Data Reduction for the Deepsearch

by R. Knop
with S. Deustua

Version 1.4.0

2002 June 12

Chapter 1

Images & Diagnostic Tools

1.1 Deeplib and IRAF

There are two different tools you may choose between for doing the bulk of your photometric data reduction.

IRAF NOAO's "Image Reduction and Analysis Facility" is a huge and powerful package, and the closest thing there is to a standard for reduction of optical images. It has most everything you need, and is a robust and well-tested system. Unfortunately, it's cantankerous, in many ways annoying, and a bit too much of a black box. Every command has so many options that it's easy to do the wrong thing without realizing it. IRAF tools must be run from within IRAF.

You may wish to learn IRAF and do most of your data reduction therein. There are some final steps which will require other software, but you can get most of the way there entirely with IRAF. IRAF is a standard, and is well supported, and there are some advantages to doing this.

This manual will assume that you will be using Deeplib (see below) for most of the reduction. However, there will be some steps which require either IRAF (e.g. splitting up an image stack) or Deep/IDL (e.g. loading images into the database).

Deeplib Deeplib refers both to a C++ library and a set of utilities being developed by Rob Knop, Alex Conley, Michael Wood-Vasey, and Nicholas Regnault (with some help from a few others) for use with SCP data and database. The Deeplib utilities all run from the command line. They often have a lot of options, although defaults tend to be set to "sane" values for typical Deepsearch data. They do not support parameter files as does IRAF, which while in some cases less convenient, also avoids the trap of your using the defaults for a different

data set on the current data set without realizing it. With almost any Deeplib utility, you can get a description of how to run it by running the command with the single argument "`--help`"; for example, try "`overscan --help`". Although many tools immediately useful for data reduction are fully usable, Deeplib is still under development. Most of the Deeplib utilities on computers where it is installed can be found in `/usr/local/deeplib`, although this may vary on your system. If you are lucky, the utilities are already in your path and you don't have to think about it.

Deep/IDL IDL is standard data reduction language. "Deep/IDL" refers to the large collection of software which has been written for the Deepsearch in IDL. There are some steps of data reduction (particularly towards the end) which are either easier with Deep/IDL, or which require Deep/IDL. To use Deep/IDL, your system must be set up with the right IDL paths and so forth. Talk to your local Deepsearch computer guru.

1.2 Command Line Conventions in this Docuemnt

Note that many Deeplib (or other Unix command-line) commands below will be longer than fit on one line of the page. These will have a backslash (\) at the end of every line but the last line. If you are typing on the Unix command line, and you have a backslash as the very last character of the line (i.e. with no spaces after it), Unix will know that you're not done typing the command yet, and will prompt you for more. Alternatively, you may type the whole mess on a single line (perhaps wrapping if there are more characters tha fit across the width of your terminal). For any Unix command in this document which is very long, *omit the backslashes if you are typing the command into a single line on the terminal.*

1.3 Terminology

This manual will glibly throw around a number of terms, the definitions of which are necessary to know in order to understand the text. These are some of the more important terms:

image The term "image" means two things. The most basic definition is a two-dimensional collection of pixels, that together represent the spatial distribution of light on a given region of the sky. The second definition is a file in memory or on disk that stores the image data. This second definition is sometimes referred to as an "image file." Note that in the process of data reduction, you will create "images" that don't represent any given region of the sky.

frame Sometimes used as a synonym for "image." Also a Deep/IDL command.

flatfield (1) A noun that means an image used to calibrate out gain variations in the pixels of an image. A "combined flat" is one that has been put together from several individual flatfields. Sometimes, in the case of sky flats, your individual flatfields may be the same as your target images. (2) A verb indicating the act of applying a flatfield correction to an image.

zero image (Also sometimes called a "bias image".) This is an image that has the pixel-to-pixel variations and overall structure of a "zero-time" exposure. You want to to subtract this structure from your data. A "combined zero" is a better zero image which has been created from several (5-11, usually) individual zero frames.

reduce The term "reduce" has unfortunately acquired a nonstandard meaning in the Deepsearch, and so tends to be used ambiguously. In Astronomy in general, to "reduce data" is to take the raw data from a telescope, and apply whatever corrections and calibrations are necessary to produce a usable image from which measurements can be directly taken. Among the Deepsearch, this process has traditionally been called "cleaning". You will occasionally hear the terms "reduce" and "clean" used interchangeably. The second definition of "reduce" comes from the Deepsearch IDL routine `reduceimages`, which finds all of the objects on a frame, stores their positions and fluxes in a file, and calculates and stores some rough estimates of the sky noise, atmospheric blurring, and photometric solution for an image. This second form of "reduction" is normally done on images which have already been "reduced" (or "cleaned") in the first sense of the word.

observer The person who was at the telescope taking the data. Also known as the "yahoo at the telescope." You are not the observer. You are the data monkey.

RA and Dec Right Ascension and Declination. The standard celestial coordinate system. Pretend that the sky is a spherical shell about us, with stars and galaxies painted on it. RA and Dec are coordinates on the inside surface of that spherical shell exactly analogous to the coordinates longitude and latitude as used on the surface of the earth. A declination of 0 indicates the equator; positive is Northern hemisphere, negative is Southern hemisphere. RA is measured in hours, minutes, and seconds, for reasons that make a lot of sense if you think about what happens on the sky when the Earth rotates. 24 hours of RA corresponds to 360 degrees of RA.

IRAF See Chapter 666, "Hell".

1.4 FITS Images

Although occasionally you will run across different file formats, the most popular image format, and about the only image format used by the Supernova Cosmology Project (SCP), is the FITS format. FITS stands for "Flexible Image Transport System," but that isn't really important. FITS images usually have filenames that end in "`.fts`", "`.fits`", or "`.fit`".

This document will only talk about "simple" FITS images. There are FITS extensions, and FITS image stacks, and FITS tables, and any number of other FITS variants, many of which you will run into if you reduce Hubble Space Telescope (HST) data.

Unless you are using IRAF and `mscred`, most of the steps of the data reduction will be performed on "simple" FITS images. Many modern cameras (including the WIYN MiniMosaic and CTIO Mosaic cameras) have multiple chips, and return extended FITS files or image stacks. In this case, the first thing you must do is break the image stack out into individual images.

A simple FITS image is divided into two logical components: the header, and the body or image data.

1.4.1 FITS Headers

The header is a series of character strings that gives you information about an image. The exact specification of a header varies distressingly from telescope to telescope, but normally there is enough information in the header of an image to tell you almost everything you need to know to reduce that image. Each header record has a keyword, a value, and an optional comment. Typical information encoded in a good header includes the coordinates (right ascension and declination) of the observation, the time (date and UT) of the observation, the name of the telescope and detector, and the length of the exposure (usually in seconds). Additionally, a good header has other information about the camera, such as the gain, any modifiable operating parameters. Finally, there is usually a "title" or "object name" which the observer at the telescope is able to set before an observation.

WARNING: Fits headers are mercurial. Different observatories have different standards. ESO, in particular, has a very strange idea of what a FITS Header ought to look like. (If you have images from the VLT, NTT, or another ESO telescope, use the Deeplib command `dehierarch` to change the FITS headers into something sane.) Be very careful to insure that what you find is what you are looking for. For instance, most headers report the "gain" in units of e-/ADU, but some report the inverse as the "gain". As another example, the equinox quoted in the WIYN header

is usually wrong; what is quoted is the date of the observations, *not* the equinox of the coordinate system.

The following is an excerpt from a header produced by the LRIS camera on the Keck telescope:

```
SIMPLE  =                    T / Fits standard
BITPIX  =                  -32 / Bits per pixel
NAXIS   =                    2 / Number of axes
NAXIS1  =                 1640 / Axis length
NAXIS2  =                 2048 / Axis length
EXTEND  =                    F / File may contain extensions
ORIGIN  = 'NOAO-IRAF FITS Image Kernel Aug 1 1997' /
DATE    = '24/03/99  '         / Date FITS file was generated
IRAF-TLM= '16:16:07 (24/03/1999)' / Time of last modification
OBJECT  = 'Keck031 '           / Name of the object observed
TRAPDOOR= 'open    '            /
SLITNAME= 'direct  '            /
GRANAME = 'mirror  '            /
REDFILT = 'I       '            /
UT      = '06:47:32.56'         /
AIRMASS =           1.00247227  /
TARGNAME= 'Keck031 '            /
RA      = '23:20:21.50'         /
DEC     = '+15:56:08.1'         /
EQUINOX =                 2000  /
TELESCOP= 'Keck II '            /
PONAME  = 'LRIS    '            /
FRAMENO =                  202  /
OBSNUM  =                  202  /
EXPOSURE=                  120  /
NUMAMPS =                    2  /
AMPLIST = '2,1,0,0 '            /
COMMENT = '* This image was generated by the Low Resolution Imaging'
COMMENT = '* Spectrograph'
```

Although there is more in the Keck LRIS headers than is shown here, this illustrates the sorts of things that may be found in a header. The names in all caps at the beginning of each line are the *keywords*. The information between the equals sign and the slash on a line is the *value* associated with that keyword. The text after the slash is the optional *comment*, and is present merely for the illumination of humans reading the header. Sometimes you will see `COMMENT` or `HISTORY` keywords where there is no equals sign; this is allowed according to the FITS standard.

When you are trying to figure out what an image is, or you need information about it such as what was the filter used during the observation, there are two places you should look. One is the header of the image. The other is the logsheets taken by the observers during the observation. It varies which is more reliable. For most information, such as coordinates and filters, the header is usually more reliable. For the titles of the images, the logsheets are usually more reliable. (This is because observers tend to forget to update the titles of images before taking observations.) Note that the *best* way to determine what the title of an observation should be is to trust neither the object keyword in the header nor the logsheet. Rather, look at the RA and DEC keywords in the header, figure out the position of the observation, and then compare that to a target list. Normally this procedure is only necessary when resolving confusion or fixing problems. Sometimes, even this doesn't work, as some telescopes have been known to put incorrect coordinates into the image headers. (At that point, the problem gets harder, but its solution is well beyond the scope of this manual.)

Too look at a header in Deeplib, use the command `imheader` from the Unix command line:

```
% imheader filename | less
```

This will show you the contents of one header (piping the output through the pager `less`; you may substitue `more` or another way of viewing the text output of a command). Once you figure out what the important keywords in a given set of data are, you can get a summary of all the data files by using a command something like:

```
% hselect *.fits -v EXPTIME,FILTER,OBJECT
```

Of course, you will adjust the pattern appropriately to show the files you wish to list. (For instance, if you have data from a multi-chip camera, the header information will usually be the same across all chips. As such, you may only need to show the header information from a single chip, and might use "`*_1.fits`" rather than "`*.fits`".) Similarly, you will want to make sure that the keywords you list are right for this run, and you may wish to look at additional keywords.

1.4.2 FITS Image Data

The image data in a FITS image is stored after the end of the header, in a standardized way that all software which can read FITS images (including IRAF) should recognize. The FITS specification allows for images to be of any dimensionality between one and 99. This document only deals with normal, two-dimensional images. Such images

have a "NAXIS" header keyword with a value of 2. "NAXIS1" gives the width of the image (in pixels), and "NAXIS2" gives the height of the image. All three of these keywords should be near the top of the header. Normally, you will never have to think about this because FITS reading software understands how to take care of it.

Signal and Noise

Each pixel represents the amount of flux (light) that is in the portion of the image covered by the size of that pixel. (Really, it's related to a number of detection photoelectrons, which is in turn more closely related to a photon count than a real flux, but you shouldn't have to worry about that for purposes of reducing data.) For a normal raw sky image, this flux can come from a number of different sources:

$$f = b + d + s + i$$

Where f is the total flux in the pixel, b is the bias value (a constant offset, which can be positive or negative), d is "dark current" (flux that accumulates in the pixel even when no light is falling on the CCD), s is sky background, and i is the actual image data that we care about; usually, when somebody refers to the *signal*, i is the value they mean.

Each pixel has a *noise* value associated with it as well. The noise is the fundamental limitation to how well you understand your signal. The symbol σ is usually used to represent noise. Sometimes, it is more convenient to work with σ^2, called *variance*. The *signal to noise ratio* (S/N) is just that; higher S/N indicates a better measurement. One primary purpose of data reduction is to eliminate as many sources of noise as practical, so that the signal may be measured as well as possible given the limitations of Physics. The following expression lists some of the sources of noise in an image:

$$\sigma_{tot}^2 = \sigma_r^2 + \sigma_z^2 + \sigma_d^2 + \sigma_f^2 + \sigma_s^2 + \sigma_i^2 + \sigma_?^2$$

σ_r is "readout noise," a parameter of the CCD. σ_z is "zero noise," which is any variation in the bias of the chip from pixel to pixel; this noise is usually eliminated during the "zero subtraction" step of image reduction. For most modern CCDs, both of these sources of noise are relatively insignificant compared to the noise from the sky background. (This is generally *not* the case with the HST, where the sky background is extremely low.) σ_d is the noise due to the dark current, and ideally has a value $\sigma_d = \sqrt{d}$. σ_f, or "flatness noise," is (effectively) noise due to pixel-to-pixel gain variations. Correctly performed flatfielding (see chapter 3) should eliminate the bulk of this noise. σ_s is photon noise in the sky background, and ideally has the

value $\sigma_s = \sqrt{s}$. σ_i is photon noise from the object observed, and ideally has the value $\sigma_i = \sqrt{i}$. In the case where $i \ll s$ (which is true for most of the ground-based observations of supernovae in the Deepsearch), σ_i is insignificant compared to σ_s. $\sigma_?$ is noise, measured in an image, of uncertain source. If you understand your images, and have performed a careful reduction procedure, it should be very close to zero, but frequently the image quality of reduced images does not get exactly all the way down to the photon shot noise limit.

The Lower-Left Pixel

FITS images are normally stored and displayed so that the lowest numbered pixel is the lower left pixel. (This is in contrast to other computer images such as are found on the World Wide Web, where the lowest numbered pixel is the upper left pixel.) Frequently, it is useful to refer to positions on the image using a continuous coordinate system, i.e. floating point numbers, so that distances between objects can be given to precision better than one pixel. There are a number of different conflicting conventions for mapping pixels onto a continuous coordinate system. The difference can be summarized by indicating how each system defines the coordinates of the lower left pixel of the image (see Figure 1.1). There are two issues. One is whether the lower left pixel of the image is (0,0), or (1,1). The second issue is whether the "integer" coordinates of a pixel refer to the center of the pixel, or the lower left of the pixel. IRAF defines (1,1) to be the center of the lower left pixel. The IDL Deepsearch software defines (0,0) to be the center of the lower left pixel. The C++ Deeplib software, for the sake of simplicity and backwards compatibility, shares the definition of the IDL Deepsearch software. This is a tragedy, because he most logical and (alas) least common definition is that (0,0) is the lower left corner of the lower left pixel (and hence of the entire image). Rob Knop's (obsolete) Orchid software, uses this latter convention. Most of the time, you won't have to worry about this, as each image reduction software package will be internally consistent. However, if you write down pixel positions determined from one package (e.g. IRAF and ds9), and then wish to use these pixel positions in another package (e.g. the IDL Deep software), you will have to think and translate the pixel positions.

Note that *all pixel positions stored in the SCP databases use the IDL Deepsearch convention!* Although IRAF and the Deeplib software tries to be as internally consistent as possible (and Deeplib performs the necessary translations when reading both FITS images and the SCP databases), sometimes some painful by-hand translation will be necessary when explicitly specifying pixel positions to anything other than the IDL Deepsearch software.

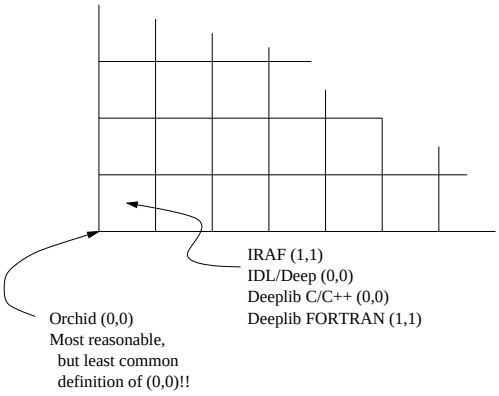

Figure 1.1: Conventions for the coordinates of the lower left pixel for different image reduction software packages.

Data Types

For most of astronomy, it is generally most useful to think about the value for the flux at each pixel as being a floating point number. Sometimes, images are stored on disk as short integers (16-byte integers). This may be because the data is entirely composed of integral pixel values in the range (-32768..32767) (those numbers which can be represented by a 16-byte integer). Frequently raw data from telescopes fit this description. Once data has been reduced, it usually no longer fits this description. However, often there is no more than 16 bits of real information in the image. In this case, the amount of disk space used by the images may be reduced by storing the image as 16-bit integers and encoding in the header of the image the translation to the full floating point value of each pixel. Most image packages handle the translation automatically; you will almost never had to worry about this issue when reading an image. Sometimes you will have to think about it when writing out images. (MORE TO BE WRITTEN.)

1.4.3 Data Compression

Occasionally, you will come across what looks like a FITS file that you can't read. Frequently, what you have is a FITS file that has been compressed using one of a number of different data compression algorithms. This is important for two reasons; one, because you need to know the program to use to decompress the data. Two, because there may have been information loss in the compression. Compression algorithms that cause some information in the image to be lost are known as "lossy" algorithms. You usually get more compression with a lossy algorithm then you do

with a lossless algorithm. When performing careful reduction of data, it is almost always better to find images which have *not* been through lossy compression.

Frequently, you can recognize the compression algorithm used on an image by the extension on the image (the letters in the filename after the last period). Of course, this requires that the person who named the file (who may have been you!) did it correctly. Table 1.1 summarizes the most common types of compressed programs you will run across.

File Extension	Type of Compression	Command to Decompress
.gz	lossless	`gzip -d` *filename*
.Z	lossless	`gzip -d` *filename*
.bz2	lossless	`bzip2 -d` *filename*
.H	lossy	`fdecompress` *filename*
.nz	lossy	*talk to Rob*
.mz	lossy	*talk to Rob*

Table 1.1: Types of file compression frequently found for Deepsearch FITS Images.

1.5 Running IRAF

Note: Although most of this manual will assume that you are using Deeplib, you may need to use IRAF for a small number of steps. This section will tell you how to get started running IRAF. In addition, it will give you some pointers at figuring out how to do most of the reduction in IRAF, should you wish to do it that way.

IRAF is a general image reduction package that many of us find extremely maddening, but which provides a vast array of tools useful for reducing astronomical images (as well as other things).

To set yourself up in the first place to run IRAF, first make sure your path is set up correctly. For many computers, this will mean adding `/iraf/bin` to your path. (On the Deepsearch Suns, add `/home/astro21/iraf/bin` to your path. On the Deepsearch PCs, add `/home/lilys/iraf/bin` to your path. (**WARNING: these may be out of date!**)) Check first to make sure it isn't already there. Once your path is set up correctly, you need to create yourself an IRAF directory, and run the `mkiraf` command. You can do that with these commands (given to the Unix command line):

```
% cd ~
```

```
% mkdir iraf
% cd iraf
% mkiraf
```

In order to actually run IRAF, you will find that things tend to work best if you first open an `xgterm` with the command:

```
% xgterm -geometry 80x40 &
```

Move your mouse so that the newly opened xgterm window has focus, and then issue the following commands to run IRAF:

```
% ds9 &
% cd ~/iraf
% cl
```

The first command runs the image display program (ds9) used by IRAF. If you already have an ds9 running, you can skip this command. Once you are in IRAF, you will see the IRAF prompt. This is a two letter prompt followed by a "greater than" sign (>). Before doing anything else, issue the command:

```
cl> stty xgterm nlines=40
```

If you are running an xterm, you would use `stty xterm` instead. Note that it is very important to give IRAF the correct number of nlines in an stty line. Without it, you will have a very hard time reading help files and editing parameter files.

The tasks within IRAF are organized in *packages*. In order to use the tasks in a package, you must first load that package. Some of the most common packages will be loaded for you in various startup files that are run as you start IRAF. To load another package, simply type its name. For instance, if you want to load the `ccdred` package (which includes many useful tasks for basic image reduction of single-chip cameras), you would type that command to IRAF:

```
cl> ccdred
cc>
```

Notice that the two characters of the IRAF prompt tell you what your current package is. The IRAF command `help` lists the tasks and help topics available in the current package. The command `package` will list which packages you have loaded. For more information, do:

```
cl> help package
```

The `help` command, of course, works with other IRAF commands. Use it early, use it often. To exit the current package, use the command `bye`. If you are confused, keep using `bye` until it gives you an error message; then you know you're back to where you started.

If at any time you hit CTRL-C to abort an IRAF task, or IRAF gets otherwise confused, you should use the command

```
cl> flpr
```

repeatedly. This is pronounced "flipper", and stands for "flush the process cache" or something equally banal. If you don't do it, IRAF might get in a confused state that will cause further problems. It never hurts to do lots of flippers.

Many IRAF tasks have a huge number of parameters and options, which can be specified on the command line. However, frequently it is easier to use the IRAF parameter editor, an editor somewhat similar to *vi*, to specify all of your parameters. To use the IRAF parameter editor, do:

```
cl> epar command
```

Use the arrow keys to move up and down from one parameter to the next. There may be more than one page of parameters. When the cursor is positioned on the line of a parameter you wish to edit, type the new value of the parameter. When you are done, hit the colon (:) key, and type "wq" followed by a carriage return. This saves the values of the parameters; they will have the same values the next time you enter IRAF. Alternatively, instead of ":wq", you can use the command ":go" to immediately run the IRAF command with the parameters specified. The ":go" method is usually the easiest way to run interactive IRAF commands.

To quit IRAF, use the command `logout`:

```
cl> logout
%
```

1.6 Inspecting Images with IRAF

Header

The three most useful commands for sniffing image headers in IRAF are `imheader`, `hselect`, and `ccdlist`. The command "`imheader` *filename*" tells you the size and name of the image stored in file *filename*. To see the full header, try:

```
cl> imheader filename l+ | page
```

The "| page" at the end of the line tells IRAF to send the output through a pager similar to Unix's `more`. The "l+" tells IRAF to show the full header. (This is actually a shorthand version of editing the `imheader` parameters with `epar` and setting "longheaders" to "yes".)

The command `hselect` is useful when you need to see a small number of header values from a large number of files. For instance, if you want a list of the OBJECT, RA, and DEC fields from all of the FITS images in the current directory, you could use the command:

```
cl> hselect *.fits $I,RA,DEC,OBJECT
```

(You may wish to add "`| page`" to the end of this command. You may alternatively redirect the output to a text file by appending "> *filename*" to the end of the command line, just as in Unix.) The first parameter is the name of the files you wish to show; the wildcard in this example selects all files whose names end in "`.fits`". The second parameter is the list of header keywords you wish to see. The "`$I`" tells IRAF to print the filename of each file whose header keywords will be listed.

The command `ccdlist` is a part of the `ccdred` package. Try the command:

```
cl> ccdlist *.fits
```

(Remember to load the `ccdred` package first if you have not already.) If there are any files whose names end in "`.fits`" in the current directory, you will see a line of information for each one. This includes the name, size, and title of the image. Additionally, among the last several sets of brackets, there will be information about the filter of the image (usually something like "R" or "I", although the name of the filter may be longer), and information about which reduction steps IRAF has performed on the image. When you have raw data, that latter set of brackets will be empty. Once overscan correction is done, there will be an "O". More letters will

be added for subsequent steps: "T" for trim, "Z" for zero correction, and "F" for flatfield correction. If you perform some of the steps outside of IRAF, unless those tasks were written to interface with IRAF, IRAF won't know about it. Always be careful, keep track of what you are doing, and don't trust IRAF to know everything for you.

Display and Imexamine

The IRAF command `display` shows a FITS image in the ds9 window you opened before first starting IRAF (see section 1.5). There are (at least) four image buffers in ds9, which may be selected using the "Frame" menu in ds9. To display to the first of these image buffers, use the IRAF command:

```
cl> display filename 1
```

Replace 1 with 2, 3, or 4 to use the other ds9 image buffers. This command uses a default mapping of image pixel values to greyscale colors. There are two ways to change this mapping. One is to hold down the right mouse button in the ds9 window and drag the mouse around. (Play with it; you will learn how it works.) The other is to use the various parameters to `display`, which may be edited with `epar` as usual. The "Control Window Gadget" brings up a small control window that lets you adjust the color map, zoom, and other properties of the display. The best way to learn about this and other features of ds9 is simply to play with it. Note that the coordinates displayed in ds9 use the IRAF convention (see Section 1.4.2). The first two values are the X and Y coordinates of the pixel, and the third is the pixel value, or flux in the pixel. Note that ds9 may only know about the limits of the display as passed from IRAF (i.e. the highest pixel value mapped to black and the lowest pixel value mapped to white). If the actual pixel value is outside the range understood by ds9, a "+" or "-" will appear in addition to a number on the ds9 window. In general, it is best to use the pixel values reported by ds9 for only quick qualitative diagnostic purposes. There are other IRAF tasks for determining real pixel values and statistics.

The IRAF task `imexamine` is a general purpose task that lets you determine statistics, perform quick aperture photometry, and otherwise inspect images. You may type the command by itself on an IRAF line following a display command, e.g.:

```
cl> display lris0203.fits 1
cl> imexamine
```

After this, the ds9 window should have focus with a circular cursor. Positioning this cursor over a region of interest and hitting various keys should give you information

about the image. This information will usually appear in the terminal window in which you are running IRAF, so it will frequently be necessary to shuffle windows in order to see what is going on. The first thing to try is to hit the ? key. This gives extensive help in the terminal window. When you are done looking at the help, hit the q key (in the terminal window), and you should be returned to the ds9 window.

The "m" key, pressed while the ds9 window has focus, reports statistics (mean, standard deviation, etc.) on a local region of the image. The text appears in the window in which you are running IRAF, but focus is returned immediately to the ds9 window. On an image which has been fully reduced, hitting the "m" key over a region of no stars should give you an estimate of the sky brightness and sky noise (see Section 1.4.2). To change the size of the box used for statistics, you have to change the values of the "ncstat" and "nrstat" variables. To do this, press the colon key while the ds9 window has focus. This will return focus to the IRAF terminal window. Type the name of the variable, and its value, followed by a carriage return. For example:

```
: ncstat 5
```

Focus will return to the ds9 window, and next time you hit "m" the new value will be in effect.

There is a huge number of variables, which affect different parts of imexamine, which may be set similarly. There are also a huge number of commands that give you various other information via imexamine. Reading the help information (by hitting the ? key) is the best way to learn about all of it. A couple other very useful commands in addition to the "m" statistics command are "s" (to get a surface plot in yet another window of the local area), "v" (plot a one dimensional "vector" or cut in a window; it will prompt you to indicate the other end of the vector), and "a" (perform aperture photometry, a complicated topic even in the simplified environment of imexamine).

Play with imexamine, to get a feeling for what it can do.

Greyscale Control in Display

By default, when you use `display` to display an image, IRAF picks what it think is a reasonable range for the greyscale, and usually it does a good job. However, sometimes you want to see a different range. You can do this by changing some parameters of display, either on the command line, or using `epar`. Display follows these rules for determining the greyscale mapping:

1. If the parameter `zscale` is set to "yes", then IRAF automatically picks a range

for the greyscale based on the mean and standard deviation of the image.

2. If `zscale` is "no" and `zrange` is "yes", then IRAF maps the lowest pixel in the image to black, and the highest pixel in the image to white. (Using the right mouse button in display, you can cycle the colors to flip this around.)

3. If `zscale` and `zrange` are both "no", then IRAF maps the pixel value `z1` to black and `z2` to white.

It is frequently easiest to edit all these parameters with epar. If you want to set them on the command line, you could specify that the greyscale should stretch the colors between pixel values of 0 and 1000 with the following command:

```
cl> display filename 1 zscale- zrange- z1=0 z2=1000
```

Seeing the Whole Bloody Image

You may have noticed when displaying a large image that the whole image doesn't get displayed with the IRAF task `display`. Look at the edge of what is shown in ds9; use the panner to make sure you are looking at the very edge. The pixel coordinates at the edge may not be the minimum (1) or maximum pixel coordinates of the image as shown by `imheader`. There are two ways around this.

Iraf uses a "device" called "stdimage" to determine the capabilities of the ds9 display. By default, stdimage is "imt2048". This is capable of showing a 2048×2048 image. If your image is bigger than that, the central 2048×2048 section of the image will be displayed. To display a larger image, if you have the memory available in your computer, you can tell IRAF to use the "imt4096" device (capable of showing up to 4096×4096 images):

```
cl> set stdimage=imt4096
```

After that, run display.

The second way around it is to just display a corner of the image. Suppose you want to look at the pixels all the way to the edge on the lower left of the image. You can display just a section of the image:

```
cl> display filename[1:1024,1:1024] 1
```

Imstat

The iraf task `imstat` is a very useful tool that performs statistics on regions of images. It is very simple to use; the command

```
cl> imstat lris0203.fits
```

will return the mean, standard deviation, and other information about the image lris0203.fits. If you want to get the statistics of a region of an image, the syntax is (for example)

```
cl> imstat lris0203.fits[900:1100,900:1100]
```

This will return the mean and standard deviation for the specified region of the image 201×201 pixels in size. What imstat does is nearly identical to what the "m" key in imexamine does, but it is easier to control. You can, with ds9, locate a small region of the sky that you know contains no objects (and hence only sky background). Write this region down. You can use imstat before and after a given step of the data reduction, to track how your sky noise ("STDEV", the standard deviation) was affected by that step. Explicitly specifying the same region before and after insures that you are looking at the same only-sky region of the image, whereas with ds9 and imexamine you have to have good aim. Note that if you perform a step such as trimming that may cut out part of the image or otherwise move pixels, you may not want to look at exactly the same range of pixels before and after the reduction step! As always, pay attention to what you are doing, and display and inspect images as often as possible.

1.7 Deeplib

Deeplib is two things:

1. A C++ library written (so far mostly) by Rob Knop to allow reading/writing of FITS files as well as interaction with the Deepsearch database.
2. A set of standalone Unix programs, linked with the Deeplib C++ library, to do image processing tasks.

Deeplib is currently very much under construction, and the library is not available for general use. However, a handful of the individual programs are available for general use.

1.7.1 Setting up Deeplib

Deeplib may be set up for your environment already. If not, try adding `/usr/local/deeplib` to your path. At that point, the Deeplib utilities should be available. By convention, Deeplib programs give a brief help message if you run them with the single argument "`--help`".

1.7.2 Inspecting Images with Deeplib

The Deeplib program "`imheader`" will list the entire header of an image. As this is a standard Unix command line utility, you can do fun things by piping the results into `less`, `grep`, and so forth. If you want to select a small number of keywords from a large number of utilities, try the Deeplib program "`hselect`".

Deeplib's display program is currently called "`testfitswidget`". Run this with the name of the file you wish to view.

1.8 Deep IDL

Before doing anything with IDL, and with the Deep IDL software in particular, take a look at the documentation for the Deep IDL software on the web, which may or may not be at

`http://panisse.lbl.gov/collab/documentation/`

In particular, read the introductory section on setting up and living within the IDL environment:

`http://panisse.lbl.gov/collab/documentation/settingup.html`

1.8.1 Inspecting Images with Deep IDL

TBW; in the mean time, try "`dlib,'imagezoomer'`".

Reading FITS Files

In IDL, you can read in any FITS file with the command:

```
IDL> im=readfits("filename",hdr)
```

This will read the image data into the variable "im" (as a two dimensional array), and the header into the variable "hdr" (as an array of strings. Note that there are better ways to read images that are in the Deepsearch database, but that will be discussed later. (ROB, FIX THIS.)

Sky

The Deep IDL program `sky` measures the level and noise in the sky background of an image. It uses an algorithm that attempts to reject any objects (stars, galaxies, cosmic rays, and other "bright" pixels), and then measures the sky from the rest of the image. You can run it with the command

```
IDL> sky,im,sky,noise
```

It will print the "average" sky level in the image represented by the two-dimensional array im, as well as the noise in the sky level ("sigma"). The variable sky will be loaded with the sky level, and the variable noise will be loaded with the value of the sky noise. Of course, you may substitute your own variable names for the last two parameters.

The `sky` program can be useful for determining whether or not a given data reduction step measurably reduced the sky noise in an image. You may also determine this by explicitly performing statistics on a region of the image which has no objects, using a program such as Deeplib's `testfitswidget`.

zimage and imagezoomer

This section is still to be written, but you may be able to figure it out for yourself. Try reading an image into image variable `im1`, and then issuing the following command:

```
IDL> zimage,im1
```

Was that exciting? When you are done, exit the window, and try:

```
IDL> zimage,im1[100:300,100:300]
```

Getting the hang of it? Try playing with the zero, span, and redraw gadgets in zimage.

Chapter 2

Preliminary Image Reduction

2.1 Anatomy of an Image

A raw image from a telescope is an array of pixels (see Section 1.4.2). Each pixel in the image file corresponds to a physical pixel on the CCD chip used to take the image. However, there are usually additional pixels in raw data that are for calibration purposes, and which do not correspond to physical CCD pixels. Figure 2.1 shows a sample single-amplifier image. If a the CCD chip used to obtain the image was 2048×2048 pixels in size, then the "Image Data" portion of the image will be 2048×2048 pixels, even though the size of the image data in the raw data file may be 2080×2048 pixels. The additional pixels usually represent what is called an *overscan* region. These pixels measure the zero-offset, or bias, value for the image. Because these pixels do not correspond to physical CCD pixels, no light hit them; in a reduced data frame, they should have a value of 0 (although in reality they will have by then been trimmed out). The bias value, as measured by the overscan region, must be subtracted from the entire image (see Section 2.3).

Figure 2.1 shows the simplest image, which is the sort most easily reduced in IRAF. In this example, the CCD only used one amplifier. Real CCDs sometimes use more than one amplifier, and each amplifier will have a slightly different bias (and gain), requiring (among other things) a different overscan region for each amplifier. The location of this overscan region will vary depending on the instrument used to obtain the data. Frequently, you can figure out where to find the overscan regions, and which portions of the image they apply to, by perusing the headers of the images, or reading the manual for the observatory and instrument in question. Sometimes, you have to figure it out yourself, using tools such as imexamine (see Section 1.6). Normally, it will be easy to tell where one region ends and another region begins by looking at the image, as there will be a discontinuity in the background level due to differences in the bias and gain of the amplifiers. Figure 2.2 shows a sample two-

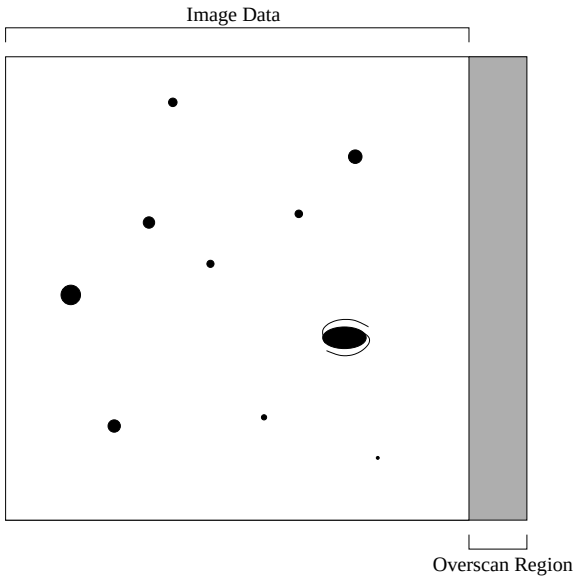

Figure 2.1: Example single amplifier image, showing the data section and the overscan region.

amplifier image, similar to those produced by the LRIS camera on the Keck telescope, and Figure 2.3 shows a sample four-amplifier image, similar to those produced by the ARCON system at the CTIO observatory in Chile.

Note that even though these images indicate specific positions for the overscan region, each instrument and observatory may place the overscan region in a different places. For instance, the overscan region of a two amplifier image might be on the right, as show in Figure 2.2, but it could just as easily be on the left, or in the middle of the two images. (In the latter case, it would then be very important to be extremely careful when trimming out the overscan region, so as not to distort the astrometry of the image across the amplifier boundary.) An overscan region may also be at the top of bottom of an image, instead of the left or right. Finally, some images won't even have an overscan region! Note also that this structure applies to all sorts of images, including zero and domeflat images, not just those images that show a region of the sky.

2.2 Multiple Chips

Many modern detectors, including the Mosaic on the 4m telescope in Chile and the MiniMosaic camera used at WIYN, have multiple CCD chips that take data simultaneously. The conceptually simplest way to reduce data from these telescopes is

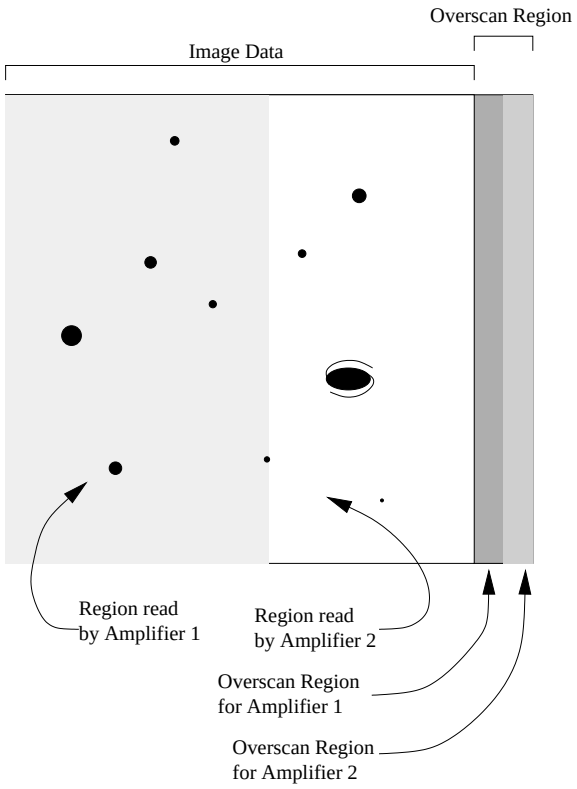

Figure 2.2: Example two amplifier image.

to treat each CCD as a separate instrument. In other words, you must build separate calibration images (zero, flatfield, etc.) for the data from each chip. There are some problems with this, however, and it can lead to challenges with photometric calibration later. The Deeplib tool "`pclipflat`" (and the parallel processing version "`parpclipflat`") allows you to simultaneously build zero frames (see Section 2.6) and flatfields (see Chapter 3) for all chips of a multi-chip camera; they will be normalized to be consistent with each other. Thereafter, you can reduce the data as if each chip were a different camera.

There are two common ways which data from multiple chips comes packaged. One scheme is to have the data come as a FITS image stack. Deeplib cannot handle image stacks, so you will need to use IRAF to split the stack. In IRAF, type `mscred` to get into the right package, and then `epar splitmsc` to set up the parameters for the command "`splitmsc`". This will split out a Mosaic image into individual files `filename_0.fits`, `filename_1.fits`, `filename_2.fits`, etc. Usually, `filename_0.fits` will be much shorter than the other files (use "`ls -l`" to check this) and will only contain a header. `filename_1.fits` will have the data for the first chip (or amplifier), and so forth. At this point, you may proceed with the reduction using Deeplib on these simple FITS images.

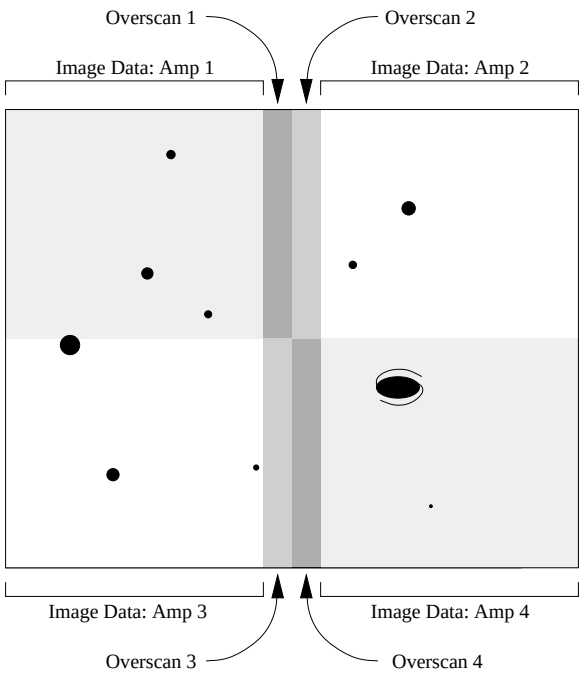

Figure 2.3: Example four amplifier image.

With some cameras, not only will separate chips, but separate amplifiers will be split out into individual files. In this case, you will later need to "knit" the files for individual amplifiers of a single chip back together; see Section 2.5 below.

The second way that multi-chip data may come packaged is in a large uber-image which has the data from the individual chips laid out next to each other; the TNG telescope returns data in this format. In this case, you should use the Deeplib utility `imclip` to extract the data for each chip into individual files. (When you do this, make sure to keep the overscan region for a chip with that chip's data, because you will need it below!)

Occasionally, there are also IRAF tools that will simultaneously build and apply multi-chip flatfields. While they will tend to be easier to use than Deeplib's `pclipflat`, they also can be a little challenging to configure for any instrument other than the specific one for which they were written. MOSRED is an IRAF package designed for the BTC, and MSCRED is an IRAF package designed for the Mosaic and MiniMos cameras. In the former case, a version of `ccdproc` that understands the BTC filename conventions operates on multiple files (one for each chip) at a time. In the latter case, the raw data from the telescope actually comes in files which are *image stacks*, as opposed to the simple FITS image discussed in this document.

If you've built separate flats for different chips (either by doing them one at a time, or by using Deeplib's `pclipflat`), make sure to apply the flatfield for chip 1 *only* to that data which comes from chip 1, and so forth. Flatfields are discussed in greater

depth in Chapter 3. The same concern applies to zero images (Section /refsec:zero.)

2.3 Overscan and Trim Correction

Overscan correction refers to the process of subtracting the bias of an image, as measured in its overscan region (see Section 2.1). *Trim* correction (sometimes called just "trimming") refers to clipping out those pixels on an image you wish to keep, usually to omit the overscan region after overscan has been corrected.

2.3.1 Overscan and Trim with Deeplib

Deeplib includes a program "`overscan`" which will do most of what you need for overscan correction. Run "`overscan --help`" for usage information. If you have a single image, a sample command line might look something like

```
% overscan image001.fits image001_new.fits -ov 2049 2079 \
   -kw OVERSCAN -p -pd /XSERV
```

Substitute the right overscan region for `2049 2079`. If the fit to the data does not look good, you may need to use the `-o`, `-sp`, or even `-nf` keywords on this data set. By default, `overscan` assumes that the overscan is done along the rows of the image, i.e. the overscan region is at the left edge or right edge (or *down* the middle with some multi-chip cameras). If the overscan region is actually at the top of bottom (or *across* the middle), use the `-c` keyword to tell `overscan` this.

When you have a large quantity of data to apply the correction to, it will usually be most convenient to write a simple shell or Perl script to run overscan on each image individually. (If you are really good, you can just do this from the command line with tcsh's "`foreach`" command.) Before running through a whole set of data automatically, you should carefully run it on a few images from that data set by hand, inspecting the results to verify that things are working well. In particular, use the `PLOT` parameter to verify that you are getting a good fit to the overscan region. When you use `PLOT`, you will need either to specify a plot device, either when prompted or with the `PLOTDEV` parameter. This is a standard PGPLOT device name. If that means nothing to you, just specify either "`/XWIN`" or "`/XSERV`" and things should work for you.

The simplest way to construct a shell script is to list the files you want into a text file (e.g. "`ls *.fits > dooverscan.com`"). Edit that file with `emacs`, and use the search-and-replace, rectangular cut and paste, and keyboard macro functions to

build up a series of commands. Save the text file, and then from the command line run

```
% source dooverscan.com
```

to run all of the commands in the text file. This is a generally useful procedure for doing a whole lot of repetitive, similar commands, and is best for people who are not comfortable writing perl scripts with `foreach` loops to handle the task.

You can perform trimming with the Deeplib program "`imclip`". Its operation should be obvious. Note that if you have to piece together an image with the overscan region in the center of the image (e.g. the image in Figure 2.3), you must be *very careful* to get your trim region *exactly* right. Normally, if you trim off an extra row or column at the edge of an image, it doesn't matter; however, if you trim off a column or row in the *middle* of the image, it will screw up the image (all the stars and galaxies to the right of a missing column will be incorrectly offset one column to the left, for example). See Section 2.5 below for information about knitting together individual amplifiers.

Use your favorite image viewer (e.g. the Deeplib program "`testfitswidget`") to verify that you know the proper overscan regions and the regions of the image to which they apply, and that you know the proper trim region. **Assume nothing!** Don't assume that it will be the same for a given detector and telescope combination from run to run, because it may change. Don't assume even that the header is correct (although the header can be useful in pointing you where to look, and in clarifying ambiguities). Always look at the image and verify for yourself that you know you are doing the right thing.

Choosing the Overscan Region

You may be able to find hints as to where to look for the overscan region by looking at the image's header. Look for the header keyword `BIASSEC`, or other keywords with promising-looking comments. In particular, if there is what appears to be an overscan strip on three or four sides of the image, the header can hopefully tell you which is the readout direction of the chip and which is the real overscan region.

It is worth taking care when selecting that portion of the overscan region you will use for estimating bias values. Don't always fully trust the header; verify what it says by looking at the image with an image viewer. Ideally, you can use all of the pixels on the overscan region in each row for this purpose. However, look closely at the overscan region to make sure that it doesn't get brighter toward one or the other edge. If they do, only use the consistent and reliable fraction of the overscan region for your overscan correction.

Multiple Amplifiers

If you are reducing data from a CCD that used two or four amplifiers, be very careful to apply the correct overscan region to the correct portions of the image. This may be very difficult to do in IRAF, unless you're using `mscred` and an image stack it knows about. The Deeplib utility "`overscan`" does let you specify explicitly which portion of the image you wish to work on each time you run the program; run "`overscan --help`" for more information. If you do have data with multiple-amplifier chips, you will need to run `overscan` on the image multiple times, once for each amplifier. For example, suppose that you have an image which looks like that in Figure 2.3. The commands you might use to apply the overscan correction to this image might be:

```
% overscan image001.fits image001_new.fits -ov 1024 1063 \
   -i 0 1023 0 1023 -kw OVERSCAN
% overscan image001_new.fits image001_new.fits -ov 1024 1063 \
   -i 0 1023 1024 2047
% overscan image001_new.fits image001_new.fits -ov 1064 1103 \
   -i 1104 2127 0 1023
% overscan image001_new.fits image001_new.fits -ov 1064 1103 \
   -i 1104 2127 1024 2047
```

Notice that while `image001.fits` is used as the input image for the first command, `image001_new.fits` is used for subsequent images. If you think about this for a moment, it should be obvious; if not, discuss it with people until it becomes obvious.

With this image, you would then have to use `imclip` twice and then knit the image back together (Section refsec:knit). If the overscan regions are at the left and right edges of the images, then a single imclip would suffice.

Of course, you should very carefully use `testfitsimage` to find exactly the borders of the overcan regions and the image regions which correspond to each amplifier before running any of this.

Verifying What You've Done

After performing an overscan correction on a set of images, you should inspect these images to make sure that the overscan correction worked correctly. Use display and imexamine to do this (see Section 1.6). Look for the following things:

- The overscan region itself should be zero. (Note that this won't be true if you used Deeplib to overscan correct a subset of the image that didn't include the

overscan region, or if `datasec` in IRAF's `ccdproc` didn't include the overscan region.) If you use "m" in imexamine to get the statistics of the overscan region, it should average to zero. The average should be well within σ of 0, where σ is the standard deviation reported by imexamine. (Ideally, it should be within 1–2 times $\sigma/\sqrt{N}$ of 0, where N is the number of points for which "m" gives you statistics.)

- The sky level should have gone down by the amount of the overscan. Look at a given region of background pixels before and after the overscan; make sure that the average value has gone down by the right amount. On a multiple-amplifier image, verify this for every amplifier. (It is very easy to leave out one or more amplifiers, or overcorrect one of the amplifiers, on such an image if you are not careful when constructing your list of `overscan` commands to run.)

- If you included any zero images in the set of images to which you applied overscan correction, the image data in the zero images should also be close to zero. (This will not necessarily be the case with dark images.)

- Make sure that you haven't introduced any new structure to the image. New bright or dark lines along the rows, or horizontal "ripples," which weren't present prior to the overscan correction, are some possible indicators that there was a problem with your overscan correction.

If you've run `overscan` on a large number of images, generally you only need to perform these verifications on a few of them (two or three). If it worked on those, chances are it worked on all of the images. Only the extremely careful would look at every single image after the overscan step to verify that they all worked properly. If you do only choose two or three to look at, don't just choose the first two or three; select them randomly from the list of images to which you applied the correction.

Trim Correction

The trim correction is very simple: once you're done with them, you cut off calibration and "garbage" pixels, so that the image only contains useful data. Sometimes, you will want to cut off more than just the overscan region. For some telescopes (including the LRIS imager on the Keck telescope), the field of view of the instrument and telescope is actually smaller than the area covered by the CCD. In this case, there will be some obscured or vignetted pixels on the edge of the field which contain no useful information. Sometimes, for one reason or another, there is a bright stripe at the edge of an image. When you are trimming an image, it's usually best to remove this sort of garbage in addition to the overscan region. You may use `testfitswidget` to figure out what the best trim region is. The Deeplib task `imclip` will actually cut out the images.

There are only two cases in which you must be careful when performing trim corrections:

1. You must apply exactly the same trim correction to every image of a set. In particular, you must apply the same trim correction to the images that will be used to build a flatfield (see Chapter 3) as you do to each image to which you will apply that flatfield. Otherwise, the flatfield will not be properly aligned with the other images. While your reduction procedure may appear to be successful, your final images will not be properly reduced, and will have much higher noise than is optimum.

2. If the overscan region is in the middle of the image (as is the case in Figure 2.3), you must be careful to trim out *all of* and *only* the overscan region from the middle of the image. Columns of pixels which were adjacent on the physical CCD that obtained the data must be adjacent in your final, reduced image, or your ability to measure relative positions of objects will be compromised, and photometry on objects near the "seam" will be incorrect.

2.4 Gain Multiplication

At this point, you should mulitple all images by the gain of the image. Previous versions of these instructions had you doing this at the end. However, because different amplifiers will have different gains in multiple-chip images, it helps to reduce that difference as much as possible before proceeding with the reduction.

The "gain" of the image, as defined here, is the number of photoelectrons per "count" in the image. "Counts" are the raw pixel values in the image as you have them right now. Photoelectrons are what was actually measured. Typical astronomical images have gains in the range 1–10. You can usually find the gain somewhere in the header— but be careful that you're finding a e-/count (or e-/ADU, ADU being "Analog to Digital Units", which is just counts) ratio, and not a count/e- ratio!

The step of multiplying an image by the gain is not considered a part of standard data reduction by most data reduction procedures. You will see this listed as something you must do if you find another text or set of instructions for reducing data. What's more, some people (probably including even many in the SCP) will consider it loony that we multiply all of our images by the gain. Truth to be told, that we have historically done this *does* create some rather severe problems. However, there are two very convincing reasons to do it. Photoeletrons are what you have really measured. As such, statistical noise properties and so forth should make more sense if you consider data in units of photoelectrons, and the numbers in the image are more directly related to something physically meaningful. The primary reason to do

this, however, is that a lot of the Deepsearch software assumes all images loaded into the deepsearch database are gain multiplied... so you gotta do it. This condition may be relaxed in the future (and indeed it will have to be to solve some problems we currently have), but it will take Rob a lot of work before it may be relaxed.

The Deeplib task `mulconst` is what you need to perform gain multiplcation; it's operation is obvious. (Run "`mulconst --help`" to see the calling sequence.) For multiple amplifier images, be very careful to use the right gain for the right region of the image, and to limit each `mulconst` command to the proper region of the image. (Similarly, for multiple-chip cameras, make sure you use the right gain for the right amplifier of the right chip. CTIO's Mosaic camera, for instance, has 16 different gains for its 8 chips.)

Sometimes, the gain keywords in the header are wrong. E.g., the VLT has the wrong chip numbers associated with what seem to be right gains. Tread with caution, and make sure you're doing the right thing. With a mosaic stack (e..g WIYN MiniMos ro CTIO Mosaic), you can use the IRAF "mscfindgain" task on a pair of raw domeflats and a pair of zero images to measure the gain for all of the chips.

Generally, with multiple amplifier chips (such as the VLT), in a long-exposure image you will clearly see the different regions or quandrants of the chip; the sky will be brighter in one quadrant, dimmer in another. After gain multiplication, those differences should mostly go away (although they will probably still be visible; flat-fielding will take care of the rest). If the differences do not substantially reduce, or if they get worse, then chances are that you used the wrong gain for one or more regions of the image.

2.5 Knitting Images Back Together

Data which all comes from one chip generally should be saved as a single contiguous image. Sometimes, at this point, you will have separate FITS images for individual amplifers. This will be the case for an image like that in Figure reffig:fourampimage after the trim step, and is the case for some image stacks such as CTIO Mosaic and WIYN MiniMos. After the gain multiplication, you should knit the images back together. You do this with the Deeplib command `knitimages`; run "`knitimages --help`" for documentation. Use of this command should be obvious. Make sure you knit the images together in the right order! A good thing to check is to look at some of the images, and look for an object that straddles the "seam". Make sure that it does not look discontinuous. (There may be a slight flux discontinuity due to residual gain differences, which won't be corrected until the flatfielding step. However, there should be no morphological discontinuities.)

One of the trickiest things here is keeping track of your filenames before and after

knitting. For instance, with the Mosaic camera, before knitting for each exposure you have 16 separate files files `filename_1.fits` through `filename_16.fits`. After knitting, if you use a zero-offset convention, you will only have `filename_0.fits` through `filename_7.fits`. You have to be *very* careful to do this in an order such that you don't overwrite an individual amplifer image while it is still needed; you should also be careful in cleaning up leftover images which are no longer useful, if you are in fact deleting intermediate stage images.

You may find it safer to use a different naming convention for images post-knitting. For instance, with the Mosaic camera, you could name them `filename_a.fits` through `filename_h.fits`. This will make it very clear to you which images are and aren't knit. (This convention also has the advantage of being similar to a convention you will use for filenames when loading the images into the database.)

2.6 Zero and Dark Subtraction

The purpose of Zero (or Bias) subtraction is to subtract any pixel-to-pixel structure in the bias of an image. Overscan correction (Section 2.3 takes care of the gross bias of the image. For most modern CCDs, the noise due to zero correction (σ_z from Section 1.4.2) is so small compared to other sources of noise that the zero correction is usually relatively unimportant. (The WIYN MiniMos camera seems to have ~ 30 counts left over in its zero images.) However, it's also relatively easy, so usually you do it.

The first task is to create a "combined zero" image. You may do this by making a list of your zero calibration frames (explicitly taken as such by the observer at the telescope) and running that list through Deeplib's `pclipflat`. It is very important that you use the parameter "`SCALEBY none`" when using `pclipflat` to create a Zero frame! A sample command line might look like:

```
% pclipflat zeros.lis Zero.fts -sc none -ls 3 -hs 3
```

Here, "`zeros.lis`" is a text file which lists, one per line, the files to be combined into output zero image Zero.fts. *These files should be the files which have already been overscan corrected, trimmed, and gain multiplied.* There must be no spaces, even at the end of the line or the end of the file, in this list file! That is the most common source of "image not found" errors which seem to make no sense whatsoever.

If you have multiple chips, the command line would look like:

```
% pclipflat zeros.lis Zero_@chip@.fts -ch a b c d e f g h \
   -ls 3 -hs 3
```

Here, after the -ch parameter are the parts of the filename that are different from one chip to the next. These might be numbers rather than letters, depending on the data files you have and what you did at the knitting stage if any. One output file will be written for each chip, with the `@chip@` in the output filename replaced with the appropriate string in the list after the `-ch` parameter. The file `zeros.lis` must list the filenames, also with `@chip@` in place of the "chip" part of the filename, e.g.:

```
image001_@chip@.fits
image002_@chip@.fits
image003_@chip@.fits
...
```

In this case, `image001_a.fits` is the first zero image for the first chip, `image001_b.fits` is the first zero image for the second chip, etc.

The `-p`, `-ls`, and `-hs` parameters control the "pclipping" algorithm used in the combination. This is disucssed at greater length in flatfields chapter, Chapter 3. Choosing the right values for these paramters requires understanding the algorithms. However, in most cases, when you are combining 7 or more images, the values listed on this command line are right for a zero image. The `-sc none` parameter tells `pclipflat` *not* to scale the images before combining them. Scaling is necessary for flatfields, but must not be done for zero images.

A zero image should usually look pretty boring, and they should all more or less look the same. If your combined zero looks a lot different from most of the individual zeros that went into it, you have problems.

Once you've created a master zero image for one night's worth of data, you subtract it from every other image taken that night. Subtract the zero image from the other images which have already been overscan corrected, trimmed, and gain multiplied. The Deeplib program `imarith` is a quick and simple way to subtract images. On multi-chip cameras, make sure to subtract the right zero image from each image! `imarith` does not have any sort of "`@chip@`" support, and must be run separately for each individual chip of each individual image.

CCDs also have what is called "dark current." This is the rate at which charge (basically, false signal) accumulates in the pixel even when no photons are falling on it. For most modern CCDs used on the ground (with the notable exception of that on the NEAT telescope), dark current is so much lower than sky background that it is generally ignored. (ROB, WRITE MORE ABOUT DARK SUBTRACTION?)

2.6.1 Evaluating Zero and Dark Correction

Once you have performed zero and/or dark correction, you should evaluate your images to make sure that everything worked properly. The first step is to just look at some (or all) of the corrected images to make sure that you introduced no gross artifacts (see Section 1.6). Next, you should make sure that you have not adversely affected the sky noise of your image (see Sections 1.6 and 1.8.1). The sky noise should either be approximately the same (if, as is the case with most ground based CCDs, the zero correction is insignificant), or lower (if, as is the case with NEAT, dark correction is an important step of the data reduction).

Chapter 3

Flatfields

3.1 General Introduction

Flatfielding is often the final sten in standard data reduction. Sometimes you will need also to do fringe correction (see Chapter 4). For the Deepsearch, there is a whole bunch of additional foo-faa (Chapter 5) you must do once the images are fully cleaned (reduced).

3.1.1 Why Flatfield

Because each pixel of a CCD camera is not precisely like its neighbor, the light sensitivity of each pixel will differ. This variation in the pixel response is typically on the order of a few percent, but can be as high as ten or so percent. This can have a significant effect on the analysis of the image. This pixel-to-pixel variation results from the not quite perfect manufacturing process, tiny dust grains and other imperfections. Astronomers remove this effect by creating a "flat" image of a uniformly lit field, and dividing it into the object image.

Because the pixels may have slightly differing color sensitivity, you have to build a separate flatfield for each filter. If you took images during the night through the V, R, and I bands, then you will need three flatfields, one for each of the filters.

3.1.2 Types of Flats

There are several methods of creating a "flat" field. Which one you use, depends on your application, including what kind of photometry you are interested and so forth.

Dome Flats

Many astronomers image a brightly lit screen inside the telescope dome. Typically used are bright tungsten lamps, often placed at the end of the telescope, behind the secondary mirror. Sometimes lamps on the dome floor or along the wall are used to illuminate the screen. Since the light source is so close, the telescope is completely out of focus but the light from the dome follows the same optical path as focused starlight. The result is that the ccd is wholly illuminated (flat).

Often one can control the brightness of the lamps by means of a rheostat so that exposure times are short, about 1 or 2 to 15 or so seconds, for a light level roughly "half welldepth", or half the level at which the detector saturates. This means that for a given filter, the time integration is kept constant. A series of 10 or so dome flats per filter are taken in the afternoon by the now-awake observer, usually before dinner.

One problem with dome flats is that they are redder than the night sky, since the lights used are tungsten lamps. Thus, many observers also use twilight flats.

Twilight Flats

Twilight flats are images of the twilight sky. These are typically obtained in the narrow time window between sunset and and the end of twilight, though they can also be taken just before sunrise (but this is when the observer is practically comatose after a long night's observing!). The telescope is pointed about 15 degrees from the zenith, toward the east (opposite the sun's position). A series of exposures through each filter is taken, just like with dome flats. However, since the sky's brightness is not constant (it's getting darker), one has to continuously increase the exposure time to get the same level of illumination on the CCD. While one tries for a light level close to half welldepth, frequently you settle for levels as dim as half this bright, or as bright as 50% brighter. In addition, between exposures it is wise to dither the telescope. Dithering requires moving the telescope by several arcseconds, and is done to avoid putting the same star on one pixel. Details are covered below in building a flat. Twilight flats, like the dome flat, are also not the same color as the night sky.

Sky Flats

"Super flats" are where the object images themselves are used to create a sky flat. Object images tend to have long exposures, and during these long exposures they will accumulate a substantial number of counts from the sky background. These counts may be used as flatfields in the same manner as twilight flats. They have

the advantage that you are directly measuring your flatfield image from what is contributing most of the light to the images; as such, if you have enough images to make a good superflat, often this will provide the best possible flatfield correction.

Which form of flatfield you use depends on the data in question. Sometimes there aren't any twilight flats, or there aren't good enough twilight flats, and you must settle for domeflats. Sometimes there are different gradients in the night-sky images (e.g. due to a nearby moon), making construction of a superflat difficult or impossible. If you are able to make a good superflat, that is generally the best sort of flatfield to use. Sometimes you use two different sorts of flatfields on the same image, as discussed in Section 3.5.2 below.

3.2 Building a Flat

Generic Flat Recipe

The first step toward building a flat is to perform any overscan, trim, zero subtraction, and dark subtraction steps (see previous chapters) on each flatfield image. (Note that in the case of super flats, even your target images count as flatfield images!) The next step is to combine the individual flats into one final flatfield image– one for each filter. The simplest way to do this is to take the median of all the images; in practice, usually you use a more sophisticated procedure (read on). The result should be a smooth looking, "flat" image. In theory, the average count value across the CCD would be constant. In practice there may be a gradient across the chip, caused by uneven illumination (possible for both dome and twilight flats) either due to lamp placement (dome flats), or to instrumental effects, unwanted objects like stars and cosmic ray hits. So in practice the final flat is made by medianing after pixel rejection.

Now you see why the telescope could be dithered between exposures... if you don't, a star in your twilight flat will fall on the same pixels in multiple images, meaning either that you have to rejejct those pixels from a lot of images, or (worse) that you will not have enough images where those pixels are "clean" to perform a satisfactory rejection.

3.2.1 Bad Pixel/Object Rejection

Median

The easy way to get rid of bad objects (e.g. unwanted stars, cosmic rays) is to take the median of a number of images. Since the median is the mid-value point of a

sequence of numbers, the highest and lowest values will be rejected. The remaining value will be the median.

[picture of a bunch of ccd pixels with a line joining the same pixel along each ccd]

When there are many images or pixels or numbers (> 10 or so), a simple median may be sufficient, at least for dome flats. However, due to observing constraints, the number of flat fields available is often small, more like 7 to 10 per filter, so additional cuts are used prior to taking the actual median so that a very high value or very low pixel value don't skew the resulting median. Even with superflats, where you may have a large number, it is advantageous to use a more sophisiticated algorithm to avoid biasing your median. Examples of bad things you don't want to include are: cosmic ray hits, hot pixels, stars, bad columns. This is why one dithers the telescope between flatfield exposures: so that if there is a star on a given pixel in one image, it won't be on that pixel in all the other images, thus allowing it to be rejected with a median.

Some of the algorithms used are pclipping, minmax rejection, avsigclip. In IRAF, these are selected in "flatcombine" or "combine" procedures. The primary algorithm, and the one implemented by the Deeplib routine `pclipflat` (which is what is recommended for buildling flatfields), is pclipping.

3.2.2 P-Clip Rejection

Reject pixels using a sigma based on percentiles. This algorithm looks at the distribution of the values in one pixel across all images being combined, and from that distribution tries to determine a "noise" (σ) level, and then rejects all images whose value in that pixel is more than a certain number times σ from the median.

Figure 3.1 demonstrates the pclip algorithm. All of the values of a given pixel on the images to be combined are listed, and sorted. A median (or mode) values is chosen as a "center" value. (The median value, by definition, will be at the center of this list, while the mode may not be.) The pclip parameter tells the algorithm how far to count away from the center value to define the image which is "one σ" away from the center. For example, if the pclip parameter is -0.5, then the algorithm will count halfway from the center image to the image with the lowest pixel value (from image 10 to image 6 if there are 19 images). The difference between the pixel value of that image and the center image is defined as 1-σ. A negative pclip parameter means that the algorithm counts down from the center; a positive pclip parameter tells the algorithm to count up. You almost always want to use a negative pclip parameter. The primary reason for outlyers which you want to reject to avoid biasing your "average" combined value is that some images will have objects (stars, galaxies, cosmic rays) in some images, which will be brighter than the center value. Most of

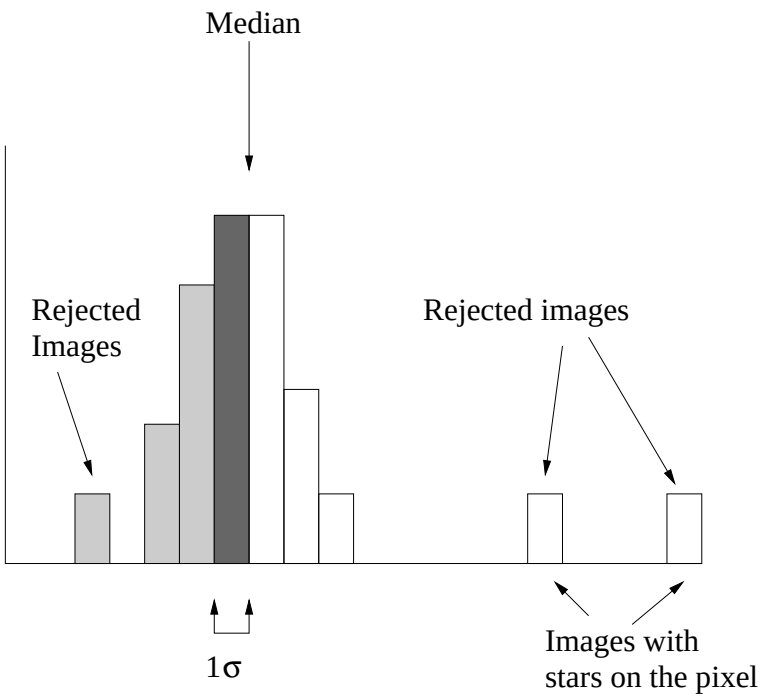

Figure 3.1: A demonstration of a pclip algorithm. The plot is a histogram of pixel values. The shaded pixels are those whose values are below the median. The darker shaded values are those who represent the 45% (corresponding to the pclip parameter of -0.45) of the pixels below the median which closest to the median. The value σ is defined by the spread of these darker shaded pixels. Any images whose values are more than 3 times this defined σ away from the median are rejected.

the values lower than the center value are lower simply because of the natural spread of the flux distribution. Thus, measuring a "noise" of the flux distribution will be less biased if you measure it off of the low side of the distribution than the high side of the distribution.

Once the σ value is defined, all images whose value in that pixel is more than a certain cutoff (e.g. 2 or 3) times the value of sigma from the central value are rejected. Those images who are not rejected have their pixel values averaged, and the result of that average gives the value of the flatfield in this pixel.

This entire procedure is repeated for every pixel of the image.

Note that all images must be appropriately *scaled* before this algorithm will work. Consider two domeflat images, one of which was taken with lamps twice as bright as the other. In general, most of the pixels in the first image will have a value twice as

bright as the pixels in the second image— but you don't want to reject them simply for that reason! Both flatfield images may be good and useful flats. Before the pixels of different images can be compared, the images as a whole have to be scaled to a common normalization. This is generally done internally in flatfielding routines. (In Deeplib's `pclipflat`, the `-sc` (or "`SCALEBY`") parameter specifcies the method of scaling.)

3.2.3 Preemptive Object Rejection

Although in principle, the pclip procedure should be sufficient to prevent stars in some flatfield images from biasing your combined flatfield, sometimes it helps to give the pclip a head start. In particular, if you have a relatively small number of images (less than, say, 21), or if the images are correlated (i.e. taken of the same field of sky with only small telescope dithers), it is often necessary to perform some sort of object rejection. Before the images are sent to the pclip algorithm, an object finder is run on them. This object finder locates stars and galaxies, and masks out the appropriate pixels on the image where they are found. The pclip algorithm will then hopefully have an easier time getting rid of any remaining outlyers.

3.2.4 Making a Flat

The Deeplib command to make a flat is `pclipflat`. A typical invocation might look like:

```
% pclipflat flatsR.lis FlatR.fts -im 2 5 -t tmp \
   -p -0.3 -ls 3 -hs 3
```

Note that for this command to work, the directory `tmp` must exist as a subdirectory of the current directory. Create it with "`mkdir tmp`". The file `flats.lis` is a text file which lists the images you intend to combine into a flatfield; this file looks exactly like the file you used for creating a Zero image (Section 2.6). The output flat image will be written to `FlatR.fts`. Here, a pclip parameter of -0.3 is used, along with cuts of $3-\sigma$ both below (`-ls`) and above (`-hs`) the center value (which is by default the median). The `-im` parameter tells the program to build an "isomask", which is a mask of objects located and rejected as discussed in Section 3.2.3; 2 and 5 are good standard parameters to use with `-im`. The `tmp` directory is only necessary when you use the `-im` parameter. In general, you will want to use `-im` for twilight and super flats, but will not need it for dome flats (which tend not to show much in the way of images of stars).

You may find that you get better results by changing the `-p`, `-ls`, and `-hs` parameters. Bear in mind what they mean. If you have a very small number of images, then values of `-p` too close to zero become troublesome. (E.g., if you are only combining 7 images, Then only 4 will be at or below the median. If you set `-p` to -0.3, then the difference between the central pixel value and the one below it will define your "1-σ". If those values happen to be very close together, σ will be very small, and most of the rest of the images will get rejected.)

If you have a multiple chip camera, the command might look like:

```
% pclipflat flatsR.lis FlatR_@chip@.fts \
   -ch a b c d e f g h \
   -im 2 5 -t tmp -p -0.3 -ls 3 -hs 3
```

The `-ch` parameter, and the use of `@chip@` (both in the output flatfield filename and in the filenames in `flats.lis`) are discussed above in the section on creating a Zero image (Section 2.6)

There are a whole host of additional parameters which may be given to `pclipflat`. Run "`pclipflat --help`" to see what they are. Some are worthy of mention. You may want to use `-sb` to improve the background subtraction used internally in the creation of an isomask. After you have been through this procedure a few times, you will understand what a "surface box size" is (Section 5.1). If you have a very large number of images (say 21 or more), you may wish to use "`-pf mode`". Here, the central pixel value in the pclip algorithm is defined by the mode of the distribution rather than the median. For too few values, the mode of the distribution becomes ill-defined and difficult to calculate meaningfully; for that reason, by default the central pclip pixel is normally defined by the median of the values.

Since sometimes combining flatfields can be a computationally intensive process, it is possible to use multiple CPUs at the same time with the `parpclipflat` command. Use of `parpclipflat` is beyond the scope of this manual, but if you know how to use MPI and `mpirun`, that's what you use with `parpclipflat`. In so doing, you will need to know that the first (zeroth) process is a mere driver process that will use very little CPU, so if you are on a two-cpu machine, you might actually want to specify `-np 3` with mpirun; if you explicitly specify the machines to use with `-p4pg`, remember to take this into account. Be aware when using multiple machines that eventually the speed of NFS will limit you, and adding more CPUs will only further burden the poor NFS server hosting the disk without improving your running time. If this paragraph made no sense to you, then ignore it, and deal with how long `pclipflat` can take.

Separate Flats for Separate Filters

You need to make a **different flat for each filter**. If a data set includes images taken through the V, R, and I filters, you need to make three different flats. Only use the R flatfield to correct R data, and so forth. Anything else will cause big problems.

Normally, you will make a different flatfield for each night of data. If you are reducing data from a three-night run, you will make one flatfield for the first night, another for the second night, and a third for the third night. If, however, you only have a handful of images from some of the nights, it is OK (if not optimum) to either use a flat from another nearby night (on the same telescope, of course), or to make a combined flat from individual flatfield images collected from different nights.

How Many Flats is Enough?

The general rule is: more is better. However, it may take too much processing time to combine too many flats. For dome flats, frequently 7 is plenty, and anything more than 15 individual flats is overkill. For twilight flats, use all that you can get your hands on for a given night. Twilight usually doesn't last long enough for the observer to obtain more flats than is necessary.

For sky (super) flats, if possible use every image in the appropriate filter to make your combined flatfield. If that is excessive, then using just 15 is barely adequate; 21 is usually adequate; more is better. Especially if you are using a smaller number of target (or sky) frames to make a superflat, you will have better results if you deliberately omit frames that have extremely bright stars or very large galaxies.

Selecting Images From Which To Build a Flatfield

For dome and twilight flats, usually you will select every dome or twilight image of the appropriate filter, and combine them all into a flatfield.

For super (sky) flats, you want to make sure that you are only including the "right" sky images. In particular, do *not* include dome images when building a superflat; also, be sure not to include bias images, focus images, or test images. Also, standard star images usually have a very short exposure time, and will not have enough sky signal to make them a useful contributor to the flatfield. Indeed, short exposure time images may be actively detrimental, in that they will skew the tests used for rejecting objects in the flatfield. For sky flats, it is usually best to include every frame of the right filter, which is not somehow screwed up, that is of the night sky and has a "long" exposure time (usually, greater than something like 20 or 40 seconds). Use of `hselect` is a good way to get a quick list of the images and their exposure time and filters.

3.3 Evaluating a Flat

How flat is flat? In principle, the flatfield image should be flat, that is, all the pixels would have the same values and clean, without any features. One way to check the quality of your flat is to look at it (see Section 1.6). Look along both the columns and rows, the pixels should have about the same value with a few percent variation. (If you used Deeplib's `pclipflat` to create the flatfield, they should all be close to 1, although in a multiple-chip camera the average value may vary a little from chip to chip.) In practice, though, you're likely to see a gradient (a full discussion is saved for later). Another good check on your flatfield is to flatten one of your individual (preprocessed) night sky images or twilight flat images with the resulting Flat (see Section 3.4). The resulting flattened image should be free of dust features, and be flat. Verify this by looking at the image with `testfitswidget`.

If the flattened twilight flat looks odd... extra features that weren't there in the unflattened image, for example, check the individual flats since sometimes one of them is really not useful. If the twilight flat was taken toward the end of twilight, as the sky got darker, there could be several stars in the image, or the average count level is lower than others, or the telescope wasn't dithered between exposures which would also show up as extra features. After you've eliminated the "odd" flat images, rerun flatcombine and generate a new Flat.

In particular, if making a twilight or a sky flat, look for residual stars in the flatfield, which will show up as very dim, small, smudges. Ideally, the rejection and medianing procedures eliminate stars, but if you don't have enough images, they may leave behind little signals. It is possible that they are real; try making a domeflat in addition to the other flat. If the smudges are real features, they should show up on the domeflat as well. One quick test is to divide a twilight or superflat by a domeflat (using `imarith`). While frequently you may see a broad but shallow gradient in this divided image, you should not see much in the way of localized features or smudges.

The first thing to do in order to try to get rid of these, especially in the case of sky flats, is to try again using more individual flatfields and rebuilding the flatfield. For instance, you may conclude that it's impossible to get a decent flatfield from one night of the run, and combine a flatfield from two or three nights together. If that doesn't help, try fiddling with your rejection parameters. If that still doesn't work, then, woe is you, for you must refer to section 3.5, "Advanced and Iterative Flatfield Creation."

NOTE TO ROB: Write about checking the sigma of flat.

Sometimes, especially on nights with moon, and especially when making those prized sky flats, each image will have a slightly *different* bulk gradient across the image. This will tend to completely screw up any median or outlyer rejection procedure

you used, and make it seemingly impossible to make a good flatfield. If this is the case, you must refer to Section 3.5, "Advanced and Iterative Flatfield Creation."

Once you're happy with the Flat, it's time to apply it to your object images.

3.4 Applying Flats to Your Object Images

With Deeplib, it is as simple as using `imarith` to divide your object by your combined flatfield. Make sure that your flatfield is normalized to 1 before you do this! (By default, the output from `pclipflat` will be.) For multiple chip cameras, all of the chips should *together* be normalized to 1, but each chip may not be; again, if you use `pclipflat`, things will "just work". Be sure to apply the flatfield of a given filter *only* to images of that filter... and conversely that each image gets flatfielded, using the right flatfield. Also, with multiple-chip cameras, make sure to apply the flatfield for the right chip to each image.

3.4.1 Evaluating How Good a Job the Flatfielding Did

Once you've applied a flat to all of your target images, you should evaluate the images to make sure that the flatfield procedure did what it was supposed to. A few things to look for:

- Make sure that no new features were introduced to the images when dividing by the flatfield. If new features (bumps, wiggles, "holes" (bumps down), and generally junky-looking things) were added, there is probably something wrong with your flatfield.
- Artifacts such as "dust donuts" should be gone from your sky images.
- Measure the noise of the sky background (for instance, by dragging out a box on a small (e.g. $\sim 25 \times 25$) area of blank sky with the left mouse button in `testfitswidget`). The noise (standard deviation of sky pixels) should be measurably lower after you apply the flatfield, and certainly should not be any higher; the point of flatfielding is to eliminate the noise σ_f from the image (see Section 1.4.2). If everything has gone perfectly, if you had the right gain when you did gain multiplication, if the read noise is insignificant, and nothing else is awry, then the standard deviation measured on a region of blank sky should be very close to equal to the square root of the mean flux of the sky in that region. (This would not be true if you had not applied gain multiplication.)

3.5 Advanced and Iterative Flatfield Creation

3.5.1 Why Do It?

Sometimes, the procedure outlined above for building the flatfield will fail. The primary reason is that when medianing a lot of images together, you are implicitly assuming that the images all have the same continuum (background) shape. Because of the optics of the telescope and the gain profile of the CCD, a constant (flat) sky or defocused dome image may be artificially measured to be brighter on one part of the image than another part of the image; flatfield correction will ideally remove these differences. Since all of the images have the same background shape, simply scaling them all to the same average level should mean that, except for brighter pixels like stars and cosmic rays, all the images look basically the same.

Sometimes, however, there is a different varying background from one image to the next. The usual reason for this is if there is a bright moon. The sky may be brighter nearer the moon than it is further from the moon. Since all images in one night won't be taken in the same place, a gradient in the background due to the moon may be different from one image to the next. This will cause problems when trying to reject stars by medianing images.

INSERT FIGURE SHOWING 1D CASE OF THIS PROBLEM

Most of the time, hopefully, the simple flatfield procedure outlined above will be sufficient for making super flats (sky flats). You can tell that you need to do something more sophisticated if you see any of the following things:

- There are features, perhaps looking like residual stars, in your flatfield image which can't be eliminated by combining more sky images into your superflat. (See Section 3.3.)
- Look at several of the sky (or target) images of the same filter that are going into your super flat. Is the gradient in the sky background different between these images? For instance, is it brighter in the upper left corner in one image, but brighter in the lower right corner in another image? Sometimes you will have to do something more careful (i.e. plot, or divide two images) than just looking at the images.

3.5.2 Two-Stage Flatfields

If you are able to build a domeflat for the night's data, you can sometimes solve the problem by using both a domeflat and a specially constructed superflat. The

procedure for this would be, generally:

1. Do preliminary data reduction on all images, dome and sky and miscellaneous (Chapter 2).

2. Make a combined domeflat (Section 3.2).

3. Flatfield all images with the combined domeflat (Section 3.4.)

4. Select the flatfielded (with a domeflat) images you will combine to create a secondary superflat. Copy those images into a work subdirectory. (You will corrupt these images during the following steps.)

5. Subtract a low-order surface from each sky images that will go into the superflat. You want to subtract just the slope of the image, not its mean value. You can do this with the Deeplib task `surface` (See section 1.7), using the command line:

   % `surface` *filename* `-s` *outfilename* `-sz`

   That will subtract a first order surface (a plane) centered about zero from *filename*, and write the resultant (subtracted) image into *outfilename*. The input and output filenames may be the same, in which case the input file will be overwritten. (Not a big deal, since you are in a work directory.)

   Do `surface --help` to see the flags you can specify to tune object-rejection parameters and to do higher order fits to the background. To decide how high an order you need, run `surface` with the `-surf` flag instead of the `-s` flag, to write out the surface that would be subtracted. Compare this to the image from which you will be subtracting the surface, using IRAF's display and imexamine (Section 1.6). Also, after you have subtracted the surface, look at the image to make sure it is flat. Try this with a number of different orders on a sample of images from the data set.

   ROB, MAKE A FIGURE

   You may also wish to try using Deeplib's `edgesurface` instead of `surface`. (More discussion of the differences should be inserted here.)

6. Perform a `pclipflat` procedure on the surface-leveled sky images to create a secondary superflat.

7. Back in your main directory, divide all of your images by this surface-leveled sky. You do this to the images to which you have *already* applied the domeflat. Note that IRAF's `ccdproc` will probably balk at doing this, because it things the images are already happily flatfielded. You may use the Deeplib program `imarith` to divide images by the flatfield (assuming it is properly normalized, which it should be if you used `pclipflat` properly).

8. Evaluate how well it worked (see Section 3.4.1). In particular, make sure that the sky noise is *no higher* after applying the secondary the secondary flat than it was before applying the secondary flat.

3.5.3 Iterative Flatfields

Sometimes, a two-stage flatfield isn't possible; either you don't have individual domeflats from which to make a combined domeflat, or for another reason domeflats or unsatisfactory. Or, perhaps, you decide that even two-step flatfielding procedures were insufficient. In this case, you need to iteratively create a superflat. The outline of the procedure is that you make a first cut at a superflat. In a work directory, you apply this superflat to all of the individual images that went into the superflat. Use these to fit a gradient in the background to each image, and subtract that gradient from the *original* (pre-flatfielded) image. After that subtraction, combine again to make a second cut at a superflat. Repeat the procedure until you have a good superflat, or until the superflat does not change from one iteration to the next.

ROB, WRITE THIS OUT IN MORE DETAIL

Chapter 4

Fringe Correction

4.1 Introduction

Sometimes, after flatfielding, I-band (and even occasionally R-band) images will show "fringing." This is an interference pattern of sky photons within the thin wafer of a thinned CCD. Although you might expect that the continuum of the sky would wash out the fringes, there are enough narrow emission lines in the I-band sky that fringes are visible. These fringes need to be subtracted from the image. They represent a very high-order variable background which is extremely difficult to deal with using traditional background subtraction methods.

Sometimes, flatfielding with a superflat removes most of the fringing. This is because the fringing pattern is also visible in the fringing. Purists will object that this is actually not a perfect correction, because the fringing does not represent gain variations between pixels, but rather an additive pattern on top of the sky continuum due to the sky emission lines. As such, while dividing by a superflat may take out the fringing, in terms of what needs to be done for objects, you are dividing by too *much* where the fringes are high, and by too *little* where the fringes are low. In practice, the differences are small enough that we do not worry about this (we just accept that as an additional part of $\sigma_?$; see Section 1.4.2); additionally, it flattens out the sky, removing much of the adverse affect of the fringing on aperture photometry. If you want to be a purist, then you will use a twilight or domeflat for flatfielding, and then perform a fringe subtraction as described in this chapter. After that, you may wish to go back to the previous chapter and iterate one more time on a superflat, in case there are any residual gradients due to a different broad gradient in the domeflat and the night-sky images.

Even if a superflat removes most of the fringe pattern, sometimes there is enough left over that it's still worth performing the fringe correction procedure in this chapter.

The fringe correction procedure basically involves first creating a map of the fringes. Then, for each image, you must determine the proper amplitude of that map to subtract from the image.

4.2 Making a Fringe Map

Ideally, a fringe map should have just the pattern of the fringes, and no other structure (such as stars or galaxies, flatfield pixel-to-pixel variations, or broad gradients). The best way to make a fringe map is to make both a dome flat (or, possibly, a twilight flat) and a super flat (section 3.1.2). Because the superflat represents a night sky signal, it will include the emission lines that cause the fringing, and so will show the fringing pattern. The domeflat, which is normally an image of broadband lights on the dome surface, show very little or none of the fringing pattern. By dividing the superflat by the domeflat, you are effectively flatfielding the superflat. This should remove any pixel-to-pixel variations. After this, subtract the smooth background (a single "sky" value or a low order surface using `surface` or (best of all) `edgesurface` with a relatively large box size (200 or so)) from the fringe map, so that you are left with just the fringes varying around a zero mean.

If you do not have the calibration frames necessary to make a fringe map (i.e. both super and dome flats), you can use a fringe map made for the same telescope and camera from a previous run. This is not ideal, but usually good enough. However, you must make very sure that exactly the same trim correction (section 2.3) was applied to the fringe map you are using and to the data set you are reducing. You do have to be very careful to make sure that the fringe map from the previous run and your fringe map used exactly the same trim region (Section 2.3)!

4.2.1 Dividing Super and Dome Flats

This may be done very simply using the IRAF task `imarith` or the Deeplib program also called `imarith`. Alternatively, you can do this in IDL using `readfits` a couple of times, followed by a division and a `writefits`.

Only the pattern of the fringes is important; the exact amplitude is not. The amplitude will tend to vary from image to image anyway, so that will be fit later when removing fringes from each image. However, it is convenient for the rest of this procedure if the amplitude of the fringes in the fringe frame is in the right ballpark. The super flat, combined from all of your target images, will have an effective exposure time, and hence an effective background and fringe amplitude, which is about right. After dividing the super flat by the dome flat, multiply the resultant image by the average (or median) value of the dome flat (or just by 20,000, which is almost always

in the right ballpark). The Deeplib routine `mulconst` can accomplish this:

```
% mulconst FringeI.fts 20000 FringeI.fts
```

4.2.2 Subtracting the Low-Order Background

You want the fringe map to have the fringes oscillating about zero. Ideally, at this point, all you should have to do is take the mean value of your fringe frame and subtract it. However, the super flat and dome flat may have slightly different broad gradients across the image. (This is one reason a super flat is usually preferred.) Thus, it is better to fit a low order "sky" to the fringe map and subtract that.

One cleaner way to subtract a low order background is to fit a plane, or another low-order (2nd, 3rd, etc.) surface with the Deeplib program `surface`. This routine will calculate an outlyer-rejected mean for every (by default) 100×100 region on the image. It will then fit a low order surface to those averaged points, and subtract that surface from the input image. Type "`surface --help`" for information on using this program. In most cases, your use of this program will look like:

```
% surface rawfringe.fits -o 3 -s finalfringe.fits
```

where `rawfringe.fits` is the name of the fringe file produced in the previous section, the number after `-o` is the order of your fit, and `finafringe.fits` is the fringe map with the low-order background subtracted.

Alternatively you may use the Deeplib routine `edgesurface`, e.g.:

```
% edgesurface rawfringe.fits deleteme.fits -so finalfringe.fits
-bx 200
```

Be sure to use a box size small enough to capture the shape of the background, but big enough that it doesn't start subtracting out the actual fringe features. The file `deleteme.fits` will have the background subtracted; you can look at it to see if it shows too much evidence of the fringe pattern itself.

When all is said and done, use your favorite method for looking at images to make sure that the fringe map looks reasonable. It should have a mean or median of (or very close to) zero, there shouldn't be any broad low order gradients, and it should generally look the same as any fringing which may be visible in any of the I-band images.

4.2.3 Smoothing a Fringe Map

You don't want to introduce additional noise in your fringe map. Ideally, the Super and Dome flats used to create the fringe map are of high enough signal-to-noise that you won't. However, you may wish to smooth the fringe map, for instance by using a median filter, to reduce the pixel-to-pixel variations in the fringe map. You may also wish to clip out bright spikes and anti-spikes.

To clip out bright spikes and anti-spikes, use the Deeplib procedure `levelextremepix`; run

```
% levelextremepix --help
```

for more information. Choose your `LOW` and `HIGH` pixel values to be comfortable outside the range of the fluctuations of the *real* fringes, but not too much more than that. After this, run Deeplib's `medfilt` to smooth the image. Choose the box size such that it's smaller than the typical width of a fringe; `medfilt`'s default box size of 5 is often a good choice.

4.3 Applying the Fringe Map

You apply a fringe map by subtracting a fraction of it from each flatfielded image that requires fringe correction (usually, all I-band images). In other words:

$$\mathbf{i}' = \mathbf{i} - a \times \mathbf{f}$$

where $\mathbf{i}$ is the flatfielded image, $\mathbf{f}$ is the fringe map you made in section 4.2, and a is an amplitude. The game is to figure out the best value for a. One laborious way of doing this is to try lots of values of a (using, for instance, Deeplib programs `mulconst` and `imarith`), looking at the image each time and decided which one minimizes the fringe pattern by eye. The surgeon general has warned that doing this for more than a couple of images can cause permanent insanity.

A better way is to use a program that will try a range of values of a, measuring the noise of the sky background in $\mathbf{i}'$ for each value. This program should then find the value of a which minimizes that sky background. The Deeplib program `fringecor` will do just this. It starts by trying a range of a (by default, 0–2), finding the sky noise in $\mathbf{i}'$ for something like 10 values of a in this range. It plots the sky noise as a function of a, fits a parabola, and zooms in around the minimum of the parabola. It repeats this process until the values of a which go into the parabolic fit are at least as close together as a user specified "minimum step" (by default 0.01). In order to determine the sky noise, `fringecor` first uses a pclip algorithm (see section 3.2.1 to

make a mask which will block out all the objects (stars, etc.) in image **i**. (I suggest using "lowsig" and "highsig" limits of 8., and a "dilate" value of 5. You can get the program to write out a copy of the mask it uses, if you want to make sure your masking values make sense.) It calculates the sky noise of unmasked pixels about a local average sky value (which is by default the average sky value within a 200×200 pixel box). This method should be robust to slow gradients in image **i**.

For information on the calling sequence of `fringecor`, issue the command `fringecor --help`. The `PLOT` (or `-pl`) keyword is particularly useful when you are first running `fringecor` on a batch of data, to make sure that you have chosen a good initial range of a and that things are running correctly. Once you have the parameters set up, `fringecor` is slow enough that you will probably want to set up a batch job to fringe correct an entire run in one go.

4.4 Evaluating the Fringe Correction

You evaluate the fringe correction in pretty much the same way as you evaluate anything else. Look at the image to make sure that the fringe pattern has indeed decreased or disappeared, and to make sure that no artifacts were imposed on the image by the fringe correction procedure. Use your favorite image statistics routine (see, for example, Section 1.8.1) to make sure that the sky noise does in fact go down with the fringes, and that the median and/or average value of the image as not overly changed.

Sometimes, you may want to try two different fringe maps (e.g. if you can't make one for the current run, but you have two fringe maps on record). If it is not visually apparent which one did a better job, you can look at the last plot produced by `fringecor PLOT` to figure out what minimum sky noise value `fringecor` found for that fringe map. The fringe map that produces a lower minimum sky noise value will usually be the better one.

There will be times when the fringe correction makes the images *worse*. Currently, the `fringecor` program determines what it thinks is the "best" scaling a of the fringes by figuring out what value of a minimizes the sky noise. This may not actually be the best way to do it, but I'm not sure what is. Especially when the `fringecor` program comes up with a *negative* value of a, be suspicious. Look at the "fringe-corrected" images, and make sure the fringes didn't get worse. If they did, either you will have to laboriously determine a manually (trying lots of values and looking at how they affect the fringe pattern), and then subtract the fringes manually using the "-a" argument of `fringecor`. Alternatively, if the fringes are weak enough in images flattened with a superflat, you may be able to skip the fringe correction step.

Chapter 5

Loading Images into the Deepsearch Database

In order to use your now nicely reduced images with much of the rest of the Deep/IDL software (as well as, eventually, the Deeplib software), you will need to load the image into the Deepsearch database. The image itself must be placed in one of the directories where the software knows to look for images "in" the database, and a program (the IDL routine `ltelescope`) must be run in order to add the header information about the image into the database proper.

5.1 Dotting the t's and Crossing the i's

There are a handful of things which need to be done before loading the image into the database.

Gain Multiplication

By convention, images in the Deepsearch database are scaled such that 1 ADU (or 1 dn) represents 1 photoelectron. If you did not multiply the images by their gain in section 2.4, then you must do so now. *Do not multiply by the gain twice!* That would be just as bad as not doing it at all.

Determine Surfacing Parameters

Once upon a time, all the images loaded into the Deepsearch database were "surfaced," i.e. sky subtracted. This was generally done using the Deep/IDL routine `edgsurface`. There were a number of problems with this, so the surfacing requirement has been relaxed. Nonetheless, there are still some routines that require surfaced images. The procedure now is to surface them on the fly. So that this may be as painless as possible, you should determine the parameters of `edgsurface` which people surfacing on the fly will want to use. Note that you will *not* actually keep the surfaced images, or load them into the database, you will just figure out which parameters you *would* have used.

You may use the Deeplib prodcedure `edgesurface` to edgesurface an image; run "`edgesurface --help`" for more information. You may also use IDL, where you do the following:

```
IDL> im1=readfits("filename",hdr)
IDL> ims1={ims_struct}
IDL> ims1.nx=n_elements(im1[*,0])
IDL> ims1.ny=n_elements(im1[0,*])
IDL> sky,im1,sky,noise
IDL> ims1.sky=sky
IDL> ims1.exp_sigma=noise
IDL> skyim=edgsurface2(ims1,im1,boxsize=boxsize, diff=diff, stepsize=stepsize,
oversamp=oversamp, dilate=dilate)
IDL> subim=im1-skyim
```

The variable `im1` holds the image you are surfacing. `skyim` has the sky image determined by `edgsurface`, and `subim` is the difference (i.e. the surfaced image). You should look at `skyim` and `subim` to make sure that the surfacing is going as planned. Ideally, you want to be able to remove the sky background at all points on the image, even if there is a gradient or curvature to that background. However, you do *not* want to oversubtract the sky underneath real objects. (Sometimes, this is impossible for very bright stars.) If you see spots on `skyim` which correspond to the positions of stars and galaxies on `im1`, then you know that you are oversubtracting some objects. It is nearly impossible to do a background subtraction which is good enough over the whole image which doesn't also oversubtract some of the brightest objects. (This trade-off is the main reason we have decided that it is no longer a good idea to load surfaced images into the database.)

The game is to figure out the parameters *boxsize*, *diff*, *stepsize*, *oversamp*, and *dilate*. Those parameters will be stored in the database entry for each image, allowing somebody to quickly surface the image without themselves having to spend the time to find the parameters. Normally, one set of parameters will work for an entire run of

data, so you figure out the parameters by testing them on a handful of images, and then use those parameters when loading the whole run into the database.

The meaning and typical values of these parameters is as follows:

boxsize The surfacing routine averages all pixels within a given box. *boxsize* is the size of a side of this box in pixels (although the program does some scary things to make an even number of boxes fit on the image). Typical values are 100-500 pixels. Smaller box sizes correspond to higher order fits, and will have trouble with subtracting brighter objects. Larger box sizes correspond to lower order fits, and may not be able to fully subtract a background with a curved gradient.

diff I don't even really know what this is. 2 is a default value, so maybe it's a good one. I *think* that it has something to do with how many sigma (in terms of ims[1].exp_sigma) above the local mean a point must be in order to be recognized as an outlying object (star, cosmic ray, whatever), but I could be wrong. You can feel free to play with this to see if you can improve the results of your background subtraction.

stepsize I don't know what this is either, and I haven't played with it, so I have no intuition for it. The default value is 3. Use that unless you have a better idea.

oversamp Another great mystery of the universe... but I *think* that boxes are (somehow) separated on the image by 1/oversamp box sizes... or something like that. Play with it, and figure out what it does. The default value is 5; I've sometimes used 2.

dilate When an object gets recognized as significantly above the background, the masking routine "dilates" the mask about that object to include adjacent pixels. This is so that if only the center pixel of a star has high enough S/N to be recognized as a bright object, the whole star may still be masked. This parameter indicates the radius of dilation. It defaults to 1 (which is either no dilation or a 1-pixel dilation, I'm not sure which). You can try playing with values up through 5 or 10, or even more if you think your images warrant it.

5.2 Rotating Images

We used to require a certain orientation in Deepsearch images. That requirement has been relaxed; however, you still need to *figure out* the orientation (rotation and flip) of the image so that you can set a field in the Deepsearch database properly!

The best way to figure out the right rotation is to look at the image and compare it to some other image of the sky whose rotation you know. This might be from the Palomar Sky Survey, obtained from a place such as SkyView:

`http://skyview.gsfc.nasa.gov/cgi-bin/skvadvanced.pl`

Alternatively, you can use the IDL routines `getapmcatalog` and `makeapmimage` to make a fake image of the sky based on the USNO star catalog.

If you simply cannot match the image at any rotation to the sky, it's possible that the RA and Dec in the header are wrong, either because it's a multiple chip camera and you don't understand where the fiducial point of the coordinates is, or because the RA and Dec are simply... wrong.

5.3 Checking RA and Dec in the header

You want the "RA", "DEC", and "EQUINOX' (or "EPOCH") keywords in the the FITS header to correspond to the center of the image. Frequently, they do not. There are a variety of reasons. Sometimes, as with the WIYN MiniMos, it is because the EQUINOX keyword in the header is just wrong. Sometimes telescopes aren't pointing exactly where the header thinks they are. Frequently with multiple-chip camears, every chip's header will have RA and DEC keywords indicating the center of the whole array, not of that chip.

You must fix the RA and Dec in the header of the images so that they are correct for the center of each image.

5.4 Writing 16-bit Images

Unless you've coadded a whole bunch of images, despite using 32-bit floating point numbers for the file format you usually only have 16 real bits of information within each image. In order to save disk space, it is worth writing the images out in a 16-bit format. FITS images have a standard way of scaling floating point images into short integers. In this scaling scheme, all pixels below a minimum value, and all pixels above a maximum value, will be clipped to the minimum and maximum values respectively. Naively, you might choose the minimum and maximum pixel values in your image as the clipping parameters. However, consider the case where most of your images has pixel values between 1000 and 4000, but you have one pixel (because of a cosmic ray, or a low value on the flatfield) whose value is 10^{12}. Choosing 10^{12} as your clipping maximum will cause all of the dynamic range in the region you care about (1000–4000) to be lost. You'd rather screw up the 10^{12} pixel than the pixels with good information about sky, galaxies, stars, and supernovae.

The best way to choose the maximum clipping value for an image is to keep track of

the "welldepth" of the image. The "raw welldepth" is the maximum pixel value which would have been returned from the telescope. For many cameras, the raw welldepth is 32768 or 65536. However, some detectors will have a lower raw welldepth. (In those cases, the pixels will saturate before they reach the limit of counting for 16-bit integers.) You can often figure out the raw welldepth by looking at a raw image from the telescope, finding some saturated stars, and looking at the values of the saturated pixels.

The raw welldepth will be modified by all of the operations you apply. Ideally, it won't be much affected by flatfielding, but it will be affected by overscan correction (the bias level being subtracted from the effective welldepth). It will also be affected by gain multiplication! If you coadd images before loading them into the database, the welldepth of the added image will be the sum of the welldepths of the images that went into the coadd. If you successfully keep track of all of this, you can choose your clipping maximum to be equal to the welldepth (or something like 1.05 times the welldepth) of your reduced image.

There are two ways to choose the minimum clipping parameter. The first, assuming you've done no sky subtraction (which you shouldn't have), is to just use 0. A better way is to look at your images and find what looks like the minimum sky value. (You could do this in an automated fashion using the Deeplib program `surface` to fit a low-order surface, and then read the minimum pixel value from that surface.) You can use the Deep/IDL routine `SKY` to find the sky noise in the image. Set your clipping minimum to the minimum sky value minus 10 or 15 times the sky noise. (Note that sometime the `SKY` routine screws up for short-exposure standard star images. If this is happening to you, figure out what the sky noise roughly is, and hard-code the minimum value to a negative value more or less "several" times that sky noise.)

Once you know your clipping maximum and clipping minimum, use the Deeplib routine `bscale` to produce a 16-bit version of your final reduced FITS images. The images which come out of `bscale` should be roughly half the size of the images that go into them. Look at these images, and use things like `imstat` and the Deep/IDL `SKY` on them, to make sure you didn't completely screw them up when you wrote the 16-bit versions.

5.5 Setting Header Parameters

Note: often you will find that writing 16-bit images, and adding random header keywords (section 5.5) is easily done at the same time using a "preload" IDL script which you write. Two examples may be found in ***`~rknop/idlpro`****: for single chip cameras,* ***`preload.pro`*** *is a script Rob has used in the past, and for the eight-chip CTIO Mosaic camera, there is* ***`preload2.pro`****.* ***Do not use these script without copying***

them to your space and modifying it! *They almost certainly do the wrong thing! (Note also that they currently rotate the images based on an old requirement of the Deepsearch database. You may choose to do so, or you may choose not to do so.) Rob finds it simpler just to do so at this point. There are arguments for both.*

There are a handful of header parameters which must be set right for the images to be correctly loaded into the Deepsearch database. All of these parameters should be set in the final images you intend to load into the database. If you are clever, you can keep track of them as you go. Otherwise you will have to do some work now to get them set right.

The easiest way to edit header parameters is to use IRAF's `hedit`. However, you may also get some mileage out of IDL's `readfits`, `integerfits`, `sxpar`, and `sxaddpar`. It's easy to do the wrong thing with IDL's FITS header routines, so be careful, and make sure you know what you're doing. (Hint: use the "format" keyword with sxaddpar.) If you do this with IDL, it is convenient to do the writing of 16-bit FITS files (with `integerfits`) at the same time.

5.5.1 detspec, dettag, numchips, *etc.*

These are **very important** to get set right. Look at

```
$DEEPHOME/calibration/det_spec.html
```

for more information on these fields and their values. Speak with Rob or somebody else who has a clue about them before doing anything rash.

5.5.2 Welldepth

This should be set to the welldepth determined as described in section 5.4.

5.5.3 RA and Dec

These keywords will almost certainly be in the header already. If you are reducing data from a single chip camera, then they are probably right. However, if you are reducing data from a multiple chip camera, it is probable that the RA and Dec will be of some reference point of the whole camera's footprint. This is discussed above.

Your best bet is to use methods similar to the method of finding the rotation (section 5.2). Also, try to find, either by asking somebody in the group or looking

at the telescope's web page, known offsets and sizes of the chips in a multiple chip camera. At that point, it's a matter of trial and error. It is especially painful if the RA and Dec in the header of a multiple-chip camera are *wrong*. In those cases, the telescope logs may be of help; otherwise, you may just have found out how you're going to tediously spend the next several hours.

Try for example the following IDL commands; they may work, or they may not.

```
IDL> r=freadimage2(ims1,im1,'./filename.fits')
IDL> print,sixty(ims1.ra),sixty(ims1.dec),ims1.epoch
Make sure that these are what the header says!
IDL> sky,im1,sky,sig
IDL> window,0
Or wset,0 if the window already exists.
IDL> frame,im1,zero=sky,span=10.*sig
IDL> dx=0
IDL> dy=0
IDL> apmim=makeapmimage(ra=ims1.ra+dx/60./15./cos(ims1.dec*!pi/180),$
   dec=ims1.dec+dy/60.,/xy,fielddiax=15.,fielddiay=15.,$
   nx=1000,ny=1000)
IDL> window,1
Or wset,1 if the window already exists.
IDL> frame,apmim,zero=0.,span=20000.
```

(Note that the $'s at the end of the lines are the character that IDL uses to indicate a line is continued. If you type the `makepamimage` command all on one massive line, omit the $'s.) Replace the actual size of the image in arcminutes (which you can probably find out from the observatory's web page, or from reading the source code to the Deep/IDL routine `ccdsize`) for `fielddiax` and `fielddiay`; they will be different if the chip is not square. The ratio of `nx` to `ny` should be the same as the ratio of `fielddiax` to `fielddiay`. Repeat the last five times, altering the value of `dx` and `dy`, until you see a visual match between the star pattern in the image and created from the USNO catalog by `makeapmimage`. When you have a match, `dx` and `dy` are the offsets for this image.

If you know where a given chip is relative to the center of the array, you might want to start with better values than 0 for `dx` and `dy`.

5.5.4 Others

I've almost certainly forgotten something.

5.6 Loading the Images

The Deep/IDL routine `ltelescope` is currently the *only* way to add images to the Deepsearch database. What it does is add information about the image into an SQL database, and into a (probably redundant) NetCDF file. The image itself is *not* actually added to the database proper, just information about the image.

To use `ltelescope`, build a "list file" that has a list of all the images you wish to load into the database. (It should go without saying that you should only do one run's worth of data at a time. Do not list images from multiple detectors (i.e. different values of `det_tag`) within one list file. Then issue the command:

```
IDL> ltelescope,list="listfile",det_tag=" telescope", $
       suffix="cln",optional parameters... , $
       set_keys=['SURFACED=0', $
              'SURFBOX=boxsize', $
              'SURFOVERSAMP=oversamp', $
              'SURFDILATE=dilate', $
              'SURFSTEP=stepsize', $
              'SURFDIFF=diff' $
              'ROTATE=0']
```

`det_tag` should be exactly the same as the values of `det_tag` you put into the image headers. The sundry parameters whose name start with "surf" are the surfacing parameters you found in section 5.1. Note that the "`SURFACED=0`" keyword indicates that you are loading *non-surfaced* images. You did realize during the surfacing above that you were just figuring out which parameters you *would* use, and that you weren't supposed to really surfaces images, yes?

If you did *not* rotate the images to the Deepsearch standard orientation, then `ROTATE` should not be 0, but rather the parameter you would pass to the IDL `rotate` command in order to rotate the image from its current orientation to the standard Deepsearch orientation (north down, east to the left).

The "optional parameters" may not be optional; which additional parameters you need to use depend very much on the specifics of the telescope for whose data you are adding. Some of the ones which you may need to use:

order Set `order='mdy'` for telescopes who store the date parameter in the header as month-day-year. Other options you may need to use are `'dmy'` and `'ymd'`. Use your good sense.

namerun Some particularly obnoxious telescopes don't put a "run number" into the header. This is just an ordinal number used to tell one image from the next. If there is nothing in the header, you can specify this keyword to indicate that `ltelescope` should should attempt to glean the run number from the filename of each image.

nostop Specify "/nostop" to keep `ltelescope` from warning you when key header parameters are missing. See section 5.6.1.

nooffset If you've manually adjusted the RA and Dec in the header of multiple-chip cameras (usually a good idea!), then specify "/nooffset" to make sure that the offsetting hacks in ltelescope don't get used.

Read on for what to do once you get `ltelescope` started. Whenever anything scary happens, refer to section 5.6.2.

5.6.1 Interactively Specified Parameters

Image Type

If you managed to get `ltelescope` started correctly, the first thing it will ask you is for the "image type." Usually, you will indicate 5 here for "Cleaned" images. The only other type you might use is 6, for "Com" images. Com images are images which have been cleaned, but which have previously been through some sort of lossy compression (such as hcompress).

Filter

`ltelescope` is deliberately too stupid to automatically recognize filters. It will give you an output indicating the text filter it has gleaned from the header of an image, and ask you to enter the number that corresponds to this text. Usually, this will be obvious. If you are loading a lot of images at a time, it will only ask you the first time it reaches any filter description.

It will also ask you to enter a filter description. Most of the time, you can just type the same thing that the program found itself.

Keyword was not set!

Frequently, `ltelescope` will stop warning you that one or more keywords weren't set. This is because either you didn't put something in the header that you should have, or because the telescope whose data you are loading has not been properly defined within the bowels of the Deepsearch software. If it is only the keywords BIAS and RONOISE that are missing, you can normally ignore this; of course, if you are a very good citizen, you will address the problem. If any other keywords are missing, you should probably figure out how to either get them in there, or get `ltelescope` know where to find the appropriate values.

There are two ways to keep `ltelescope` from stopping every bloody time it finds an image with header keywords missing. The first is to specify the "/nostop" option to the ltelescope call. You should only do this if you've restarted `ltelescope` after a first trial run, and know that nothing crucial is missing. The other is to issue the IDL commands

```
IDL> nostop=1
IDL> .c
```

during one of these stops. It should not stop thereafter.

File Being Renamed...

At this point, `ltelescope` will almost certainly rename your file to conform to the Deepsearch standard. You should make very sure that the filename is what it ought to be. In particular, make sure that `ltelescope` has parsed the date correctly. If it hasn't, you may need to use another keyword to `ltelescope` such as "order". Also, make sure that the run number being set is the one you'd expect. For most telescopes, this should be exactly the run number of the image. For the BTC, it should be the run number times ten, plus the chip (1 through 4).

Once you are convinced that `ltelescope` knows what it's doing while renaming images, you can answer "all" to the question about whether it's OK to rename the file; thereafter, `ltelescope` will not bother you further.

If there is a problem with the filename, you should answer "n" to the question about whether it's OK to rename the file. Then use the IDL commands `retall` and `close,/all`, fix whatever the problem is (e.g. by adding keywords to `ltelescope`), and restart `ltelescope`.

5.6.2 In Case of Trouble

Whenever you have trouble with ltelescope, it should be all right to do CTRL-C to quit the program. (You may have to type a character and hit return after that to get it really to quit.) Thereafter, as always when doing CTRL-C in IDL, issue the commands `retall` and `close,/all`. You can then try a modified command line, and restart `ltelescope`. Hopefully, from the output of the program, it will be clear to you when `ltelescope` is actually renaming files and loading things into the database. Up until that point, it hasn't done anything irreversible.

5.6.3 Watch for Errors!!

Watch ltelescope carefully to make sure everything gets loaded which you think is getting loaded. At the end, run `tracker`, with parameters to list the files you've just loaded in. Count the files (presumably using code rather than just counting lines on the screen by hand), and make sure as many got loaded as you wanted to load. If something didn't work, try to figure out why and fix it. Be careful! It's easy to let files fall through the cracks if you don't pay enough attention. Ideally, everything is automated and just works. In practice, it's always something new that goes wrong; there are *so* many things that could be screwed up about individual astronomical images and image headers, that there's no way the software could anticipate all of them.

5.7 Removing Images from the Database

There is currently no safe and documented way to do this. So, please be careful before loading something in. Rob is able to remove things from the database, but the tools he has are not user friendly enough that he's willing to release them to something else. So, if some images must be deleted, talk to Rob.

5.8 Moving the Images

Now that you have loaded the information about the images into the Deepsearch database, you must move the actual images to a directory where our software will know to find them. The images must be somewhere on the "deep image path," which is a list of directories set in the environment variable `DEEPIMAGEPATH`. Pick a disk on the deep image path with enough free space to accept all the images you have just loaded into the database. Move the images to the proper directory on that disk.

If you are not on the master site (i.e. on the Deepsearch Suns and PCs at LBNL), then the images will have been sent to the master site by the `ltelescope` procedure. If you find you have to reload something into the database, and you are not on the master site, it could be complicated. Talk to Rob, who will probably develop a way to handle this after being asked to do something by hand a couple of times.

5.9 Cleaning Up After Yourself

Delete unnecessary files. Note that until you're really sure that everybody's happy with the images, you might want to keep flatfield and other calibration files around. That will make it much easier both to redo things, and to diagnose any problems in the reduction. You may want to use the trick of giving the flatfield images names that make it clear that they are flatfields for this particular night of data. (Give it a name that looks something like the standard Deepsearch naming convention, with enough other obvious stuff like "flat" or "super.") Then, copy them into the deep image path. They will be automatically backed up by Ivan's and Rob's image backup system the next time anybody bothers to catch up with those backups.

(**NOTE:** If you are not on the master site, then you must *explicitly* copy anything you wish to save to the Deepsearch PCs at LBNL. Just moving them to $DEEPIMAGEPATH at your site won't do much good.)

For example: for the data from the BTC starting on the evening of 1999-April-10, I copied the following files to $DEEPIMAGEPATH (where n is an integer between 1 and 4, representing the 4 chips of the BTC):

Original File	File in DEEPIMAGEPATH
`Iflat_`n`.fits`	`apr111999btcIdome_`n`.fts`
`Rflat_`n`.fits`	`apr111999btcRdome_`n`.fts`
`Isuper_`n`.fits`	`apr111999btcIsuper_`n`.fts`
`Rsuper_`n`.fits`	`apr111999btcRsuper_`n`.fts`
`Zero`n`.fits`	`apr111999btcZero_`n`.fts`