coincides with the higher system's origin, then the effects *are* the same; however, if the origin points do not coincide, then the lower system's origin will be seen to swing on an arc around the higher system's axis. Hence, the rotation will also cause a translation in two of the axes of the higher system. Figures 3.2 illustrate this, and also show how the order of motions is important.

3dsyn works by reading a file which describes the image to be synthesized. This

file contains the quadric coefficients for each surface and constraint (in their own local coordinate system), the density of the material contained within that surface, and movement commands which specify how the coordinate systems are located. **3dsyn** reads this specification file and generates the image; when it is done, it prompts for the name of a file to write the image to. The output file is a 3-d IFS format image. The data type for the image will be unsigned byte, with the value of a voxel indicating its density.

The format for the image specification file is relatively simple. Figure 3.8 shows a sample specification file; the format (described below), is identical to the description for QSYN except for the inclusion of density value on each line describing a quadric.

It is a text file composed of several blocks. These blocks may contain other blocks within them. At the beginning of the file is a header block which specifies the size (number of frames, rows and columns) of the image to be generated, the number of quadric sections to synthesize, and a set of motions which will translate the reference coordinate system to the viewpoint coordinate system.

The motions which specify the relationship between the two coordinate systems are actually an instance of a type of sub-block known as a *coordinate transform block*, or just *transform*. These transforms occur in several places, and always have the same format. There are six basic motions which may be specified in a transform block. There are also several complex motions which are simply composites of the basic motions. All of the motions are specified by some keyword describing the motion to perform, followed by the parameters appropriate for that keyword. The six basic motions are those mentioned earlier: *movex, movey, movez, rotz (roll), roty (pitch),* and *rotx (yaw)*. The *movex, movey,* and *movez* commands can also be shortened to *x, y,* and *z*. Each motion takes a single argument which is the amount to move or the angle to rotate (in degrees). To specify a complete transform block, you merely specify an arbitrary number of basic motions (in arbitrary order), followed by the keyword *end*. The basic motions are performed in the order specified. A sample transform block might look like:

movex 20 movey 10 pitch 30 end

This will shift an object 20 units along the x axis, 10 units in y, and swing the object 30 degrees around the y axis.

The composite motions are just shortcuts for specifying certain common sets of motions. The combination *'movex 10 movey 20 movez 30'* can be specified more rapidly as *'movexyz 10 20 30'* or just *'xyz 10 20 30'*. Similarly, *'rpy 10 20 30'* is short for *'roll 10 pitch 20 yaw 30'*. All six basic motions can be expressed using the commands '*rpyt* $\theta_z\theta_y\theta_x\Delta_x\Delta_y\Delta_z$' and '*trpy* $\theta_z\theta_y\theta_x\Delta_x\Delta_y\Delta_z$'. *Rpyt* does the rotations first, then the translations; *trpy* performs the translations first. Note that the syntax for *trpy* is inconsistent in that although the translations are performed first, they are the last arguments specified for the command.

```
32 32 32
7

rpyt 0.0 0.0 0.0 15.0 15.0 15.0 end


end
0.7164 0.4030 0.4030 0 0 0 0 0 0 -87.3151 255.0 0.0
end 0


rpyt 0.0 0.0 0.0 0.0 -0.2944 -0.2944 end
       0.5835 0.3352 0.3352 0 0 0 0 0 0 -65.5430 -255.0 0.0
end 0


rpyt -18.0 0.0 0.0 3.52 0.0 0.0 end
       0.00923521 0.00116281 0.00116281 0 0 0 0 0 0 -0.028607 31.0 0.0
end 0


rpyt 18.0 0.0 0.0 -3.52 0.0 0.0 end
       0.02825761 0.00430336 0.00430336 0 0 0 0 0 0 -0.1851891 31.0 0.0
end 0


rpyt 0.0 0.0 0.0 0.0 5.6 0.0 end
    0.00390625 0.00275625 0.00275625 0 0 0 0 0 0 -0.04410 23.0 0.0
end 0


rpyt 0.0 0.0 0.0 0.0 1.6  0.0 end
      1.0 1.0 1.0 0 0 0 0 0 0 -0.541696 47.0 0.0
end 0


rpyt 0.0 0.0 0.0 0.0 -1.6  0.0 end
      1.0 1.0 1.0 0 0 0 0 0 0 -0.541696 47.0 0.0
end 0
```

Figure 3.8: Example 3Dsyn input file (A synthetic head)

After the header block there comes a set of *section* blocks, one block for each quadric section to synthesize. Each block describes a surface to generate, and all the constraints for the surface. The quadric section blocks are in turn composed of smaller blocks. The first sub-block is a transform block which converts from the section coordinate system to the base coordinate system. This is followed by a *surface* block. A surface block contains a set of quadric coefficients to describe a single quadric surface, the interior and exterior densities of the object, and also contains a transform block which translates the surface's local coordinate system to the section coordinate system. After the surface block comes the number of constraints, followed by a set of constraint blocks. A constraint block is identical to a surface block except that it contains a flag indicating which way the constraint inequality goes (e.g., choose all points in the surface *below* some plane).

3.3 Matte - synthesize luminance images

Name: ifsmatte.c

Action: Produces a matte luminance image given a range image and one or more light sources of any brightnesses.

ifsmatte converts a 2d range image into a matte luminance image.
USAGE:
ifsmatte [flags] [Lfile [Rfile [Mfile]]]

All of these command line arguments are optional. Read on for a description of the flags and filenames. If the filenames are present, they must be in the order shown above. Some examples of good and bad usage follow...

ifsmatte (good: input will be interactive)

ifsmatte -h (good: displays this helpscreen)

ifsmatte Lfile Rfile Mfile (good: This is the most common usage. Light sources scanned from Lfile, range image scanned from Rfile and matte image written to Mfile.)

ifsmatte Rfile Lfile Mfile (Error! Files out of order!)

FLAGS: command line flags are...

```
-h  short help-screen
-H  long  help-screen
-d  debug output
```

```
-n  turn off 0-255 scaling
-f  float output; default is ubyte
-b  process background pixels also
-v  echoes version, history, date, etc
```

LIGHTS: This program generates a matte image illuminated by a set of light sources that you define. You can either store the light sources in a file and then enter the file on the command line (like Lfile in the examples in the 'usage' section, above), or you can enter the lights in response to prompts (by leaving the the filenames off of the command line).

(Example) Here is what a light source file would look like for 2 lights sources, centered above a 300x300 image, with a bright light (500) to the left (column zero) and a dim light (100)to the right (column 300).

```
2               /* number of lights */
150 0 1000 /* coordinates of light 1 */
500 /* brightness  of light 1 */
150 300 1000 /* coordinates of light 2 */
100         /* brightness  of light 2 */
```

Notes

1. Comments (like in the example) aren't allowed.
2. Careful with the z coordinate. Large positive z values place the light source in front of the object (good). Negative z, or even small positive z, may place the light BEHIND the object (bad), which may generate a null image.
3. When you sit directly in front of the parallax looking at a 300x300 img your face is approximately at coord- inates 150 150 1000. Use this as a reference when positioning light sources.
4. 50 lights max.

Files: if given on command line, must be in order shown below...

```
Lfile - Input;  holds light data; see above description.
Rfile - Input;  ifs range image, 2d, assumed float
Mfile - Output; ifs matte image, 2d, ubyte unless -f is used
```

Algorithm:

```
    if (filenames on command line)
        read lights from Lfile
    else
        interactive input;
    call ifsderiv to compute gradients of range image
    for every pixel
        Compute normal to thispix;
        Compute vector from thispix to each light;
        Take dot product of normal and light vectors;
        Compute cosine of angle between normal and lights;
        Light thispix by summing the following...
\begin{displaymath}
            pixbrite = \sum_{i = 1}^{num} (light_i \cos \theta_1 )
\end{displaymath}
```

FLAGS:command line flags are...

```
-h  short help-screen
-H  long  help-screen
-d  debug output
-n  turn off 0-255 scaling
-f  float output; default is ubyte
-b  process background pixels also
-v  echoes version, history, date, etc
```

USAGE: Some examples...

```
matte                       \{Good. Interactive input\}\\
matte [flags]               \{Good. Will prompt for files\}\\
matte Lfile Rfile Mfile     \{Good. Files in order\}\\
matte Mfile Rfile Lfile     \{Error! Files out of order\}
```

3.4 Tomosim - simulate tomographic X-ray source

This program to simulates a 3-D beam tomographic sensor Either cone-beam or parallel-beam sensors may be simulated An ifs 3D image is used as input (as produced, for example, by **3dsyn**) The program produces a set of ifs 2D images, where each of the output images corresponds to one projection

USAGE:

```
tomosim  inputimage numofpoints radius detector\_rows detector\_cols<-o> <-d n>

  IFSIMG inputimage; /*three dimensional*/
```

```
  int numofpoints; /*The number of points around a 360 degree circle,
*/
   /*  centered at the */
   /*  center of the volume specified by the input image*/

  int radius;      /*The distance (in voxels) from the center of the */
   /* volume to the detector. Will be treated as    */
   /* identical as the destance from the     */
   /* center of the volume to the center of the detector array*/
  int detector_rows;  /* number of rows on the detector*/
  int detector_cols;  /* number of columns on the detector*/
```

The names of the output files are read from stdin as they are needed. Since the program may take quite a while to run, manually typing these names is tedious. The recommended way to run the program is to first create a file containing the names of all the output image files, and then to run **tomosim** redirecting stdin to this file.

Switches

-o The -o switch, if used, will use parallel-beam (orthographic) rather then conebeam projection

-d The -d switch, if used, means that the next argument will be the debug control value (really only of interest to the programmer)

Application: For cone beam, just specify the number of points. The sensor rotates in x-y plane, about an origin at the center of the 3D volume provided. For fan beam, simply specify a detector with only one row. The -o switch provides parallel beam simulation in either single row or multiple row cases.

Chapter 4

Programs for processing images

The following is a list of programs which exist in the ifs bin directory. (Depending of local installations, this is usually /usr/local/bin/ifs)

These programs are for the most part, simply "mains" wrapped around some of the standard subroutines documented in earlier chapters. These programs are only documented briefly here, since the operation is generally obvious.

Generally, on-line help for any program can be obtained by simply starting that program up, but providing it an incorrect number of arguments.

add – add two ifs images, point by point Author: Wes Snyder

addhdr – adds an IFS header to a raw data file. "rmvhdr" is the reverse function. Perpetrator: Mark Lanzo

atoi – Converts an ascii file to ifs. Input file is to be in the format produced by itoa using the -v switch. The -v switch on itoa adds two lines at the beginning of the file which specifies the size and data type. Author: Wes Snyder

c2imag – take imaginary part of an ifs image, point by point Author: Wes Snyder

c2mag – take magnitude of an ifs image, point by point Author: Wes Snyder

c2phase – take phase of an ifs image, point by point Author: Wes Snyder

c2real – take real part of an ifs image, point by point Author: Wes Snyder

compmag – produces an ifs file (type float) equal to the log of the square of the magnitude of a complex image Victim: Wes Snyder

ipde_to_ifs – converts ipde format images to ifs Author: Gary McCauley

itoa – prints an IFS 2D image in ascii format. Author: Mark Lanzo

mkdoc – makes a LaTeX compatible version of this index on standard out. Author: Wes Snyder

mult – multiply two ifs images, point by point Author: Wes Snyder

profile – Take a cross section of an IFS 2D image. Output is in standard plot filter format (to stdout). Author: Mark Lanzo

prthdr – Print the header structure for an IFS image (in human readable format). Author: Mark Lanzo

rmvhdr – Remove the header from an IFS image to yield a raw data file. Author: Mark Lanzo

subsample – subsamples an arbitrary ifs image to be of a specified size: Author: Wes Snyder

recip – take reciprocal of an ifs image, point by point Author: Wes Snyder

sub – subtract two ifs images, point by point Author: Wes Snyder

vidscale – video scale an ifs image

window This program extracts a window from an ifs image. The resultant output image is of the same data type as the input. Call: window input output xleft ylower xright yupper

input and output are two dimensional ifs image files. output will be created by this program.

xleft is the index of the left-most column of the input image which should be in the window.

ylower is the index of the lowest-index row of the desired window.

xright and yupper are the other extremes. NOTE: yupper must be greater than ylower. Thus, upper and lower correspond to indices, not to a top-bottom relation on a display screen. Author: Wes Snyder

Chapter 5

Programs for displaying images

Any X-11 device can be used to display ifs images.

5.1 IMP - system for displaying, manipulating, and processing ifs images

IMP is a new package which supports many of the features of X11 release 4. It includes a number of processing functions as well as display. Complete documentation on IMP is available as a separate publication.

5.2 Xdisp - driver for X-windows devices

Xdisp is a general-purpose display driver which will display IFS images on any display which supports Xwindows, version 11. If the device is only binary, the user may select to view either a thresholded version of the image (with user-defined threshold), or a dithered version.

Usage:

```
xdisp [-f filename] [-z zoomxy] [-l threshold] [-s] [-h]
```

Options:

```
-f flag: File name that contains ifs image.
```

```
        If not specified, program prompts for name.

-z flag: Specifies integer zoom. Default is 1.

-l flag: Specifies threshold for bitmap displays. All pixels
         below threshold are displayed black. Default is 0.

-s flag: Specifies shading on bitmap display. Uses 2x2 random
         dither which gives 5 shades of gray.

-h flag: Prints this help statement.
```

Note: Other X options like -display,-geometry etc. can be used. The opened window can be dynamically resized, as allowed by the window manager. Clicking the image after resizing will either

1. center the image within the window, if the resized window is bigger than the image, but not bigger than 1.5 times the size of the image, in which case the image is automatically zoomed in that dimension

OR

2. shrink the image to a smaller size if the resized window is smaller than 67

The horizontal and vertical scrollbars are used to scroll the image within the window, if it is smaller than the image.

In this version of xdisp, only displays with $depth \leq 8$ planes are supported. Furthermore, it is optimized for single bit and 8 bit plane displays.