IRAF Paso a Paso

Forewords

This is a brief, introductory course on IRAF, written by a IRAF user, rather than by an IRAF programmer. I have tried to clarify those points which may present problems to first-time users. The course is focused on the IRAF structure and language, and won't provide any info on how to process and analyze astronomical images and data.

I suggest that you try the various examples presented here, experiment freely with similar IRAF tasks, different combinations of parameters, on a variety of image formats and types, etc.

These pages might also be used as a quick reference, even if they are not exhaustive or complete at all.

This manual is in constant evolution, as existing topics may be revised or new contents added. I will be happy to receive comments, critics and suggestions on how to improve this presentation. Also, if you would like me to add (or expand on) a specific subject, just send me an email: nicola.caon@iac.es

This course is found in:
http://www.iac.es/sieinvens/SINFIN/CursoIraf/Intro.php

Note: This presentation has been formatted using Cascading Style Sheets (CSS). If you are using a non standard-compliant browser, you may notice inconsistencies or errors in the rendering of these pages.

Table of Contents

Last updated: 11-July-2006 by Nicola Caon (nicola.caon@iac.es)

IRAF Paso a Paso: 1 - Basic notions

A quick ID of IRAF

What does IRAF stand for?	Image Reduction and Analysis Facility.
Who wrote it?	People at the National Optical Astronomy Observatories (NOAO).
Why did it become so popular?	It was selected, among other available softwares, by STScI as the one on which to layer all the software to reduce data from the Hubble Space Telescope (STSDAS).
Other reduction packages based on IRAF?	STSDAS (Space Telescope Science Data Analysis System) TABLES and many other external packages (gemini, rvsao, xdimsum, etc.).
Who supports it?	The NOAO staff (for IRAF), the STSDAS/TABLES group at STScI, myself (IAC), etc....
What's IRAF's programming language?	It is called SPP = Subset Preprocessor Language, sort of a mix between FORTRAN and C. Also, IRAF tasks themselves can be used as subroutines to write CL scripts.

The current version of Iraf, as of July 2006, is v2.12.2a.

Getting started happily

What you should have:

1. The proper definitions at UNIX level (let's assume you do).

2. A directory from which to start IRAF; **always use this directory**, never start from another one! Usually it is: "***/home/username /iraf/***".
3. A file called ***login.cl***, and a file called ***loginuser.cl***. Usually, the first file contains general definitions and settings, while the second file is the appropriate place where to put personal settings and options, as it is not overwritten by **mkiraf**.

Note: the ***login.cl*** file is created by **mkiraf**; ***loginuser.cl*** is not, but can be copied from another user, or just use this loginuser.cl template and modify it according to your needs.
IRAF is then started typing **ecl** or **ncl** (see next chapter). Use a ***xgterm*** terminal (just type **xgterm** from any terminal), which provides the most suitable graphical display and interface. Some graphical tools, and line command editing as well, may not work or produce unwanted side-effects in a ***xterm*** or other terminal type.

Do I need anything else?

IRAF requires an external image display program. You have the choice among the following:

SAOImage:	I like it for the scale options, and for some interactive cursor capabilities. It should be considered as obsolete, replaced by DS9.
Ximtool:	Useful for comparing different images, or blinking; very easy for printing images. It's developed and maintained by the IRAF Group. Unfortunately it only works in 8-bit depth.
SAOTNG:	Stands for "SAOimage, The New Generation". A more sophisticated and good looking display, with a comprehensive HTML-like help on-line. It's no longer supported, as it has been replaced by DS9.
DS9:	Deep Space 9? Successor of SAOTNG. DS9 most important feature is that it can work in 24-bit depth displays. It is also available for Windows.

Currently we only support DS9, as it is the only one to work in 24- bit depth.
To start the DS9, just type **ds9** in any terminal.

Warnings:

- **Never** use two or more display windows at the same time.
- It is also preferable to start the display first, and then IRAF.
  Avoid starting the display from within IRAF, ex. **do not do**:
  cl> !ds9 &
- It is possible to have the display in your workstation, and use IRAF remotely, or viceversa, setting the ***node*** variable. This was useful when somebody was logged on a slow machine, and was running Iraf remotely on a more powerful workstation; nowadays it's no longer an issue.

Question: do I need a graphic window too? No, it automatically pops-up as soon as you do some graphic stuff.

Know and customize your IRAF environment

Important if you want a pleasant trouble-free session.
cl> show | page
will show an almost endless list of variables.
Most are related to the directory tree of IRAF, and are relatively uninteresting. However you should become at least familiar with the variables that are listed in the table below.

Important/useful variables: (| means or)

Variable	Examples	Definition
terminal	***xgtermc*** | ***xtermjhs***	tells IRAF what type of terminal it's using
printer	***lw4*** | ***lw11***	tells IRAF what printer to use
stdplot	***lw4*** | ***eps*** | ***eps***	tells IRAF where to send plots for hardcopies
stdimage	***imt800*** | ***imt5***	sets the size of the image buffer (NOT the physical size of the display window)
stdgraph	***xgtermc*** | ***stdplot***	tells IRAF where to display graphics
imtype	***imh*** | ***hhh*** | ***fits***	sets the default image format
min_lenuserarea	***24000*** | ***120000***	sets the size of space available for the image header
imdir	***HDR$*** \| ***HDR$pixels***	defines the directory in which files .pix are stored
uparm	***home$uparm/***	defines the directory in which customized parameter files are stored
tmp	***/usr/tmp/***	defines the directory where to store temporary files
glbcolor	***pt=4,fr=9,al=4,tl=4,ax=4,tk=4,gr=4***	defines the colors for the elements of the graphic plot (more on XGterm colors)
clobber	***yes*** | ***no***	determines whether files will be overwritten or not
imclobber	***yes*** | ***no***	determines whether images will be overwritten or not
node	***cherne!***	tells IRAF where the image display is (trailing "!" is required). Rarely used.

How to set a variable to new value:

Command	Example	Description
set variable=*value*	**set stdimage=imt1600**	change value temporarily: if you exit the current package (typing bye), variable is set back to previous value.
reset variable=*value*	**reset stdimage=imt1600**	change value; will be kept during the whole session.

- Note: **set**, without argument, prints the whole environment list in the reverse order in which the definitions were made.
  If a variable has been redefined, both final and original definitions are shown.
- The logical name of a directory must be used with a trailing "$" sign. Ex:
  cl> show images ; dir images$; type images$images.cl
- The variable ***home*** in IRAF need not coincide with the analogous UNIX variable (indeed it is often ***/home/username/iraf/***)
- You may define your own shortcuts to long directory paths. Ex:
  cl> set WFPC2 = "/net/cherne/scratch/ncaon/ncaon1/WFPC2/"
  cl> cd WFPC2$
  Notice: directory paths must end with "/" sign.

You can put variable settings and definitions in your ***loginuser.cl*** file to load them automatically when you start IRAF.

Two comments:

- It is preferable to set ***imdir*** to a relative pathname, not an absolute one. I usually set it to the same directory where header files are, i.e. ***imdir = HDR$***, or, alternatively, ***imdir = "HDR$pixels/"*** (pixel files are stored in the subdirectory "pixels"). This avoids that changes in pathnames, or moving files to another directory, "disconnect" the header file from the pixel file. Check how it is defined in ***login.cl***, and change it if you wish.
  If you work with FITS files (as we recommend), you won't need bother with this variable.
- Be careful with variable ***tmp***. It may be set to ***/usr/tmp/***, which might not have room enough for temporary images created by a task. You may want to set it to your working directory, or to an area with abundant free space.

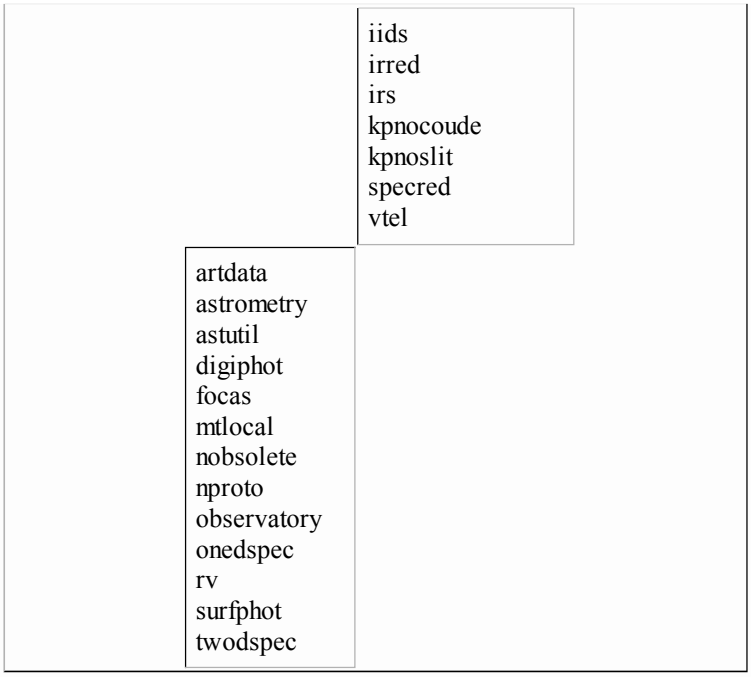
Sometimes, when you set a variable to a new value, Iraf still uses the value it has in memory and does not pick up the change. A couple of **cl> flpr** usually solve the problem.

A quick look at the IRAF structure

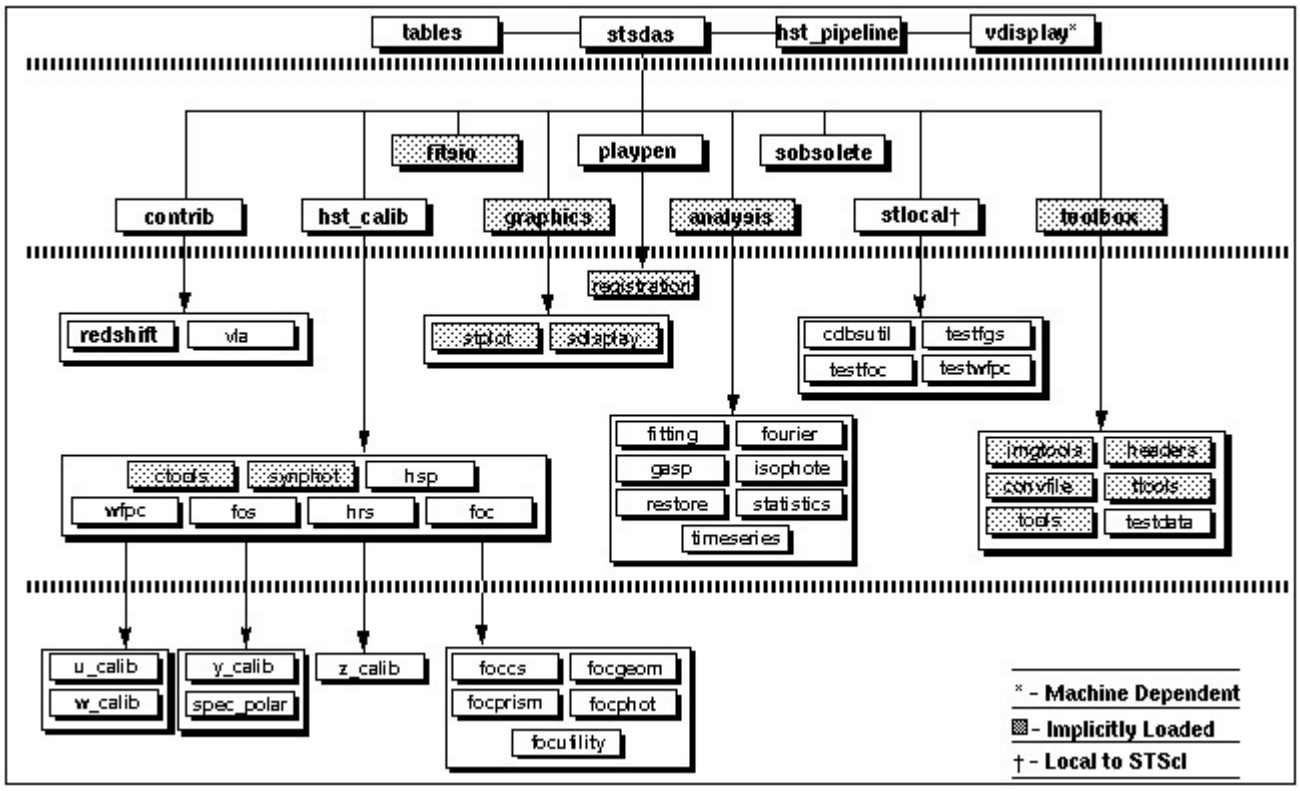
The basic element of IRAF is the **task**: command or program used to perform a specific function.
Tasks doing similar and/or logically related functions are grouped together into a **package**.
Also, packages can themselves be grouped into higher-level packages, to give IRAF an organized and logical structure.

An example of the hierarchical structure:

main package	sub packages	sub-sub packages	tasks
noao	imred	ccdred	badpiximage ccdgroups ccdhedit ccdinstrument ....
		argus bias ctioslit dtoi echelle generic hydra	



Another example: the structure of STSDAS (several versions ago; more recent versions may show many changes with respect to this figure).



Note that a task may belong to different packages; sometimes it has different default settings depending on the package it is called from.

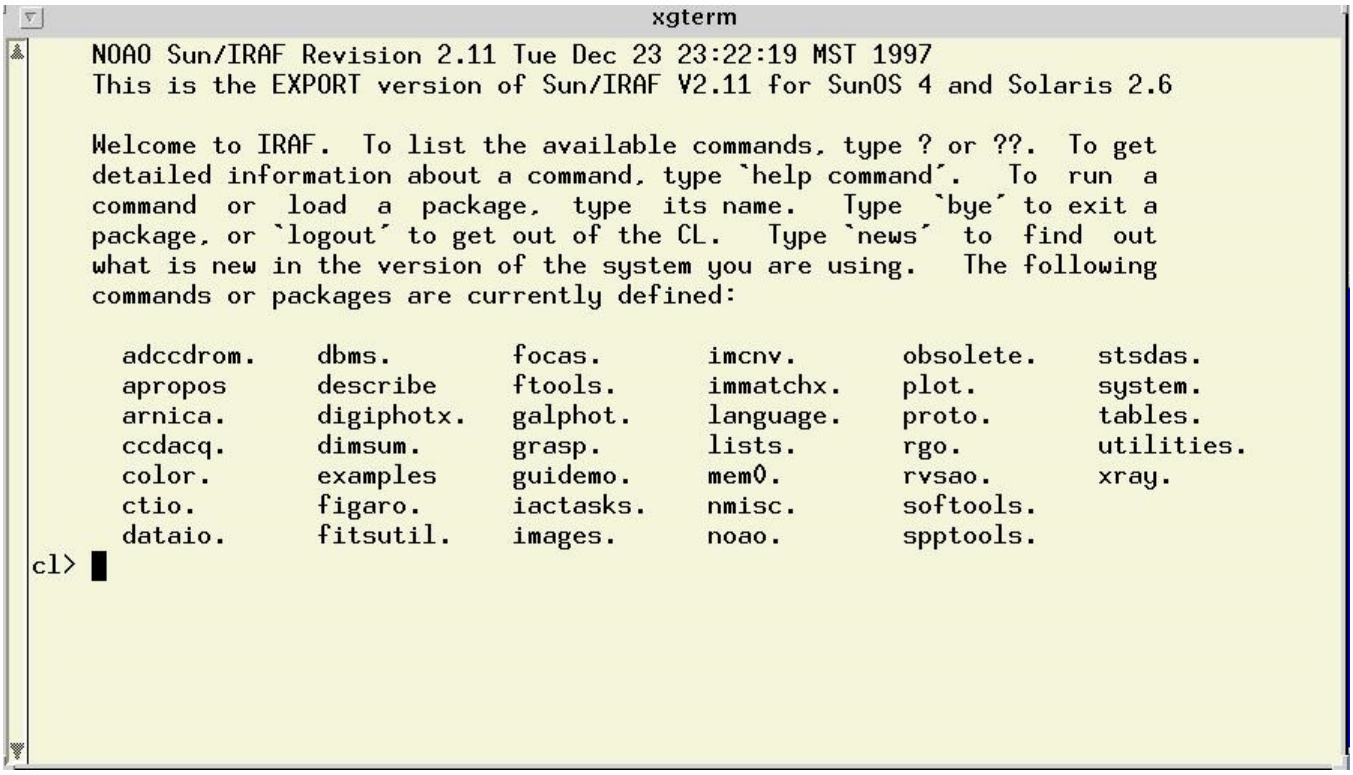
IRAF Paso a Paso: 2 - Packages and Tasks

Start: load and use packages

To start IRAF, type **ecl**.
This "Enhanced cl" provides a tcsh-style editing of the line command, a tab completion of file names, the possibility of moving through the history using the arrows, and a very much improved handling of errors in scripts and programs.
Remember that **ecl** must be started from the directory where you keep your ***login.cl*** file.

If you want to use **ecl**, and start Iraf from any directory, you should first define the environment variable ***UPARM*** to your uparm directory, for instance:
setenv UPARM /home/ncaon/iraf212/uparm/

The startup screen looks like the following:



The startup message contains useful hints, for instance the current Iraf version.
Names followed by a dot are packages (ex. adccdrom, arnica, etc.); names without dot are tasks (ex. apropos, describe, examples).
The prompt is the name of the last package loaded (in the startup screen, the prompt is "***ecl***").

Getting info on packages

You type:	What happens
name_of_package	The package is loaded; prompt changes to the package name.
?	List tasks in the most recently-loaded package.
??	List all tasks loaded, regardless of package.
package	List all packages loaded.
? images	List tasks or subpackages in package images.
bye	Exit the current package.

There are no limitations to the number of packages that can be loaded at any given time.

Getting help

The **help** command allows to get help on a package or on a task.
It has several options, described in the following table:

Getting help on known tasks or packages

Command	Action
help *task*	Show the whole help page for the task.
help *package*	Prints a one-line description of each task or subpackage in the package. If you omit the package name, it will apply to the currently loaded package.
phelp *task*	Like "help task", but allows to move forward and backword through the help file (= help | page).
example *task*	Show only the example section of the task.
describe *task*	Show only the description section.
help *task* opt=source	Show the actual code (useful if you wish to understand in detail how a task works).
help *package* opt=sysdoc	A general description of the package (available for a few, for instance **isophote**, **nebular**).
help *task* device=gui	Help is displayed in a nice GUI, which allows to navigate through the IRAF help docs (new in Iraf 2.12.2).
help *task* device=ps > task.ps **help *task* device=html > task.html**	Help is output in a postscript or an HTML file (new in Iraf 2.12.2)

The help GUI interface can also be launched by loading the **guiapps** package and then executing the **xhelp** task. This should be considered a prototype application for trial use, so it may still have some bugs or inconsistencies (such as hanging the xgterm in a RHEL platform.)

If you do not know what task does what you need, you can use:

Finding the task you need

Command	Action
apropos *<keyword>*	Look through a list of IRAF + STSDAS menus to find tasks matching the keyword. Package name is also shown in parenthesis. Notice that depending on when and how the apropos database was built, there might be packages and tasks not included in it.
refer *<keyword>*	Search the help database for tasks related to the keyword. An alternative to apropos.

Quiz:
Curious users want to know whether the task they're using is a real IRAF program (written in SPP) or just a CL script. What would you do?

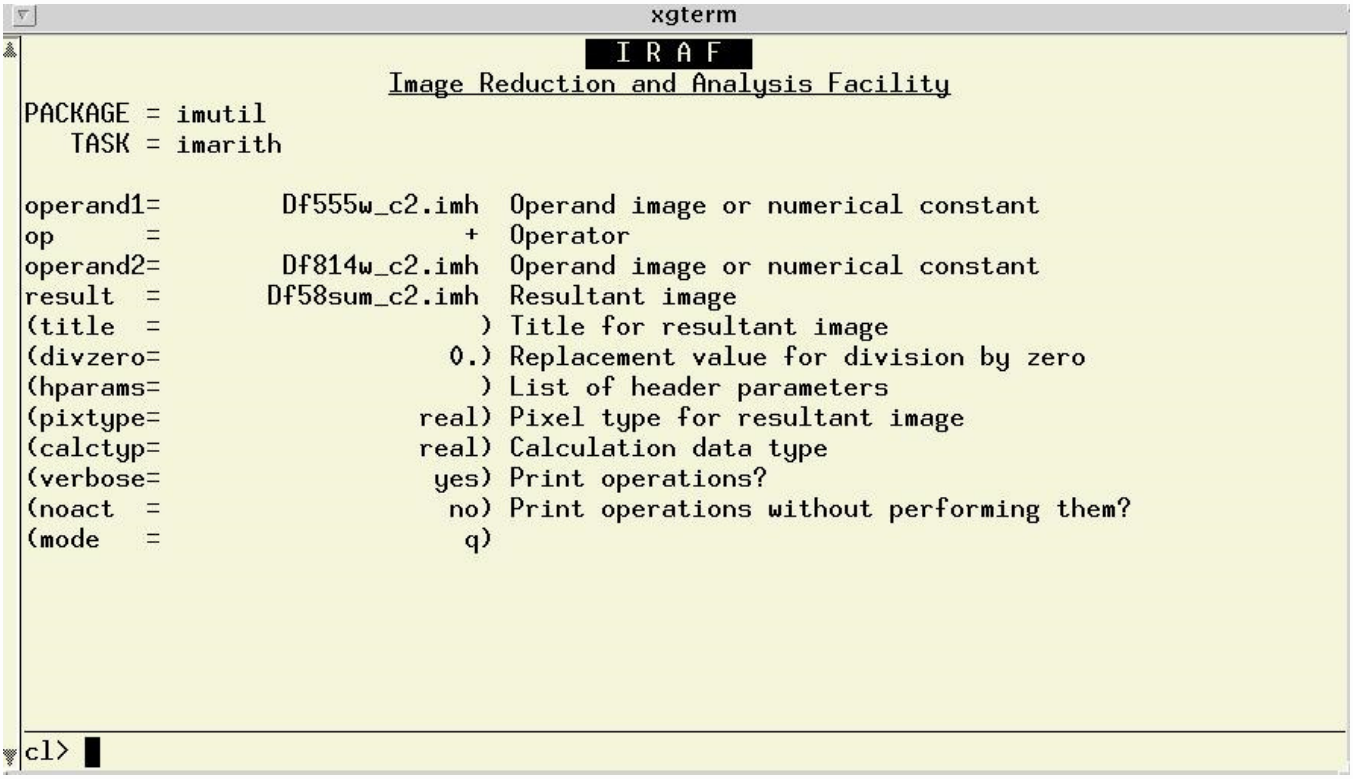
Anatomy of a task

The way a task operates is determined by its parameters.
There are two types of parameters:

required Must be specified to the task. If not, the task prompts for them.
They are always learned, i.e. become the default for the following run.
Must be given in the correct order in the command line.

hidden Do not need to be specified. Task will use default values. They might be learned or not according to how they are set.
If specified on command line, the syntax is `parameter=value`. May be put in any order, but must follow required parameters.

All parameters of a task can be displayed doing: **lpar task**:



Hidden parameters are the ones within parenthesis.
Alternatively, parameters may be listed with the command:

cl> dpar task
This is useful if they are to be written to a file.

Also, the command:
cl> =package.task.parameter
will show the value of that parameter.

Three ways to set or tell the task the parameter values:

Command	Description	Example
epar ***task***	Edit interactively parameter values. You move through the parameters menu and change them. Exit with **:q** to save, **:q!** to discard changes, **:go** to start task execution. Also, **:w** ***file*** will output the parameter list (in an unfriendly format) on file ***file.par***; **:r** ***file.par*** will read back a saved parameter file.	**epar imarith** ...editing... **:q**
task ***<list of parameters>*****i**	Parameters are specified in the command line	**imarith** ***image1 + image2 result*** **title="SUM" pixtype=real**
set explicitely	Syntax: **task.*parameter=value***	**imarith.pixtype=real** or **imarith.title="SUM"**

- Hidden parameters specified explicitely or with epar are learned; if they are set on the command line, they are not learned.
- To reset parameters to their default value, type **unlearn task** (useful if you get lost, expecially in tasks with many parameters). Also, **unlearn** is useful whenever a new release or patch has changed the parameter file of a task. A hint for this could be an error of the type:
  `ERROR: parameter 'orig_row' not found`
- If wrong type of parameter is specified in epar (ex. string instead of a number, or number outside the allowed range), IRAF will complain (ex: "Parameter value is out of range").
  Similar complaints are issued if wrong values are specified on the command line.
- When you exit from "epar", new changes are stored in a file in your "***uparm***" directory. Unlearn simply deletes this file. Whenever IRAF starts a task, it first looks for the parameter file in your directory; if not found, it uses the default values.
- Writing parameters to a file and reading them back is useful if you have several different combinations of parameter values, each applying to a specific data-set or purpose.
- Packages too may have their own parameter set. See for instance **ccdred**, **onedspec**, **echelle**. These are general parameters valid for all tasks within the package.

Though hidden parameters may seem less important than the required ones, actually they not always are. Before running any task, it is good practice to revise carefully all its parameters and change their settings to suit your needs. (As an example, in the task **fitcoords** the default value for xorder and yorder is 6. My experience with such task, applied to logn-slit spectra, is that such values produce wild oscillations in the fitted surface, and smaller values, typically 3 or 4, are much better).

Quiz:
Why do you think Iraf people defined and implemented these two different sets of parameters? What if they were all required? And what if they were all hidden?

Psets
Sometimes, the parameters of a particularly complex task (ex. **imexamine**), or which may be common to a set of tasks (as in the package **daophot**), are grouped into ***psets***.
To edit the parameters in a pset (ex. imexamine's pset ***rimexam***), do:
cl> epar rimexam or just type **cl> rimexam**

Psets can also be modified from inside the parameters of a task using them. For instance, doing **epar phot**, you can edit the ***centerpars*** pset by moving to the "centerp=" line and typing **:e**.

psets are identified in the package menu by a trailing"@" character. (Load the package **daophot** or **apphot** to see quite a few).

Expressions in parameters:
Simple numerical expressions, surrounded by parenthesis, can be given as the value of a parameter. For instance, we might do:
cl> display.z1=(200-3*60) ; display.z2=(200+3*60) or
cl> display dev$pix 1 zr- zs- z1=(200-3*60) z2=(200+3*60)

Executing tasks

There are several possibilities on how to run a task:

1. Type its name (or an unambiguos abbreviation) and enter parameters when prompted. Hidden parameters won't be queried, task will use the current ones.
   cl> imstat
   ```
   Images: dev$pix
   # IMAGE NPIX MEAN STDDEV MIN MAX dev$pix 262144 108.3 131.3 -1. 19936.
   ```
2. Specify parameters values on the command line when you run the task (required parameters are learned, and their default value is printed upon prompting; hidden ones are not).
   cl> imstat dev$pix fields="min,max" format-
3. Type **epar task**, change all the parameters you want, then type **:go**. Task starts immediately (maybe the simplest method).

Tasks can be executed in the background, tied together, or their output can be redirected.

1. **Background***:* A task may be executed in the background by ending the task command line with a &
   cl> imstat dev$pix form- &
   If not all required parameters are specified, CL stops and waits for response.
2. **Redirection***:* Output of a task can be redirected using symbols ">" and ">>" (as in Unix).
   cl> imstat dev$pix form- > imstat.txt
   Use "**>&**" to redirect both output and error messages.
3. **Pipe**: Output of a task may be used as input to another task (as in UNIX):
   cl> imhead dev$pix long+ | page
   cl> ?? | words | sort | table
   The output of imheader is fed as input to the task page; the output from ?? is broken into individual words, sorted, and printed in tabular format.
4. **Command line too long**: Commands can be extended to two or more lines using the "\" character. Line break can come at end of parameter string, or after a comma in a parameter list (no preceding space is then allowed).
   **cl> imstat dev$pix fields="mean,stddev,min,max,median" **
   >>> lower=0 upper=1000
   >>> means that IRAF is expecting completion of command line.
   If you try to edit a multi-line command, you generally get a messed-up output. A useful alternative is to write down the command in a file, and copy-and-paste it into Iraf. If you wish to modify it, do it so in the file, and repeat the copy-and-paste.
5. **Commands on same line***:* Two or more commands may be written on the same line, provided they are separated by a ";" (semicolon, exactly as in Unix). Ex:
   cl> time ; dir ; show stdimage
6. **Execution time***:* Preceding the command string by a "**$**" sign (no blanks between "**$**" and taskname!), IRAF displays the CPU time once the task is terminated. Ex:
   cl> $imstat dev$pix fields="mean,midpt,stddev,mode"

Foreign tasks
The ***login.cl*** file contains definitions of commands which are declared as ***$foreign***. They are not tasks written outside of the U.S., and for which the CIA and the FBI keep an eye out, but rather Operating System-level commands that can be used similarly to IRAF tasks, with arguments, input or output redirected, used with pipes. However they cannot be used in background, nor have parameter files. Also they may not work well with Iraf pathnames, compare for instance: **cl> ls images$** with **cl> dir images$**

Warning: If you happen to abort a task (Ctrl-C), type **flpr** twice to clean up IRAF memory, which can get corrupted by the abortion. Sometimes, if IRAF behaves strangely, a couple of **flpr** might solve the problem. At times, it may be necessary to logout of IRAF and start again.

Quiz 1:
A user, who is using a VAX/VMS system, would like to know whether also the ***foreign tasks*** may participate in I/O redirection and piping (which are not implemented in the VMS Operating system). What should he do?

Quiz 2:
A user just started to work with IRAF at his workstation when he realizes that the ">" key in her keyboard does not work anymore;

worse, she was planning to do a lot of output redirection, thus using that key very frequently. What can he do to keep on working until the ">" key gets fixed?

Note: The standard IRAF installation include some images and files with which to do experiments, try tasks, etc.
2 images are available: ***dev$pix.imh*** and ***dev$wpix.imh*** ; the latter includes the WCS in the header (see Chapter 4). Also, there is a ***metacode*** file, called ***dev$vdm.gki*** .
Furthermore, directory ***scidata$*** contains a variety of HST-related images and tables.

IRAF Paso a Paso: 3 - The CL command language

File and image templates

- **File templates**
  can be used in IRAF; the syntax is equivalent to UNIX's. Ex:
  cl> imhead noche1*.fits
  cl> imstat noche?_001.fits
  "*" means: whatever string you find, irrespective of length; "?" means: whatever character you find (but just 1 character).
- **@files**:
  text files containing a list of images (called "***at-files***"). Many IRAF tasks can operate on single images as well as on ***@files***. They can be used both in input and output. Ex.:
  cl> files noche1*.fits> n1.txt
  cl> imstat @n1.txt
  Dumps list of files matching template noche1*.imh in n1.txt; use n1.txt as input for imstat task.
  Being plain text files, @files can be edited with any editor.
  Be careful not to include other columns or extra characters in a @file, as it can confuse Iraf and produce sometimes cryptic error messages.
- **String replacement**
  is performed using the "%" character. Ex:
  cl> imarith noche1*.fits / flat1.fits %noche1%ff_night1%*.fits
  All images matching noche1*.fits are divided by flat1.fits; output images are called ff_night1*.fits.
  The more intuitive command:
  cl> imarith noche1*.fits / flat1.fits ff_night1*.fits
  will not work.
  Alternatively, the "//" operator can be used to append a string. Ex:
  cl> imarith noche1*.fits / flat1.fits ff_//noche1*.fits

IRAF usually stores commands you typed in a memory stack, and allow to go back through it and use/edit previous command. Also, a permanent record of commands can be kept.

Command	Description
history 40	Displays the last 40 commands
e	Go back to previous command
^	Go back and execute previous command
e imst	Go back to previous command starting with "imst"
keeplog = yes	All commands are stored in a file
logfile = "home$cllogfile.cl"	Name of file where to keep cl commands

Keeplog and logfile are cl parameters (do: **cl> lparam cl**).

I never use this capability, as I prefer to write all commands I use (together with relevant output and comments) in my own logfiles. Also, **ecl** now allows to go through the history just using the up and down arrows.

Unix-like commands

- IRAF has a set of commands that perform the same things as the analogous UNIX commands (ex: copy, delete, head, directory). Most of these commands are grouped in the "**system**" package (type: **help system**).
  Some useful ones are:

cd *subdirectory*	goes to subdirectory
lprint *file*	prints file
path	shows current directory
rename	renames file
page *file*	displays file using page capabilities

- To use directly UNIX commands, escape them with a "!". Ex:
  cl> !rmdir oldfiles
  remove directory "oldfiles" (there is no IRAF command to do that).

CL as a pocket calculator

Not all people know that IRAF provides a simple yes powerful calculator. It may compute simple expressions, or use predefined variables and built-in functions to do more complex operations.

- Examples of built-in functions are: ***sin***, ***cos***, ***exp***, ***log***, ***sqrt***, ***max*** etc. See **help mathfcns**.
- Computations are preceeded by a "=" sign, ex:
  cl> =tan(1.5)/sin(20)*log10(11.5)
  cl> =12:27:14 (convert sexigesimal to decimal notation)
  cl> x=12.5 ; y=log(x)*24.2/sqrt(3.0) ; print ("x=",x," y=", y)
  cl> x=3838198 ; y=0.22 ; z=x*(1-y)/12. ; print (z,"ptas per month")
  (compute after-tax monthly salary of a post-doc, in pesetas, maaany years ago).
  An expanded set of capabilities, such as compute the julian day, work on image keywords, etc. is provided by the task **astcalc**.
- The predefined variables of cl are:
  b1, ***b2***, ***b3***: boolean
  i, ***j***, ***k***: integer
  x, ***y***, ***z***: real
  s1, ***s2***, ***s3***: string
  line: struct
  list: list-directed struct
- New variables may be defined; their type must be declared, ex.:
  cl> real w=1.5
  cl> real madrid
  cl> string filename="data.txt"
  Something like **cl> i1 = 0** won't work.

Do **cl> lpar cl** to see CL's own parameters and variables

Some useful tricks

- Sometimes, it's useful to have the output from a command read into a CL variable. For instance, a header keyword contains the logarithm of the starting wavelenght. To get the wavelenght in Angstroem, we might do:
  cl> hedit spectrum.fits CRVAL1 . | scan(s1,s2,x) ; print(10x)**

  scan is a useful function that parses its input (typically from a pipe, as in the above example) and fills in the variables with the corresponding values. Another example:
  cl> imstat dev$pix fields="npix,mean,stddev" format=no | scan(x,y,z)
  cl> print ("npix=",x," mean=",y," standard deviation=",z)
- A task parameter can be changed from hidden to required or viceversa, with the command:

cl> **display.zs.p_mode = "a"**
cl> **display.frame.p_mode = "h"**

- There is a way to debug script, using the "***d_trace***" toggle. Type it once to enable debugging, type it again to switch debugging off.
- Converting numbers to/from decimal to sexagesimal value is quite simple. In the first case, just type, for instance:
  cl> **=14:12:25**
  To do it the other way around, that is to print the decimal number with the appropriate format:
  cl> **printf('%16h \n',123.567875)** convert to degrees
  cl> **printf('%16H \n',123.567875)** convert to hours

IRAF Paso a Paso: 4 - Working with images

Image formats

IRAF can read, write and work with a variety of formats. Multidimensional images are allowed (for instance with 3, 4 up to 7 axes). Here is a list of the formats it uses:

Format	Description
FITS	Flexible Image Transport System - The "Universal" format. Starting from V2.11, IRAF can use FITS in virtually any tasks dealing with images. The FITS format now allows multiextension format, i.e. a number of images in the same physical file (used for mosaic data, for instance).
OIF (.imh + .pix)	Old Iraf Format This was the "standard" IRAF format. Now FITS is the de-facto format used by Iraf. File ***image.imh*** contains the image header, file ***image.pix*** contains the pixel data. In ***image.imh*** is the information about in which directory ***image.pix*** is.
STF (.hhh + .hhd)	Space Telescope Format The standard format used for HST images; analogous to OIF. ***image.hhh*** is a plain ascii file; ***image.hhd*** contains pixel data. Also, a variety of extension of the type "***.??h***" and "***.??d***" exists for HST images, depending on the instrument (WFPC2, FOC, etc.) and the type of data (raw, data quality file, etc.). Multigroup format is often used (ex. the images of the 4 WFPC2 chips are in the same physical file).
Mask (.pl)	A format used to identify bad pixels. Values is 0 for good pixels, non-zero for bad (masked) pixels.

Unless there are specific reasons to favour another format, you are better off using FITS as the default format. Indeed, FITS files can also be used directly in other applications (for instance read directly from DS9, or opened with PyFits).

- Pixel data are stored in a variety of formats, ranging from ***short*** (16-bit signed integers) to ***double*** (64-bit floating point). The most used one is the ***real*** format (32-bit floating point).
- Numerical precision used in computations is the highest one of the images used, or may be set as task parameter (ex: ***imarith.pixtype*** and ***imarith.calctype***).
- **Do not perform image processing on integer data**, except for basic bias subtraction, etc., as some tasks may produce funny results; transform them to real data first.

Also, IRAF can operate on tabular data both in ASCII and TABLE formats. The STSDAS **ttools** contains a variety of tasks to work on tables.
STSDAS tables are in binary format, and their header contains all the info (name of columns, units, format, etc.) about the table content. The table I/O software can read/write also plain ASCII tables.

- Files should be written with their extension to allow IRAF to recognize their format (though normally it is smart enough to sense the format by itself).
- Whenever writing a file, IRAF uses either the format specified in the file extension (ex: **imcopy im1.fits im2.imh** writes a OIF image), or uses the default format set in the "***imtype***" variable. (Actually, the true story is more complex...).

- If possible avoid having two files with same root name but different extension (ex: spec1.imh and spec1.fits).

IRAF Coordinate System

IRAF supports 3 different coordinate systems:

Coord. system	Description
Logical	Pixel units, relative to the current image. Starts from (1,1) = leftmost bottom pixel.
Physical	Similar to logical, but reference system is the one of the original image. Example: **cl> imcopy image.imh[10:500,20:500] subimage.imh** pixel (1,1) in logical system of subimage.imh has physical coordinates (10,20).
World	Coordinate system having astronomical meaning. Might be Wavelenght, (RA,DEC), etc. The image header contains all the info on the type of world coordinate system (=WCS), and the parameters for the transformation between pixel units and world coordinates.

Package **imcoords** contains tasks that deal with the World Coordinate System of an image. Also, several tasks may use the WCS, ex. **imexamine, implot, listpixels**, etc.
As an example, two images of the same field, but taken by two different instruments with different scales and orientations, may be registered to a common reference system by using only the WCSs of the two images (if accurate enough).
Sometimes, a task (or a user) might get confused by the physical or world coordinates. They may be deleted with the task **wcsreset**.

Image ***dev$wpix*** contains WCS keywords in its header, so it can be used as a test image to explore the WCS.

Input/Output from tape

Probably one of the most problem-prone and dreaded operations with IRAF.

Normally, images on tapes are in FITS format. To read them from a tape, do the following things (assuming you are already with the **cl>** prompt, are in the right directory, and the "dataio" package is loaded):

1. Mount the tape on the appropriate device. Take note of the device name and of the machine it is connected to.
2. **cl> allocate machine!device**
   where "machine" is the name of the workstation that owns the tape unit, "device" the name of the device (ex: nido!mtexa)
   If unsure of the device's name, do **cl> devices** to have a list of the devices, or consult the SIC Web pages.
3. Read the images with the command:
   cl> rfits machine!device 1-10,15-98,110 WHT_
   Files 1 through 10, 15 through 98, and file 110 will be read. Name of files will be: WHT_0001, WHT_0002 ..., WHT_0015 ..., WHT_0110. Format will be the one defined by variable ***imtype***
   Usually, the default hidden parameters for rfits are OK.
4. Once finished, deallocate the tape (IMPORTANT).
   cl> deallocate machine!device
   and remove the tape.

- To get a list of the tape content without actually loading the images:
  cl> rfits machine!device 1-9999 junk make- > tape.list
  You can also use **fitsio.catfits**
- To check the status of the device:
  cl> devstatus machine!device
- If you have a tape, but do not know exactly what is in, not even the format of the tape, the task **mtexamine** might provide useful hints.
- If you get strange error messages, or are asked your password, quit your session, and rlogin in the machine tied to the tape unit: use IRAF from there (thus you do not need to write the "machine!" thing)
- With HST WFPC2 images, use **strfits** instead of **rfits**.

Writing to tape is very similar. Do the same allocation/deallocation operations described above. Now to write to the tape:

- Save directly the files to tape:
  cl> wfits im_final_00*.imh machine!device newtape+
  Or, write a text file containing the list of images you want to save (one image per row), either using the editor or with the "files" or "sections" tasks:
  Ex., if "totape.txt" already lists some other files to be saved together with m_final_00*.imh, you might do:
  cl> files im_final_00*.imh >> totape.txt
  cl> wfits @totape.txt machine!device newtape+
  Files matching im_final_00*.imh are appended to file "totape.txt". All files listed in "totape.txt" are saved to tape.
- The default hidden parameters normally have the right value. Be careful with "***newtape***": if =yes, the entire tape will be overwritten; if =no, images are appended at the end.

You are not compelled to store files on tape in FITS format. In most circumstances, a ***tar archive*** is faster and simpler.

Getting info on an image

All the information about an image is contained in its header (except the pixel data, of course). You can see and also modify the header content.

- Look at size and pixel type of an image (basic header info):
  cl> imheader dev$pix long-
- Prints all the keywords:
  cl> imheader dev$pix long+
  cl> hedit dev$pix * .
  (last command can be used to look at a subset of keywords)
  cl> hedit dev$pix $I,i_title,DATE-OBS,OTIME .
  (***$I*** = file name; ***i_title*** = identifier)
- To get a nice tabular output, useful especially for HST images:
  cl> iminfo dev$pix
- To see whether the image contains sensible values (or is all 0's):
  cl> imstat dev$pix
- Hedit also allows to modify a keyword. Ex. to change image's title:
  cl> hedit prueba.imh i_title "changed title" veri+
- Arithmetic operations using keywords are allowed in hedit, for instance:
  cl> hedit dev$wpix CCC "(2*(CRPIX1-100))" veri+ add+ show+
  (use ***add+*** to create a new keyword).

Display an image

First you have to select your display tool (normally DS9). Then, use the **display** task to load the image into the image display.
Since there are a number of things you can do with this task (not to speak of the image display itself), let's have a look at a few illustrative examples:

- **cl> display dev$pix 1**
  Basic command. 1 is the channel the image is loaded in (there are 16 channels available for XImtool and DS9).
- **cl> display dev$pix 1 zr- zs- z1=30 z2=1000**
  Disables automatic calculation of dynamic range, which is set by the user.
- **cl> display dev$pix 1 zr- zs- z1=(200-3*60) z2=(200+3*60)**
  Simple expressions, surrounded by parenthesis, can be given to the parameters in the command line.
- **cl> display dev$pix 1 ztran=log**
  Applies a logarithmic transformation to the intensity mapping
- **cl> display dev$pix 1 xmag=0.5 ymag=0.5**
  Image is squeezed by a factor two before loading (useful if the image is bigger than the display window).
- **cl> display dev$pix 1 xmag=3.0 ymag=3.0**
  Image is zoomed in by a factor 3 upon displaying
- **cl> display dev$pix 1 fill+**

Automatically resize image to fill the image buffer size

- **cl> display dev$pix 1 xcen=0.25 ycent=0.25 erase-**
  Image is centered on coordinates (0.25,0.25) in the display window (whose limits are 0-1). Previous image is not erased.
- **cl> display image1 1 ; display image2 2**
  Two images are loaded on two different display channels, allowing to switch easily to one or the other or "blink" them.
- **cl> display image1 1 bpm=badpix.pl bpdisp=overlay** overlay on the display the bad pixel areas from the badpixel mask.

Display also allows to mosaic more images on the image window, allowing a direct comparison, for instance, of the same field observed through different filters, or same filter but different epochs. Here you can find an example of script to do this.
There is also a CL task to do this: **stsdas.graphics.sdisplay.mosaic_display**

Do not forget to set the variable ***stdimage*** to the appropriate size. A complete list of the sizes available is in the file ***dev$imtoolrc***.

Working with image sections

It is generally possible to work with just a region of an image. To define what region to work on, use the syntax:
[Xstart:Xend,Ystart:Yend].
Ex: dev$pix[100:300,150:350] goes from pixel X=100 to X=300, Y=150 to Y=350, so it's a 201 x 201 region.
A "*" means: all pixels. Ex. [*,100:200] is a horizontal slice of the image, of same length in X as the original, but reduced Y size.

Many tasks accept image regions (in input; rarely on output); for instance:

- **cl> display dev$pix[200:300,200:300] 1 xmag=4 ymag=4**
  Only the central portion is displayed, with zooming factor = 4
- **cl> imstat dev$pix[200:300,200:300]**
  Statistics on central region only
- **cl> imcopy dev$wix[200:300,200:300] devpix_small.fits**
  Copy central region to new image. World coordinate system is automatically updated to take new size and center into account.
- **cl> imcopy dev$pix[201:300,201:300] other[1:100,1:100]**
  copy central part of dev$pix into region [1:100,1:100] of preexisting image "other" (this way a mosaic of 2 or more images could be done).
- **cl> imcopy im3[*,*,3] im2_new**
  "Plane" 3 of im3 (a 3-D image) is copied on im2_new (a 2-D image)

Moreover, the image section notation, coupled with the **imcopy** and **imtranspose** tasks, allows simple coordinate transformations.

Command	Description
imcopy dev$pix[*,-*] new1	flip the rows (image is upside down)
imcopy dev$pix[-*,*] new2	flip the columns (image is inverted left-right)
imcopy dev$pix[-*,-*] new3	image is rotated 180 degrees
imtranspose dev$pix[*,-*] new4	image is rotated 90 degrees counterclockwise
imtranspose dev$pix[-*,*] new5	image is rotated 90 degrees clockwise

---

Create empty images

Sometimes, it may be necessary to create empty images, i.e. with all pixels set to 0 or some other value.
The easiest thing is to use an already existing images, multiplying it by 0, using the **imarith** task.
Alternatively, you can use other tasks to create an image from scratch, such as:

- **cl> mknoise newimage out="" ncols=512 nlines=512**
  A new 512x512 image is created. The background level can be set to any value, and noise can be added (see the help for more details).
- **cl> imexpr "I*0.0+J*0.0" newima dims="512,1024"**
  A 512x1024 image called "newima" is created. In the above expression, ***I*** and ***J*** are the pixel coordinates, so that images of any kind of analytic form can be created by changing that expression.

Hardcopies of images

There are essentially two ways of printing out an image:

1. Using the "print option" of the image display (ex. DS9)
2. Using the **export** task in package "**dataio**". This task provides a number of output formats (EPS, GIF, etc.), and a variety of lookup tables, color transformations, etc.

- **DS9**
  The command **File -> Save Image as** allows to save the content of the display to a ***FITS, JPEG*** and other formats. The **File -> Print** command allows to print to a printer or to a file, with the option of using the ***CMYK Color*** format (the one for instance requested by ApJ for color figures).
  See the DS9 help for more details.

- **dataio.export**
  A range of file formats is available: EPS, GIF, RAS, RGB, etc. Moreover, it is possible to select a particular lookup table or intensity scaling. The options are many and not at all straightforward, so a few simple examples might be useful:

  cl> export mosaic_f336w A336W.ras ras outbands="zscale(i1,250,20)"
  save image on a rastefile format. The intensity range: 250-20 means that will have black sources (`i1>250`) on a white background (`i1<20`).

  cl> export mosaic_f336w B336W.ras ras outbands="zscale(log10(i1),2.3,0.5)" +
  In this case we have applied a logarithmic transformation to the intensity scale. Limits are 2.3 and 0.5.

  cl> export dev$pix dpix.gif gif outban='setcmap(zscale(i1),"heat")'
  Convert dev$pix to a GIF file using the builtin "heat" colormap and default intensity mapping

Moreover, a special task **color.rgbsun** creates a "quasitruecolor" image.
It needs three images of the same field, taken in three filters at different wavelengths. One image is associated to "blue", one to "green", one to "red". The output is a Sun 24-bit RGB rasterfile.
If the correspondence between filters and the above colors is good (for instance, a B, V, R set), the resulting image will have colors similar to the ones that would be observed by the naked eye. For an example, see:
file to create color image of a WFPC2 field

There is also a task called **import** which creates an IRAF image from a GIF, RAS, or other format. Its help is even more obscure than export's one...

Other options at the host level permit to save a displayed window as a raster image; for instance, in Unix the **xv** tool.

Working with tabular data

The TABLES package contains many tasks to handle tabular data of different types. Many analysis tasks in STSDAS indeed produce more or less complex tables as output. STSDAS tables are generally binary files with data stored in rows and columns; also, each element may be itself a vector, therefore forming sort of a 3-D table. Their default extension is "***.tab***".

Most of the tables-related tasks are in the **ttools** package. While they can unleash all of their power on well formatted and structured tables, many of those tasks can also work on plain ASCII tables, that is simple text files in row and column format. Here we illustrate a few basic operations we can do on ASCII tables (the file ascii_table.dat may be used as an example).

We can get info on the table by using the commands:
cl> tinfo ascii_table.dat ttout=yes and
cl> tlcol ascii_table.dat nlist=4

To perform an arithmetic operation, we use the task **tcalc**:
cl> tcalc ascii_table.dat norm "(c7-390.0)/c3"
As we have not given any id to the column tables, we use "c7" and "c3" to refer to their position. The output column is called "norm" and is placed as last column.

We can select a subset of rows using task **tselect**:
cl> tselect ascii_table.dat ascii_table_subset.dat 'c5=="2003-03-31"&&c4>19.0'
In this specific case, we select all flat-fields taken on March 31 in the evening.

cl> tcalc ascii_table_subset.dat tdiff '60*(c4-19:28)'
We create column tdiff which says how many minutes since the sunset the image was taken. Notice that we can mix decimal and sexagesimal formats.

cl> tproject ascii_table_subset.dat ascii_table_project.dat "c1,c2,tdiff,norm"
Select relevant columns from the input table.

Now we can use any plotting program to display the sky brightness, through the various filters, as a function of time since sunset. Or, we can use task **sgraph**, briefly described in The sgraph task

Exercise:
The file tablafinalV.dat refers to a set of flat-field images taken before sunset. Column 2 lists their mean value, column 3 the exposure time and column 4 the time the image was taken. Can you produce a plot with in X the time before sunset (for simplicity assume sunset = 7:30:00), and in Y the sky brightness (in arbitrary units: just normalize the observed counts by the exposure time)?

Another interesting task is **tjoin**. It permits to join two tables into a new table on the basis of one (or more) common columns. For instance, if you have a table with a list of stars, with, say, ID, positions and magnitude, and another table with a similar, but different, list of objects with ID and radial velocities, you can produce a table with all objects common to both tables (from their ID) together with positions, magnitude and radial velocity.

As en exercise, we can join tabla_coord.dat and tabla_param.dat, and produce a final table containing all objects common to both tables:
cl> tjoin tabla_coord.dat tabla_param.dat tabla_out.dat c1 c1

A similar task is **tmatch**.

IRAF Paso a Paso: 5 - Graphics tools

The graphic cursor

- Many tasks in IRAF produce a graphic output as their final step (ex. a contour plot) or as a way to interact with the image data (ex. **splot** to measure interactively line centroids, widths, etc.).
- Normally each task has its own set of commands to operate on the plot, but there is a set of cursor keystrokes available to any plotting task.
- These ***global cursor keystrokes*** can be listed in interactive cursor mode by typing **:.help**; the cursor commands specific to the task are listed by typing **?**
- Global cursor keystrokes are usually upper-case or "**:.**" commands, so as to be distinguished from the task's commands.
  More help is available doing **phelp cursor**
- Global cursor keystrokes are available also on non-interactive plots, and are invoked with:
  cl> =gcur

As an example, we may use **implot dev$pix**, get help on task's and global cursor commands, do a few operations, than quit. Typing **=gcur** we are back in the graphic window.

Printing/saving graphic output

1. The most straightforward way to print out a graph is to set the ***device*** variable to ***lpr*** or ***lw4*** or any other printer known to IRAF. Ex:
   cl> implot dev$pix 256 dev=lpr ; gflush
   Normally a **cl> gflush** is required to actually start the printing operation.
2. Alternatively, the graph can be saved on a postscript file, ex.:
   cl> implot dev$pix 256 dev=psi_land ; gflush
   The graph is save on a postscript file in the current directory (or in directory /tmp) in landscape format. After **gflush** IRAF should show the file name, or just look in the directory for a new "***pskXXXZX.eps***" file. Rename/move the file to an appropriate place. Useful if you want to check out the output before printing (ex. with ***ghostview***).

Notice that **cl> gflush** erases the graphic buffer (You lose the plot from the memory).

3. There is also the possibility to save/print the graph interactively, by doing, in interactive cursor mode:
   :.snap [device] and then **:.gflush**, where ***device*** might be ***lpr*** or ***psi_land***, for instance. (If you do not specify the ***device*** , the value of variable ***stdplot*** is used).
4. A (longish) list of all availabe devices can be obtained by examining the file: ***dev$graphcap***.

Graphics on the display window

Sometimes it is useful to redirect the output on the display window.
For instance, you overlay the isophotes of a galaxy from a broad-band image on the narrow-band image; or you overlay a coordinate grid.

Example 1: Overplot coordinate grid (also task **wcslab** can be used)
cl> disconlab dev$wpix disp+ cont- label+ left=0.1 right=0.9 bot=0.1 top=0.9

Example 2: Overplot HST apertures (also task **siaper** could be used)
**cl> disconlab dev$wpix disp+ cont- label- doaper+ ra=13:27:45 dec=47:27:00 **
roll=0 left=0.1 right=0.9 bot=0.1 top=0.9

Example 3: Mark on the image display stars listed in a file (ex. output from DAOFIND):
cl> tvmark 1 devpix.coo mark=circle radii=5 numb+

Note that sometimes the overplot may be bad, for instance the contours may apear offset with respect to the displayed image. There could be many reasons for this, for instance some parameters in **disconlab** as well as in the tasks called by it (**wcslab, newcont, display**) may not be set correctly, or the header of the image is not standard and Iraf gets confused and misinterpret the coordinate system.

Redirection of graphic output to image window can be done by setting **device=imd*X***, with ***X=w/r/g/b/y***: w=white, r=red, g=green, b=blue, y=yellow.
Ex. **device=imdg** overplot on image display in green.
It is not recommended to use this redirection with general graphics tasks, as it is not fully supported (and probably there is no point to do that). It's better to use only the specialized tasks for this purpose.

Image cursor from graphic window

It is possible to force tasks that normally interact with the image display to use the graphics window instead. This is done by setting:
cl> set stdimcur = stdgraph
which means: the interactive image cursor now operates on the graphics window.

Example: a user has done a contour plot of a image
cl> newcont dev$pix floor=40 ceiling=1000 ncont=10 per+ pres+
After setting ***stdimcur*** as said above, he can then use **imexamine** to do some statistics on the image.
cl> imexam (use keystroke "m")
Though, if he uses an imexamine's keystroke which produces a graphic output, he loses the contour plot (I do not know if there are workarounds).

I have no particular examples where this trick might be useful (I never used it).

The sgraph task

The **sgraph** task provides a general purpose plotting capability, as it can be used to plot from lists, images and tables. We can control the characteristics of the graphic output via three psets: ***devpar, pltpar, axispar***; parameters from these sets can also be placed on the command line.

- **cl> sgraph "ascii_table_project.dat tdiff norm" pointmode=yes szmark=0.01**
  Plot "norm" column against "tdiff" column. The example is taken from Working with tables.
- **cl> sgraph dev$pix[50:100,*] axis=2**
  Graph the average of columns 50 through 100.

  cl> sgraph dev$pix[*,10],dev$pix[*,20] xlabel=column
  >>> ylabel=intensity title="lines 10 and 20 of pix"
  Annotate the graph.

Be careful with the ***dvpar.append*** parameter. It can be set to "yes" if you want to overplot new data over existing ones, but then do not forget to set it to "no" when you want to make a fresh new plot.

Metacode files

- There is also a way of saving graphics output in a ***metacode file***. These files can be created by the task itself, or may be created by redirecting the graphic output (**>G** syntax), or written interactively (cursor command **:.write**).
- A few tasks exist that deal with metacode files: **gkidir, gkimosaic, gkiextract, stdgraph, stdplot**.
- A metacode file: ***dev$vdm.gki*** is available for experimentation.

As an example, I mention a couple of situations where I found it useful:
1) Fit non-interactively gaussian profiles to spectral lines (ex. task **fitprofs**). The overplot of the fitted profile on the datapoints can be saved in a metacode file and examined later.
2) **imexamine** is used to plot the radial profile of a list of objects. The profiles can be saved in a metacode file using the **>G** redirection option.

IRAF Paso a Paso: 6 - Important tasks

There are some IRAF tasks that are particularly important, as they allow to do a quick assessment of the nature and quality of the data, or to do a variety of operations. Here is a list of the main tasks (Some, such as **display** and **imheader**, have already been described in a previous chapter).

Imexamine

The task **imexamine** allows to do quite a lot of different measurements on an image.
It is one of the more complex tasks, as it has a big number of command keystrokes and task parameters, grouped in ***psets***.
The following table lists just a few of its possibilites:

Keystroke	Action
m	Statistic of a region around the cursor
r	radial profile of the star under the cursor; measures center, FWHM. etc.
h	histogram of a region around the cursor
s / **e**	surface / contour plot
c / **l**	column / line plot

Parameters are grouped into 9 ***pset***, which can be edited to change parameter values.
Parameters can also be set interactively using the so called ***column commands***, of the form **:ncstat 10** (number of columns used in statistics is set to 10).
imexamine allows to print logical, physical and world coordinates. In the latter case, **xformat=%H** and **yformat=%h** return the correct sexigesimal notation for RA and Dec (if the image contains the relevant and correct WCS keywords).

You can also load directly the image in the DS9 display, and the use imexamine. This what they say in the DS9 Web site:
"Due to the unique relationship between DS9 and IRAF, if you use the **imexamine** task, you can take advantage of a special feature of DS9. Instead of loading the image from IRAF with the display task, load the image directly into DS9. Then, from the cl prompt, invoke imexamine without a filename. IRAF will ask DS9 for the current filename and use it for analysis. This approach provides serveral advantages over previous methods. First, it will work with compound fits images such as mosaics, data cubes, and rgb images. Second, the image diplays includes true image data and WCS information, not the approximated data from IRAF."

Imedit

The task **imedit** provides a way to interactively editing pixels values in a image.
A few examples:

- Replacement of a image region to a given constant
- Interpolation across columns or lines
- Replacement of a region by another region

It is a useful task for removing interactively cosmic ray events, if the automated tasks fails or produce unsatisfactory results. For instance:

cl> imedit image.fits c_image.fits logfile="imed.log"
edit "image.fits", output is "c_image.fits", all cursor commands are stored into file "imed.log".

cl> imedit image.imh "" cursor="imex.cur" xord=1 yord=1
edit "image.fits" in place (output image will have same name, old image is lost), commands are read from file "imex.cur", set polynomial order to 1,1 when interpolating across bad regions.

Imexpr

The task **imexpr** is a powerful image calculator.
It accepts complicated operations, with built-in functions and logical expressions. Also values of image keywords may be used.
A few examples are in order:

1. **cl> imexpr " a * (a.photflam/1.0E-18) / (a.exptime/1000) " testu a=mosaic_f336w**
   Pixel values are rescaled taking exposure time and the HST photometric keyword into account. Notice the syntax: ***a.photflam*** is the values of keyword ***photflam*** of image "a". `a=mosaic_f336w` assigns image ***mosaic_f336w*** to variable ***a***.
2. **cl> imexpr " (a > 4000) ? 4000 : a / b " result a=image1 b=image2**
   If ***a > 4000***, set pixel value in resulting image to 4000, otherwise output is ***image1*** divided by ***image2***.
3. **cl> imexpr "I+J" plane dims=400,400**
   The 500x500 image ***plane*** is created, whose pixel values are function of their coordinate.
   Notice: ***I***, ***J*** are variables representing the pixel coordinates.
4. **cl> imexpr gauss.fits "1000.*exp(-((I-256.)**2+(J-256.)**2)/(2*50.0**2))" dims=512,512**
   Create a gaussian

Quiz 1:
A user has created image "llog" with the following operation:
cl> imexpr "100-100*log10(abs((I-256)/20))" llog dim=512,512
However, plotting line #256 (**cl> prow llog 256**) he sees something strange: the profile is not smooth, but presents marked and regularly-spaced steps. Why?

Quiz 2:
A user wants to know whether integer coordinates apply to the pixel center, or to one of its corners. Using the tasks imexpr and display, how can he find it out?

Splot

The task **splot** is, in a sense, the analogue of imexam for spectral data. It allows to do various types of measurements and processing on a spectrum. Some of the most important cursor keystrokes are:

Keystroke	Action
k	fit a gaussian to a line profile
d	deblend a group or blend of lines into single components; several functional forms are allowed.
f	perform arithmetic operations on the spectrum

t	fit a function to the spectrum. Might be used to normalize by or subtract the continuum
#	select new line or aperture

Sections

Task **sections** (and the similar but less powerful task **files**) is very useful to check or produce file and image lists.
It can handle also image sections, as well as the Unix-like notation to restrict the list to a given alphabetical range.
Its typical use is to produce a ***@file*** which in turn is passed as input or output to another task.
As usual, a few examples are in order to clarify:

1. **cl> sections *.fits[200,400,200:400]**
   Generate a list of images with the image section appended
2. **cl> sections noche![3-9]*.fits[1:10,*] > noche.txt**
   Generate a list of images whose name start with noche3 ... noche9, dump list on file "noche.txt". Character "!" tells the task that the [..] string is part of a file name, not an image section.
   Other tasks are able to use this sintax.
3. **cl> sections objs*.fits[100:200,300:400] > objslist**
   cl> sections skys*.fits[100:200,300:400] > skyslist
   cl> sections %objs%bck%* > bcklist
   cl> imarith @objslist - @skyslist @bcklist
   Subtract two set of images, ouput images will be 101 x 101 pixels. Notice the use of the substitution character "%".

The task **files** is similar, but can't use image sections, nor the "![..]" range notation.

In general, it is preferable to use "***@files***" than image templates (i.e. wildcards) as input to IRAF tasks, as there is no guarantee that image templates are expanded in the same order. For instance:
cl> imarith gal*.fits - back*.fits sub//gal*.fits
"gal*.fits" and "back*.fits" might be expanded in a different order, thus losing the correct pairing between the two images lists.

Hselect

Task **hselect** can be used to select a subset of images satisfying specific criteria. For instance, we may want to get a list of all images taken in the V filter, or those with exposure time = 0 (biases), or whose title contains the string "NGC 5555". We can also decide which keywords from the header we want to appear in the output list.

As usual, a few examples can better illustrate its usage:

1. **cl> hselect*.fits $I,title * 'FILTER = "V"'**
   Generate a list of all images whose keyword "FILTER" is V. The output includes the image file name ***$I***, and its title. Notice that alphanumeric variables must go within quotes.
2. **cl> hselect*.fits $I,title,exptime,filter 'exptime > 59.0' > long.txt**
   Generate a list of images whose eposure time is equal or longer than a minute. The output contains the image file name, the image title, exposure time and the filter, and it is saved in the file "long.txt". Before using this task, it is a good idea to inspect the exact name of the keywords (for instance by using **imhead l+**); for instance in some cases you can find "filt" instead of filter, or "exp-time" instead of "exptime", etc.
3. **cl> hselect*.fits $I 'exptime> 59.0 && filter = "V #57"'**
   Both conditions, on exposure time and filter, must be satisfied.
4. **cl> hselect*.fits $I,title 'title ?= "flat"'**
   The task looks for all files whose title includes the string "flat" (the ***?=*** operator means contains). This is different from looking for images whose title is exactly "flat". Notice that we put the string "flat" within quotes not to get the task confused.

Other operators are ***!=*** not equal, || or, >= greater or equal, etc. Type **help hedit** for more info on conditional expressions, functions and image keywords in general.

IRAF Paso a Paso: 7 - CL scripts

This is a brief introduction to CL scripts, to show that they are not as difficult as it might seem at first sight, and that they might be useful from time to time, especially the terminal scripts.

Basic scripts

In its simplest form, a CL script is a logical, ordered sequence of CL commands. No variables or special syntax are used in this type of scripts.
In other words, instead of writing commands at the **cl** prompt, they are written in a file and then executed.

Advice
It is a good idea to always write down the commands in a file, and then either execute the file, or just cut-and-paste the commands from the file to the IRAF window.
In this way all you are doing is stored in a file, in a much better organized way than just using the CL logging option; editing is easier, and you can add your own comments.

Here is a list of some of my own (old) CL script files that I have used or am using. Comments help understand what is going on, and sometimes also the output of a task is recorded for future reference.
Usually a group of commands is brought to the IRAF terminal by cut-and-paste.
N.B: Modern browsers may not allow direct access to the link below, because of security reasons. If so, open a new tab o window, and copy-and paste the links below in the new address box.

- produce color images
- recalibrate WFPC2 images
- run daofind on WFPC2/chip2 images
- linearize 2-D spectra
- gaussian fit to spectral lines

To make CL execute your procedure, do (assume your procedure is in file ***procedure.cl***):
cl> cl < procedure.cl
The "**cl <**" thing means that what is on the right-hand side is given in input to CL.

Terminal CL scripts

Next step is to write a script which uses some of the CL language capabilities, as strings, numerical variables, ifs and whiles, etc. We are still far from writing a task-like script.
This kind of script often saves from repeating many times the same command, just changing the value of one parameter or two each time.

Example: a user has a calibrated 2-D spectrum, and wants to dump each spectral line on an ASCII file having the wavelength in the 1st column, the pixel intensity in the 2nd. **wspectext** is the right task; the problem is to execute it in turn for every line.
The script might be as follows:

```
# script to wspectext each spectral line in turn on a text file.
# change name of image and output file if necessary.
#
# use with: cl < wmacro.cl

# assign file names to string variables s1 and s2:
  s1 = "ap96_"
  s2 = "w0096.ms"

# start loop:
for (i=1; i <= 111; i+=1)  {
  j = i+4
  s3 = s1//j
```

```
  print (s3)
  wspectext (s2//"[*,"//i//"]", s3, head-)
  }
```

Description:

1. A loop is performed, with variable ***i*** going from 1 to 111. Notice syntax
2. Variable ***j*** is set to ***i+4***
3. The name of the output file, in string ***s3***, is formed as a value of ***s1*** plus the value of ***j*** (note concatenation of a string and an integer)
4. The value of ***s3*** is printed out
5. The task **wspectext** is executed. Since there are string variables and string operations in its arguments, these are passed to the task using the parenthesized format (***program mode***). Notice how the image section is built by string/integer concatenation.

In this procedure, CL is used in the so-called ***program mode*** (or ***compute mode***), which takes advantage of the full syntax of CL as a programming language, where, for instance, it can process strings and variables.
The other mode of CL, with more limited capabilities but a simpler user interface, is called ***command mode***, and is the one generally used at the **cl>** prompt. It minimizes the need for command delimiters, quoted character strings, etc.

Here is a more complicate script to compute King models, and convolve them with the PSF.

```
# procedure to create King models and convolve them with
# the PSF
# A oversampling factor of 9 (good for Rcore>=0.15)
# is used to properly compute pixel values.
#
# use as:  Rcore=1.8 ; imname="King_180" ; suf="U" ;  cl < king.cl
# (Rcore and imname must be already defined)
#
string  expr, exprA, exprB, exprC
string  imnameC, outf, psfim
real    Tidterm, RcoS, Rtico, RtidS

set imtype = "fits"

# set variables
#
Rtico = 30                              # set tidal radius = 30 * core radius
Tidterm = 1.0/sqrt(1+(Rtico*Rtico))
outf = imname//suf//".prof"
RcoS  = 9.0*Rcore
RtidS = Rtico*RcoS                      # tidal radius
psfim = "psf.imh"


# compute King profile. The computed profile is oversampled
# 9 times (linearly), then is brought back to the original
# resolution by block-averaging
# King models must be set to 0 at R > Rtidal
#
# strings containing the matematical operations are defined, and then
# passed on to tasks imexpr and imcalc
exprA = "1.00+(((I-572.00)**2+(J-572.00)**2)/("//RcoS//"**2))"
exprB = "I*0.0+J*0.0+"//Tidterm
exprC = "(((I-572.00)**2+(J-572.00)**2) > "//RtidS**2//") ? 0.0 : a"
expr = "((1.0/sqrt(im1))-im2)**2"
imexpr(exprA,"King_A.fits",dims="1152,1152",intype="real",outtype="real")
imexpr(exprB,"King_B.fits",dims="1152,1152",intype="real",outtype="real")
imcalc("King_A.fits,King_B.fits","King_C.fits",expr)
imexpr(exprC,"King_D.fits","King_C.fits")

blkavg("King_D.fits",imname,9,9,option="average")
```

```
# normalize to total flux=1000.0
gstat(imname)                    # output in gstpar set
x = gstpar.sum/1000.0
imarith(imname,"/",x,imname)

# convolve with PSF (I-band, chip4)
# PSF must be normalized (sum=1)
stsdas.analysis.fourier
fconvolve(imname,psfim,imname//suf)

# remove temporary images
imdel King_A.fits,King_B.fits,King_C.fits,King_D.fits veri-

# output radial profile onto a file; sort it
pradprof(imname//suf,64.00,64.00,radius=20,center=no,list+,logx-,logy-, > outf)
sort(outf,column=1,numeric_sort=yes, > "sort.sort")
delete(outf,yes,verify=no)
rename("sort.sort",outf,field="all")

# end of script
```

To use this script, we should do:
cl < Rcore=1.8 ; imname="King_180" ; suf="U" ; cl < king.cl

Description:

1. Variable used in the script have been declared; those used in the above command line also must have been declared.
2. The name of the output file, in string ***ouf***, is formed as a the concatenation of string variables ***imname***, ***suf*** and the string constant ".prof".
3. The powerful imexpr task is used; the mathematical operations are defined in the ***exprN*** variables, and then passed to the tasks. Also these expressions are a concatenation of strings.
4. Various tasks are then executed. Since there are string variables and string operations in their arguments, these are passed to the tasks using the parenthesized format (***program mode***). When using the redirection (>), it must go inside the parenthesis.

Quiz 1:
There is a CL script that virtually every user is familiar with, and indeed uses pretty often. What is it?

Quiz 2:
A user wants to be able to start IRAF from any directory, and, once in IRAF, be right away in the directory he started from. Can you find a practical solution, using a Unix script + a CL script? (Just give the idea, not all the details)

Exercise:
Write a script that displays an image with z1=(mean-2*stddev) and z2=(mean+2*stddev), where mean and stddev are the mean and the standard deviation of an image. Of course these values must be computed inside the script.

CL scripts as new tasks

It is possible to write CL scripts that are similar in every aspect to genuine IRAF tasks. Actually, a good number of native CL tasks (i.e. present in the standard IRAF packages) are CL scripts themselves. Examples:
stsdas.analysis.gasp.regions - images.immatch.wcsmap - stsdas.describe etc.
These new tasks have their own parameter set, share the piping and redirection capabilities, might be run in background, etc.

Very simple example:

```
#  DESCRIBE --  A shortcut task to see the "Description" section of a help   #
#               file.                                                        #
#                                                                            #
#       8/91    Initial code by  PAHodge and RAShaw                          #
# Modified for clarity (NC).
```

```
procedure example (taskname)

string  taskname = "" {prompt = "Enter task name > "}

begin
        string  tasknm        # defines tasknm as a string variable
        tasknm = taskname     # reassign taskname to tasknm
        print ("\nDESCRIPTION for task: ", tasknm)    # print 1-line header
        help (tasknm, section="description")          # print help
end
```

This IRAF task simply displays the 'description" section of a help file.
Taskname is a required parameter, as is present in the task definition. There are no hidden parameters.
Note that the ***taskname*** variable is reassigned to a new variable, ***tasknm***; if they had used ***taskname*** in the following "print" and "help" commands, IRAF would have prompted for ***taskname***.

Another more complex example:

```
#  script taken from the "arnica" package. Adapted to clarify various
#  points (NC).
#  Cleans bad pixels using bad pixel mask and 3x3 median smoothing
#  CLEANED image has prefix b
#

      procedure clean ( sfrm )

string sfrm             { prompt="Frame name to clean" }
string smask = "arnica$badpix/bpf.fix.93jun" {prompt="Bad pixel mask"}


# Notice there is one required parameter (sfrm) and one hidden one (smask),
# which has a default value.

begin

        string msk1, msk0, junk0, intermed, sf, mask

        sf = sfrm
        mask = smask

# Make temporary images. Names will be like this: imf+PID+letter, and will
# be placed in the directory defined by the variable tmp

        intermed = mktemp ("tmp$imf")
        msk0     = mktemp ("tmp$imf")
        msk1     = mktemp ("tmp$imf")
        junk0    = mktemp ("tmp$imf")

# Median filter to smooth out image; output on temp file "ntermed"

        median ( sf, intermed, 3, 3  )

# Multiply smoothed image by mask
# Perform other mathematical operations, delete junk images, print output
# Notice how strings are concatenated.

        imarith ( intermed, "*", mask, msk1 )
        imarith ( sf, "*", mask, msk0 )
        imarith ( sf, "-", msk0, junk0 )
        imarith ( junk0, "+", msk1, "b"//sf )

        imdel (junk0//","//msk0//","//msk1//","//intermed, ver-, >& "dev$null")
```

```
        print (sf, ", ", "b"//sf)
end
```

A few comments to describe the above scripts:

1. The title of the procedure (in this case "describe" or "clean") must be equal to the rootname of the file (i.e., file must be called "clean.cl"/ "describe.cl")
2. All required parameters must be put between parentheses in the procedure naming line; hidden ones are the ones before the line "begin".
3. Parameters (both types) are declared, an initial value and a prompting string can be assigned. Prompt string will appear doing lpar or epar or dpar.
4. "begin" marks the start of the script body, "end" the end; both must be present.
5. New variables must be declared as strings, integers, etc.
6. Some files are created to store temporary images used during the processing. They are created in a temporary directory, and will be deleted at the end.
7. The expressions `sf = sfrm` and `mask = smask` are to avoid that IRAF prompt for ***sfrm*** whenever it is named, even in "imarith". For sake of prudence, the same reassignment is performed for the hidden parameter.
8. Arguments passed to the IRAF tasks are within parentheses, because tasks are used in ***program mode***. Constant strings are within "", variables are not.

To let IRAF know that these are new scripts, we do:
cl> task describe = describe.cl
cl> task clean = clean.cl
Magically, doing **cl> epar clean** or **cl> epar describe** will show the task parameters, which can be edited. Now they are tasks like any other one.

- The official User's Guide to IRAF Scripts suggest that scripts be written by using the **mkscripts** task. I find it rather clumsy, and prefer to edit them from scratch. Though, at the beginning it may be helpful, as it writes down all task parameters in the right order, checks the syntax, etc.

Quiz:
A user wants to normalize her images, that is divide the images in a given list by their mean, computed only in a certain range (say 100 - 7000) to avoid the influence of bad pixels. Use task **imstat** or **gstat**. Can you devise a script to do that? (just the idea, not the whole script). What if the range is different for every image?

Further things to know about CL scripts

1. To avoid confusion between string variables and constant, character strings should be quoted, expressions parenthesized.
2. It is possible to assign a value to a variable when it it declared. Ex:
   `string st = "file.dat"` or `real mag = "12.5"`
3. Never load a package within a script. Make sure all the relevant packages are loaded before starting the script.
   It is possible to check whether a package is not loaded and issue a warning with, for instance, the lines:

```
if (! defpac ("proto"))
print ("Warning: package proto not loaded!")
bye
```

4. Personal tasks may be arranged in a package, which will have the same characteristics as any other IRAF package. Also it is possible to add help files to personal tasks and packages.

Struct variables:
A ***struct*** variable is a special one with regards to reading from files using the "fscan" function. Basically, there are two type of this things:

1. Character strings treated in a special way by the ***scan*** and ***fscan*** functions, which set struct to be the remainder of the line be scanned without further parsing. Ex:

```
struct stru
fscan (list, s1, s2, stru)
```

   will read the file associated to variable list, and assign the 1st string encountered to ***s1***, the 2nd to ***s2***, the whole rest of the line to variable ***stru***.

2. Variable usually associated with a file name, whenever we want to use ***fscan*** to read lines from the file. In this case the declaration is of the form: `struct *file` with the "*" before the variable name. ***file*** is said to be a ***list-directed struct***

An example that contains the two structs:

```
procedure printfile (file_name)

  string filename
  struct *flist

begin

   struct line
   flist = filename
   while (fscan(flist, line) != EOF)
      print (line)
   flist = ""

end
```

This procedure will read from file ***file_name*** (a required parameter) and print every line in turn. ***flist*** is a list-directed struct, ***line*** is a simple struct. For some reasons I do not fully understand, the "struct *flist" definition must go before the "begin" line.
It is recommended to always close the list-structured variable with `flist = ""` to avoid that the associated file remain open by mistake (it will close if the end-of-file is encountered by fscan, but sometimes an "if" jump might prevent fscan from reaching the file end).

More detailed information on how to write scripts, organize them in packages, write help files, etc. may be found in the Introductory User's Guide to IRAF scripts

IRAF Paso a Paso: 8 - Manuals and References

This is a list of the most useful User's Guides and Manuals about IRAF, in either (compressed) Postscript or PDF format. For further info or documents on IRAF and STSDAS, please consult the Iraf Documentation Archive and the STSDAS Documentation page. Also visit http://iraf.net/, the new Iraf Web site, and in particular the very interesting users forum which replaces the old email-based Iraf help desk.

Basic Guides	
IRAF Beginner's Guide	A Beginner's Guide to Using IRAF version 2.10 - by J. Barnes, Aug 1993 Useful introduction to IRAF, with a general description of the IRAF software, capabilities, etc. It does not deal with how to use IRAF to reduce and analyse astronomical data.
STSDAS User's Guides	Guides to the STSDAS package (and reduction of HST data)
User's Guide to CCD Reduction	A User's Guide to CCD Reduction with IRAF - by P. Massey, Feb 1997 A general overview and description of how to reduce CCD data. More specialized cookbooks exist for more specific areas (aperture photometry, spectroscopic data, etc.)

Guide to IRAF Scripts	An Introductory User's Guide to IRAF Scripts - by E. Anderson, R. Seaman A good document on how to write simple terminal scripts as well as more structured and comples CL scripts. Might be discouraging though after the first reading, but do not give up.
General IRAF User's Guides and Cookbooks	
User's Introduction to IRAF CL	A User's Introduction to the IRAF Command Language v2.3 - by P.MB. Shames, D. Tody - Aug 1986
IRAF's Quick Reference Card	Quick Reference Card for IRAF
Synphot User's Guide	Guide to the STSDAS Synphot package - Updated Nov 2005
C Virtual Operating System	CVOS User's Guide and Reference Manual -- Dec 1999
Cleaning Images	Cleaning Images of Bad Pixels and Cosmic Rays Using IRAF - by L.A. Wells, D.J. Bell, Sep 1994
APPHOT User's Guide	A User's Guide to the Aperture Photometry Package - by L. Davis, May 1989
Rectifying and Registering Images	Rectifying and Registering Images Using IRAF - by L. Wells, Apr 1994
Guide to Slit Spectra Reduction	A User's Guide to Reducing Slit Spectra with IRAF - by P. Massey, F. Valdes, J. Barnes, Apr 1992
User's Guide to Echelle Spectra Reduction	A User's Guide to Reducing Echelle Spectra with IRAF - by D. Willmarth, J. Barnes - May 1994
Radial Velocity with Identify	Radial Velocity Measurements with IDENTIFY - by F. Valdes, Aug 1990
IGI Reference Manual	The Interactive Graphic Interpreter in STSDAS - 3rd edition, Sep 1998
IGI's Quick Reference Card	Quick Reference Card for IGI
User's Guides to the new FITS format	
FITS Kernel User's Guide	User's Guide to the new IRAF FITS Kernel used in IRAF v2.11 - by N. Zarate, Aug 97
STScI Guide to the new FITS kernel	STScI User's Guide to the IRAF FITS kernel: the new format used for NICMOS and STIS - by P. Greenfield, Aug 97
IRAF/DAOPHOT User's Guides and Cookbooks	
Guide to IRAF Stellar Photometry	A User's Guide to Stellar CCD Photometry with IRAF - by P. Massey, L.E. Davis, Apr 1992 - latest version
Daophot's Developments	Future Developments of the Daophot crowded-field photometry package - by P.B. Stetson, L.E. Davis, D.R. Crabtree
IRAF/Daophot's Reference Guide	A Reference Guide to the IRAF/DAOPHOT Package - by L.E. Davis, Jan 1994
Guides to the Mosaic Reduction Package	
The NOAO Mosaic Data Handling System	A description of the "NOAO Mosaic Data Handling System" - by D. Tody and F.G. Valdes - 1998
IRAF Help page for the mscguide task or topic	The IRAF help page for the package: MSCRED - by F. Valdes, v2.0, Sep 98
Programming in IRAF	
CL Programmer's manual	CL Programmer's manual - by E. Downey, D. Tody, G.H. Jacoby, Sep 1983 - out of date, but still useful
Guide to SPP Programming	An Introductory User's Guide to IRAF SPP Programming - by R. Seaman, Oct 1992
SPP Reference Manual	Reference Manual for the Subsect PreProcessor Language (the IRAF Programming Language)
User's Guide to IMFORT	A User's Guide to Fortran Programming in IRAF. The IMFORT Interface - by D. Tody, Sep 1986
Update to IMFORT	Update to the IMFORT Interface - by D. Tody, Jun 1989
Site Manager's Manuals	
IRAF Standards and Conventions	IRAF Standards and Conventions - by E. Downey etal, Aug 1983
PC-IRAF V2.12 Installation Guide	PC-IRAF V2.12 Installation Guide - by D. Tody and M. Fitzpatrick, May 2002
Unix/IRAF v2.12 Site Manager's Guide	Unix/IRAF v2.12 Site Manager's Guide - by D. Tody and M. Fitzpatrick, May 2002.

Site Manager's Quick Reference Card	Quick Reference Card for the Site Manager (version for Linux)
Test for IRAF v2.11	Preliminary Test Procedure for IRAF. Version 2.11 - by J. Barnes, Sep 1997
STSDAS Installation Guide	STSDAS Site Manager's Installation Guide and Reference - version 2.2, Aug 2000
Guides to Specific Packages and Tasks	
Guide to DOECSLIT	Guide to the Slit Spectra Reduction Task DOECSLIT - by F. Valdes - Apr 1992
Guide to DOSLIT	Guide to the Slit Spectra Reduction Task DOSLIT - by F. Valdes - Feb 1993 - latest version
Guide to SPECPROC	Guide to the CTIO Slit Spectra Reduction Task SPECPROC - by F. Valdes, Sep 1990
MEM Package in IRAF	MEM Pakage for Image Restoration in IRAF - by N. Wu, Oct 1992
Guide to DOFIBERS	Guide to the Multifiber Reduction Task DOFIBERS - by F. Valdes, Jul 1995
Guide to DOHYDRA	Guide to the HYDRA Reduction Task DOHYDRA - by F. Valdes, Jul 1995
Guide to DONESSIE	Guide to the NESSIE Reduction Task DONESSIE - by F. Valdes, sep 1990
Guide to DOARGUS	Guide to the ARGUS Reduction Task DOARGUS - by F. Valdes, apr 1992
Package Specifications, Reference and Revisions	
Photometry Using IRAF	Photometry Using IRAF - by L.A. Wells, Feb 1994
Specifications for APPHOT	Specifications for the Aperture Photometry Package - by L. Davis, Oct 1987
The IRAF CCD reduction Package	The IRAF CCD Reduction Package - by F. Valdes, Sep 1987
Overview of DTOI	Overview of the DTOI Package - by S.H. Jacoby - Jul 1988
IRAF Spectroscopy Packages	The IRAF Spectroscopy Reduction Packages and Tasks - by F. Valdes
IMRED Package Revision	IMRED PAckage Revision Summary: IRAF Version 2.10 - by F. Valdes, Sep 1990
Reduction of Long-slit Spectra	Reduction of Long-slit Spectra with IRAF - by F. Valdes, Mar 1986
Some Notes on ONEDSPEC	Some Notes on the ONEDSPEC Package - by G. Jacoby, Jun 1985
ONEDSPEC Revisions	ONEDSPEC Package Revisions Summary: Iraf Version 2.10 - by F. Valdes, July 1990
Technical Papers	
Optimal Extraction	A Quantitative Study of Optimal Extraction - by F. Valdes, July 1990
PSF Measuring Tasks	Psfmeasure/Starfocus: IRAF PSF Measuring Tasks - by F. Valdes
The Spectral WCS	The IRAF/NOAO Spectral World Coordinate Systems - by F. Valdes
IRAF Newsletters	
IRAF Newsletter n.13	IRAF Newsletter issue n. 13 - Dec 1994
IRAF Newsletter n.14	IRAF Newsletter issue n. 14 - May 1998
Older Newsletters (1-12)	IRAF Newsletter issue n. 1 (Jun 86) to n. 12 (Jul 92)