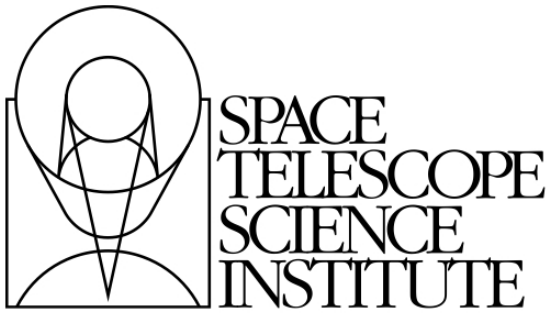
Version 4.3
June 2015

Using IDL for astronomical data analysis

This is an unofficial document for internal RIAB training purposes only



Space Telescope Science Institute
3700 San Martin Drive
Baltimore, Maryland 21218

Operated by the Association of Universities for Research in Astronomy, Inc., for the National Aeronautics and Space Administration

Revision History

Version	Date	Editor and Contributors
4.3	June 2015	Varun BAJAJ
4.2	July 2014	David BORNCAMP
4.1	June 2012	Leonardo UBEDA, Andrew COLSON
4.0	January 2012	Leonardo UBEDA
3.0	October 2009	Azalee BOSTROEM
2.0	July 2009	Abhijith RAJAN
1.0	September 2008	Pey-Lian LIM

Send comments or corrections to:
Space Telescope Science Institute
3700 San Martin Drive
Baltimore, Maryland 21218
e-mail: lubeda@stsci.edu

Contents

1

Interactive Data Language

1.1 Introduction

IDL stands for "Interactive Data Language." It is a complete computer language geared toward the interactive analysis and visualization of scientific and engineering data. It is used widely in astronomy, as well as many other fields, such as medical imaging, etc.

The intention of this tutorial is to provide the IDL basics to understand how to perform simple astronomical data analysis. We assume no previous knowledge of IDL, but at least a simple understanding of any other computer language.

IDL uses a command line interface and is fairly robust. There is also a very useful GUI called IDLDE which allows the user to edit, compile, and run the codes in the same environment, among other things.

This tutorial is organized as a set of examples with short explanations. In Chapter 1 we describe the software that you will need. In Chapter 2 we cover the simple commands that are building blocks of larger programs. In Chapter 3 we introduce the basics of plotting with IDL. In Chapter 4 we describe a procedure to obtain simple aperture photometry. Chapter 5 is an introduction to datacubes analysis. Chapter 6 describes fitting routines. In Chapter 7 you will find links to additional IDL resources.

The most effective way for you to go through this tutorial is by running IDL and trying out the commands and programs as you read the tutorial.

By changing the commands and experimenting on your own you will find ways of creating codes that perform the tasks that you need. All of the examples are adapted towards astronomy so you will find useful tools used by astronomers in everyday life.

1.2 What software do I need ?

1.2.1 IDL

You should have at least IDL version 8.0 installed in your computer. To verify which version you have, open a UNIX terminal and type "idl" at the prompt.

```
> idl
IDL Version 8.1, Mac OS X (darwin x86_64 m64). (c) 2011, ITT Visual Information Solutions
Installation number: 20....
Licensed for use by: Space Telescope Science Institute
IDL>
```

My Mac has version 8.1 installed.

1.2.2 The IDL Astronomy User's Library

Several astronomy related codes that we will use in this tutorial are from the IDL Astronomy User's Library maintained at Goddard Space Flight Center.

If the astron package is not in your computer you need to download it from

http://idlastro.gsfc.nasa.gov/

The package is usually under /itt/local/astron/. It will be useful to have the latest version installed.

1.2.3 Coyote Library

This IDL library is up-to-date and very well maintained by independent IDL consultant David Fanning. You may want to download the codes from

http://www.idlcoyote.com/

Keep in mind that this library is constantly being updated and that this could result in errors when running the codes in this document. It is therefore strongly recommended that you use the provided library while performing this training.

1.3 Disclaimer

The intention of this document is to present basic tools for astronomical data analysis using IDL. It is not supposed to be a comprehensive study. There may be different and more efficient ways of achieving the same results using IDL. If you find errors or know of better/faster ways of writing the codes and would like to contribute, please send an email to Leonardo (lubeda@stsci.edu).

2

Basic IDL concepts

2.1 Introduction

In this chapter we discuss the basic building blocks of any IDL code. We introduce each command with its definition and we also give simple examples. In order to learn the new concepts, simply run the commands at the command line (IDL>) or create your own procedure. Try to understand the output and what each command does before moving on. The exercises give you the opportunity to test your knowledge. As we progress the exercises become more difficult and challenging.

2.2 Defining procedures

IDL has 3 levels of scripts that are used for different things, `@` scripts, `.run` scripts and full procedures/functions.

2.2.1 `@` scripts

`@` scripts are usually small, very simple scripts. They are called `@` scripts because they are run using the `@` command on the command line

IDL> @script_name

`@` scripts are defined by not having a starting statement (like pro or func) and no end statement. These scripts are useful for things that are quickly coded and used in a hurry. When this program terminates, garbage collection is not run on any variables defined in the script so all variables to be "active" at the end of the program and can be played with after program termination.

2.2.2 `.run` scripts

`.run` scripts are usually much more complex then `@` scripts, containing everything that a normal procedure/function would have. They are called .run scripts because they are called by using the executive command `.run` on the command line

IDL> `.run` script_name

These scripts are defined as having an ending statement but no starting statement. As with `@` scripts, this level does not allow garbage collection to run at program termination. This also allows all of the variables in the program to remain active in memory, which is like having the command line in a script. These scripts are great for starting a program that will need to be debugged but still have complexity to them. `.run` scripts will also use active variables that are already initialized outside of the program and use them if not redefined in the script, this something to watch out for when writing this level of script and needs to be kept in mind when starting.

2.2.3 IDL procedures / functions

Programs and functions are more or less self contained. They will only use the inputs that are explicitly given to it on the command line instead of using any variable that is defined outside the program. They are defined with both starting and ending statements and are compiled using the executive command `.run` and run using the name on the command line as seen in the examples here. When programs or functions finish, garbage collection is run to flush all variables and leave things clean on the command line. The level of full programs/function is usually for mature programs that are unlikely to crash or do not need to be debugged.

Procedures are IDL codes that are stored in text files with the extension PRO; the first non–commented line of a procedure should be pro procedure_name and the last line must be end.

So here is what a procedure looks like:

```
pro procedure_name
some IDL commands
end
```

Once you are done writing your code, you may execute it from the command line using:

```
IDL> .compile myproc.pro
IDL> myproc
```

or from IDLDE by clicking on the compile and run buttons.

You may edit your procedures using your favorite ASCII editor. There is no need to worry about indentation. IDL is case insensitive. Variable declarations are almost always not necessary. The semicolon (;) is the symbol used to comment lines. The dollar sign ($) is used to tell IDL that the following line of text is actually part of the same line of code.

If the code you are writing is for yourself, then there is no need to make any explanations. However, if you plan to share the code with someone else, it is a good habit to document the code at the top of the file. Good documentation will enable the other programmer to understand the script without having to ask you. This is an example of how to document and IDL procedure:

```
; NAME:                gravitation
; DESCRIPTION:         given two point sources of known mass and distance, gravitation calculates
;                      the gravitational force using the formula from the book
;                      "Philisophiae Naturalis Principia Mathematica."
; PARAMETERS:          m1 input required (the mass of object 1)
;                      m2 input required (the mass of object 2)
;                      d input required (the distance between object 1 and object 2)
; EXAMPLE:
;                      IDL> m1 = 1.0 ; in solar mass
;                      IDL> m2 = 0.001 ; in solar mass
;                      IDL> d = 1.0 ; in astronomical units
;                      IDL> gravitation, m1, m2, d
; AUTHOR:
;                      Isaac Newton
;                      The University of Cambridge, UK
; REVISION HISTORY:
;                      Written, 5 July 1687
;                      Fixed a small problem with distance, 16 November 1688
;                      Added plotting capabilities, 14 June 2011
;
```

2.2.4 Examples of `@` scripts, `.run` scripts, and procedures

To illustrate these different levels of scripting, we need to write some small scripts/programs. To start we will create a function that will return the right ascension (RA) of an image by reading the value from the header. The specific commands that we use are explained later in this chapter.

The function will require the filename as input and will look like this:

```
function a_return, filename                          ;starting statement
hdr = headfits(filename)                             ;get the header of the image
ra = sxpar(hdr,'RA_TARG')                            get the RA of the target
return, ra                                           ;return the RA
end                                                  ;ending statement
```

We can call this on the command line by using:

IDL> test=ra_return('j8hma5ndq_flt.fits')

and IDL will return the RA of the image to be:

IDL> print, test
5.6550000

Say we wanted the RA of several images, but wanted the information to still be available on the command line after we get it so that we can see what the output is like and know what to expect for our next segment of code. This is perfect for a `.run` script. So we can write a script that looks like this:

```
                                                                    ; no starting statement
files = file_search('j8ny01t*flc.fits',count=numfiles)                          ;get the files
print, files                                                                    ;look at the files found
ra_arr = dblarr(numfiles)                                                       ;create an array for RA to live
for i=0,numfiles-1 do begin
ra=ra_return(files[i])                                                          ;get the RA using the function we just wrote
ra_arr[i]=ra                                                                    ;populate array
print,i                                                                         ;make sure the for loop is working
endfor
end                                                                             ;ending statement
```

and we save it to a text file named ra_return_test.pro. To run this, we use the statement `.run` script_name and IDL returns this:

```
IDL> .run ra_return_test
% Compiled module: $MAIN$.
j8ny01t5q_flc.fits j8ny01t9q_flc.fits j8ny01tfq_flc.fits j8ny01ttq_flc.fits
0
1
2
3
```

However, since this was a `.run` script the variables are still in memory so we can see them when running on command line. Here we have a look at some of the variables:

```
IDL> help, ra, ra_arr, i, files
RA DOUBLE = 290.25000
RA_ARR DOUBLE = Array[4]
I INT = 4
FILES STRING = Array[4]
```

If we wanted to just use the ra_return function to print the RA of specific images we could use an `@` script. The script would look something like this:

```
                                                                    ;no starting statement
print,'ra 1 is: ',ra_return('j8ny01t5q_flc.fits')                               ;print the RA
print,'ra 2 is: ',ra_return('j8ny01t9q_flc.fits')
print,'ra 3 is: ',ra_return('j8ny01tfq_flc.fits')
print,'ra 4 is: ',ra_return('j8ny01ttq_flc.fits')                               ;no ending statement
```

To use it we would use `@`script_name.pro and IDL would return:

```
IDL> @ra_return_attest.pro
ra 1 is: 290.25000
ra 2 is: 290.25000
ra 3 is: 290.25000
ra 4 is: 290.25000
```

2.3 PRINT procedure

USE

The PRINT procedure produces formatted output.

EXAMPLES

```
IDL> a = 3
IDL> print, a
IDL> b = 2
IDL> print, a + b

IDL> print, a / b
IDL> b = 2.
IDL> print, a / b
```

What is the difference between the previous examples ?

```
IDL> print, "hi"
IDL> print, "hi" + " there"

IDL> a = [1, 2, 4, 8, 12, 24, 48]
IDL> print, a
IDL> print, a[3]
IDL> print, a[3:5]
IDL> print, a[3:*]
```

SEE ALSO

Another useful procedure is MESSAGE.

2.4 FINDGEN function

USE

The FINDGEN function creates a floating-point array of the specified dimensions. Each element of the array is set to the value of its one-dimensional subscript.

EXAMPLES

```
IDL> print, findgen(3)
IDL> print, findgen(4) * 0.1
IDL> print, 3.0 + findgen(4) * 0.01
```

SEE ALSO

Other useful similar functions are INDGEN, DINGEN

EXERCISES

Exercise 2.1:
Create the sequence 0.1, 0.2, 0.3, ... 1.4 using FINDGEN
Exercise 2.2:
Create the sequence -3.2, -3.0, -2.8, ... -1.0 using FINDGEN

2.5 FLTARR function

USE

The FLTARR function creates a floating-point vector or array of the specified dimensions.

EXAMPLES

```
IDL> a = fltarr(4,4)
IDL> print, a
```

We can assign values using

```
IDL> a[0,0] = 10
IDL> a[2,1] = 12
IDL> print, a
```

SEE ALSO

Other useful similar functions are STRARR, DBLARR

EXERCISES

Exercise 2.3:
Create a 2×3 *array with 10 in all places.*
Exercise 2.4:
Create two 3×3 *different arrays (A and B).*
*Calculate A+B, A/B, A*B*
Exercise 2.5:
Using array A from the previous exercise, investigate how to obtain its inverse. Verify that $A \times A^{-1} = I$

2.6 RANDOMN function

USE

The RANDOMN function returns one or more normally-distributed, floating-point, pseudo-random numbers with a mean of zero and a standard deviation of one.

EXAMPLES

```
IDL> seed = 1.
IDL> print, randomn(seed)
IDL> print, randomn(seed, 10)
```

2.7 WHERE function

USE

The WHERE function returns a vector that contains the one-dimensional subscripts of the nonzero elements of an array.

EXAMPLES

```
IDL> a = findgen(11)
IDL> print, where(a ge 8.)
IDL> print, where(a ge 12.)

IDL> a = [1,2,3,1,2,1,1,1,1,4]
IDL> b = where(a eq 1, count)
IDL> print, b
IDL> print, count
IDL> print, a[b]
```

SEE ALSO

Other useful functions are UNIQ, ARRAY_INDICES

EXERCISES

Exercise 2.6:
Create a random real 1000 element array. Use WHERE *to create another array with those elements with counts lower than 0.5 and higher that* –0.5.

2.8 HELP procedure

USE

The HELP procedure gives the user information on many aspects of the current IDL session. The specific area for which help is desired is selected by specifying the appropriate keyword.

EXAMPLES

```
IDL> a = findgen(11)
IDL> help, a
IDL> b = dindgen(11)
IDL> help, b
IDL> c = "hello"
IDL> help, c
```

2.9 N_ELEMENTS function

USE

The N_ELEMENTS function returns the number of elements contained in an expression or variable.

EXAMPLES

```
IDL> a = indgen(11)
IDL> print, n_elements(a)

IDL> b = where(a lt 5, count)
IDL> print, n_elements(b)
IDL> print, count
```

2.10 SIZE function

USE

The SIZE function returns size and type information for its argument.

EXAMPLES

IDL> a = findgen(3,4)
IDL> print, a
IDL> help, a
IDL> print, size(a)

This IDL output indicates that the array has 2 dimensions, equal to 3 and 4, a type code of 4 (FLOAT), and 12 elements total.

SEE ALSO

Other useful similar functions is N_ELEMENTS

2.11 STRMID function

USE

The STRMID function extracts a substring from a string expression.

EXAMPLES

IDL> str = "astronomy"
IDL> print, strmid(str, 2, 3)

SEE ALSO

Other useful functions are STRLEN, STRPOS, STRTRIM

EXERCISES

Exercise 2.7:
Given the string "j12345678_flt.fits", *use* STRMID *to extract the rootname of the file.*

2.12 SORT function

USE

The SORT function returns a vector of subscripts that allow access to the elements of an array in ascending order.

EXAMPLES

IDL> a = [4, 3, 7, 1, 2]
IDL> print, sort(a)
IDL> print, a[sort(a)]

SEE ALSO

Other useful functions are REVERSE, WHERE

EXERCISES

Exercise 2.8:
Use the command SORT *to create the array* b = [7, 4, 3, 2, 1] *from the array* a *in the example.*

2.13 MIN and MAX functions

USE

The MIN and MAX functions return the value of the smallest and largest element of an array.

EXAMPLES

IDL> a = [4.4, 4.5, 4.5, 4.5, 4.7, 4.3, 4.0, 4.1, 4.5, 4.5, 3.9, 4.5, 4.5, 4.6]
IDL> print, max(a)
IDL> print, min(a)

SEE ALSO

Other useful functions are MOMENT, MEAN, STDDEV, VARIANCE, TOTAL

2.14 MOMENT function

USE

The MOMENT function computes the mean, variance, skewness, and kurtosis of a sample population contained in an n-element vector.

EXAMPLES

IDL> a = [4.4, 4.5, 4.5, 4.5, 4.7, 4.3, 4.0, 4.1, 4.5, 4.5, 3.9, 4.5, 4.5, 4.6]
IDL> result = moment(a)

IDL> print, "mean: " , result[0]
IDL> print, "variance: ", result[1]
IDL> print, "skewness: ", result[2]
IDL> print, "kurtosis: ", result[3]

SEE ALSO

Other useful similar functions are MAX, MIN, MEAN, STDDEV, VARIANCE

EXERCISES

Exercise 2.9:
Given the array a = randomn(10,1000). What are the moments of the array a?

2.15 READCOL **procedure**

USE

The READCOL procedure reads a free-format ASCII file with columns of data into IDL vectors.

EXAMPLES

EXAMPLE 1: Reading a simple text file

Use the supplied file f336w.mag. Place the file in a folder of your choice and read the content with the command:

```
path = "/Users/your path here"
readcol, path + "f336w.mag", xcenter, ycenter, mag, smag
print, xcenter
```

We assume that all four columns are floating point.

EXAMPLE 2: Reading a simple file with multiple column formats

Use the provided file sources.dat. Place the file in a folder of your choice and read the content with the command:

```
path = "/Users/your path here"
fmt = "A, F, F"
readcol, path + "sources.dat", name, ra, dec, format = fmt
print, name
help, name
print, ra
```

The first column contains a string and the other two are floating point numbers.

SEE ALSO

Another useful function is TRANSREAD

EXERCISES

Exercise 2.10:
Create your own input file with three columns and at least four rows. The columns should have an astronomical meaning. Use any editor of your choice. Develop a code that uses READCOL *to import the columns in your file into IDL vectors.*

2.16 FORPRINT **procedure**

USE

The FORPRINT procedure writes a file with columns of data from IDL vectors.

EXAMPLES

In a procedure include the following lines

```
name = ["star1", "star2", "star3"]
xcoo = [150.0, 244.3, 344.9]
ycoo = [422.1, 561.4, 554.8]
path = "/Users/your path here"
forprint, name, xcoo, ycoo, textout = path + "output.dat", format = "(A12, F10.1, E10.3)"
```

Run the code. What does the output look like ? Open it with your favorite editor.

SEE ALSO

Other useful similar functions is PRINTF.

EXERCISES

Exercise 2.11:
Use FORPRINT *to write a file with the columns from the previous exercise but inverting the column order.*

2.17 SPAWN procedure

USE

The SPAWN procedure spawns a child process to execute a unix command.

EXAMPLES

EXAMPLE 1: Some useful UNIX commands

In your current directory, type:

IDL> spawn, "pwd"
IDL> spawn, "ls"
IDL> spawn, "mkdir mynewfolder"

EXAMPLE 2: Running applications from IDL

You can execute other applications from within IDL. In order to launch DS9, simply type

IDL> spawn, "ds9 &"

We can also display images from IDL. The command

IDL> spawn, "ds9 -frame 3 fits m101_blue.fits &"

will display the FITS file m101_blue.fits in a new DS9 window in frame 3. You may need to type the whole path of DS9 depending on your configuration:

spawn, "/Applications/ST_sci/bin/ds9 -frame 3 -fits m101_blue.fits"

EXERCISES

Exercise 2.12:
Use SPAWN *in an* IDL *script that opens* DS9 *and loads 2 FITS files in different frames.*

Exercise 2.13:
Open all six extensions of file j91c12biq_flt.fits *using the -multiframe flag from within IDL. Can you make the images blink ?*
Hint: For help on DS9 *commands, go to: http://hea-www.harvard.edu/RD/ds9/ref/index.html*

2.18 MRDFITS **procedure**

USE

The MRDFITS procedure reads all standard FITS data types into arrays or structures.

EXAMPLES

EXAMPLE 1: Reading an ACS FITS image

Place the file j91c12biq_flt.fits inside a folder. Read extension 1 (WFC2) of an ACS image using MRDFITS into the variable called "image".

```
path = "/Users/your path here"
image = mrdfits( path + "j91c12biq_flt.fits", 1, head)
help, image
print, size(image)
```

EXAMPLE 2: Reading a binary FITS table

The supplied file ckp00_41000.fits gives the Kurucz spectrum of an O star with solar metallicity and $T_{\rm eff}$ = 41000 K. Read the file using MRDFITS and check its contents:

```
path = "/Users/your path here"
spect = mrdfits(path + "ckp00_41000.fits", 1, hdr)
help, spect, /str
```

IDL reads the binary FITS table that contains 12 columns and 1221 rows and stores it as an IDL structure. The first column contains the wavelength (in units of Ångstroms) and the other columns contain the flux (in units of erg sec^{-1} cm^{-2} Å^{-1}) for several gravity values. How do I know that?

We can assign the columns to IDL variables by writting, for example:

```
x = spect.wavelength
y = spect.g45
```

SEE ALSO

Another useful similar procedure is READFITS

EXERCISES

Exercise 2.14:
What is the structure of the provided NICMOS *file* n8ws10siq_ima.fits *? (Hint: use the PyRAF command* catfits.*) Display some SCI extensions using DS9. Create an IDL code to read the first extension. What is the size (in pixels) of this image?*

2.19 MWRFITS procedure

USE

The MWRFITS procedure writes all standard FITS data types from input arrays or structures.

EXAMPLES

EXAMPLE 1: Writing a simple array to a FITS file

```
a = fltarr(20,20)
mwrfits, a, "test.fits"
```

EXAMPLE 2: Creating and saving an IDL structure

We define the components of the structure:

```
id = ["m31","m101","ngc4214"]
ra = [35.2, 45.7, 140.2]
dec = [7.0, -31.3, 55.6]
```

We name the structure target

```
target = {name:id, alpha:ra, delta:dec }
```

We save the structure to a FITS file called "catalog.fits"

```
mwrfits, target, "catalog.fits", /create
```

2.20 CGIMAGE procedure

USE

The CGIMAGE procedure allows you to display an image on the computer display or in a PostScript file in a particular position of your choice.

EXAMPLES

EXAMPLE 1: Display a FITS file onto a screen window

Place the WFPC2 file uba3010em_c0f.fits inside a folder.

We set the window environment:

```
set_plot, "x"
```

We read the image using MRDFITS into the variable called "image".

```
path = "/Users/your path here"
image = mrdfits( path + "uba3010em_c0f.fits", 0, head)
help, image
```

We scale the image:

```
data = bytscl(image[*,*,1], min = 0., max = 30.)
```

We display the WFPC2 image:
cgimage, data, /keep_aspect_ratio

EXAMPLE 2: Display a FITS file within a Postcript file

We define the PostScipt output file:
set_plot, "ps"

We define the position and size of the plot:

plot_left = 8.
plot_bottom = 15
xsize = 10.
ysize = 10.
page_height = 27.94
page_width = 21.59

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We open the output device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

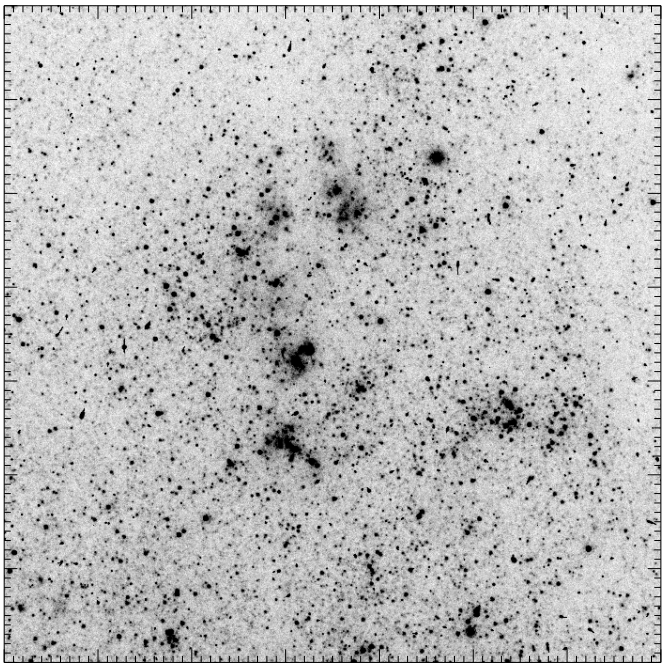


FIGURE 2.1— Output plot.

We read and scale the image
image = mrdfits(path + "uba3010em_c0f.fits", 0, head)
data = 255b − bytscl(image[*,*,3], min = 0, max = 30.)

We display the image
cgimage, data, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We close the device
device, /close

EXERCISES

Exercise 2.15:
Modify the previous code to display a 700×700 *subarray of extension 4 of the WFC3/UVIS image* ib6w71lxq_flt.fits. *(Hint: use the sub array slection* [2000:2699, 700:1399]). *The output file should look something like Figure 2.1.*

2.21 HEADFITS procedure

USE

The HEADFITS procedure reads a FITS header (primary or extension) into a string array.

EXAMPLES

Place the file j91c12biq_flt.fits inside a folder of your choice. Read the image header from extension 4 using the procedure HEADFITS into the variable called "hdr".

```
path = "/Users/your path here"
hdr = headfits(path + "j91c12biq_flt.fits", exten = 4)
help, hdr
print, hdr
```

EXERCISES

Exercise 2.16:
Read the primary header of the WFC3 image ib6w71lxq_flt.fits.

2.22 SXPAR procedure

USE

The SXPAR procedure obtains the value of a parameter in a FITS header.

EXAMPLES

Place the file j91c12biq_flt.fits inside a folder of your choice. Read the image header from extension 0 using the procedure HEADFITS into the variable called ‘hdr’.

```
path = "/Users/your path here"
header = headfits(path + "j91c12biq_flt.fits", exten
= 0)

What is the target of this image ?
print, sxpar(header, "TARGNAME")
When was this image observed ?
print, sxpar(header, "DATE-OBS")
```

EXERCISES

Exercise 2.17:
Read the primary header of the WFC3/UVIS image ib6w71lxq_flt.fits *and respond the following questions using* SXPAR*: What filter was used for this observation? Is the image bias corrected ? What is the name of the dark image used for the dark current correction ? What is the gain used for amplifier D ?*

2.23 FOR...DO statement

USE

The FOR...DO statement is used to execute one or more statements repeatedly, while incrementing or decrementing a variable with each repetition, until a condition is met.

EXAMPLES

IDL> for k = 1, 3 do print, k,k^2

IDL> for j = 100, 0, –5 do print, j

IDL> a = findgen(11)
IDL> for k = 0, n_elements(a) – 1 do print, a[k]

Within a procedure, the FOR...DO statement has the following simple structure:

```
for k = 1, 6 do begin
m = k * 3.
print, m
endfor
```

SEE ALSO

Other useful similar statements are REPEAT...UNTIL, WHILE...DO

EXERCISES

Exercise 2.18:
Create a vector with 10 values equal to the square of the index.

Exercise 2.19:
Create the following sequence using a FOR...DO *statement: 2001-01-01, 2001-02-01, 2001-03-01, 2001-04-01.*

Exercise 2.20:
Create an IDL code that loops through all four WFPC2 images in uba3010em_c0f.fits *and displays them to the screen in order. Use* CGIMAGE.

Exercise 2.21:
Using MAST, download the calibrated association product ib3p11010. *You should obtain the following images:* ib3p11p7q, ib3p11p8q, ib3p11phq, *and* ib3p11q9q. *Create an IDL code that loops through all four headers and prints the following information to the screen: root name, target, filter, exposure time, and date of observation.*

2.24 WHILE...DO statement

USE

The WHILE...DO loops are used to execute a statement repeatedly while a condition remains true.

EXAMPLES

Within a procedure, the WHILE...DO statement has the following simple structure:

```
k = 10.
while k lt 15. do begin
k = k + 1.5
print, k
endwhile
```

SEE ALSO

Other useful similar statements are REPEAT...UNTIL, FOR...DO

2.25 IF...THEN...ELSE statement

USE

The IF statement is used to conditionally execute a statement or a block of statements.

EXAMPLES

```
IDL> a = 3
IDL> if a eq 2 then print, "a is two" else print, "a is not two"
```

Within a procedure, the IF...THEN...ELSE statement has the following simple structure:

```
target = ["ngc140", "ngc4214", "m51" ]
for j = 0, n_elements(target) - 1 do begin
if target[j] eq "m51" then begin
print, "processing M51"
endif else begin
print, "processing another target: ", target[j]
endelse
endfor
```

SEE ALSO

Other useful statement is CASE

2.26 CASE statement

USE

The CASE statement selects one, and only one, statement for execution, depending on the value of an expression.

EXAMPLES

```
x = 2
case x of
1: print, "one"
2: print, "two"
3: print, "three"
4: print, "four"
endcase
```

SEE ALSO

Other useful similar functions is IF...THEN

EXERCISES

Exercise 2.22:
Given the string array root = ["j12345678","u12345678","n12345678","i12345678","l12345678"], *write an IDL code that identifies the HST camera according to the rootname. Use the* CASE *statement.*

2.27 Defining functions

USE

A function is a program unit containing one or more IDL statements that returns a value. This unit executes independently of its caller. It has its own local variables and execution environment.

EXAMPLES

We could define a function (called "avg") which returns the average value of a given array using the following commands:

```
function avg, array
return, total(array) / n_elements(array)
end

pro procedure_name
print, avg([1,2,3,4])

a = [1.0, 0.0]
print, avg(a)

end
```

EXERCISES

Exercise 2.23:
Define a function that returns the area of a circle given the radius.

Exercise 2.24:
Write and IDL function that returns the slope and y-axis intercept of a line that connects two input points $P = (x_0, y_0)$ *and* $Q = (x_1, y_1)$. *Do not forget to study the vertical line case.*

3

Plotting with IDL

3.1 Introduction to plotting with IDL

IDL has been developed to produce high quality figures ready for scientific use. In this tutorial we present several examples of how to create some figures of astronomical interest.

3.2 Creating a simple plot in a screen window

The following commands will display a plot similar to Figure 3.1 in a screen window.

We set the screen environment:
set_plot, "x"

PROCEDURE	USE
WINDOW	The WINDOW procedure creates a window for the display of graphics or text. It is only necessary to use WINDOW if more than one simultaneous window or a special size window is desired because a window is created automatically the first time.
CGPLOT	The CGPLOT procedure draws graphs of vector arguments.
OPLOT	The OPLOT procedure plots vector data over a previously-drawn plot. It differs from CGPLOT only in that it does not generate a new axis. Instead, it uses the scaling established by the most recent call to CGPLOT and simply overlays a plot of the data on the existing axis.
CGCOLOR	The purpose of this function is to obtain drawing colors by name and in a device/decomposition independent way.

We create a white window to display our data:
window, xsize = 800, ysize = 800, title = "screen plot"
erase, cgcolor("white")

We define the independent variable and the function:
x = findgen(2001) * 0.1
u = exp(−x / 30.)
y = sin(x)

We define the axes:
cgplot, x, y * u, $
/noerase, /nodata, $
xrange = [0. , 50.], $
yrange = [-1.5, 1.5], $

xstyle = 1, ystyle = 1, $
color = cgcolor("black"), $
ytitle = "y title", xtitle = "x title"

We plot the function in red:
oplot, x, y*u, color = cgcolor("red")

3.3 Creating a simple plot in a PostScript/PDF file

The following commands will create a PostScript file with the plot shown in Figure 3.1.

We set the PostScript environment:
set_plot, "ps"

PROCEDURE	USE
DEVICE	Controls the graphics device currently selected.

We define the file name for the plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and plot size: units are given in centimeters.

page_height = 27.94
page_width = 21.59
plot_left = 5.
plot_bottom = 5
xsize = 14.
ysize = 10.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

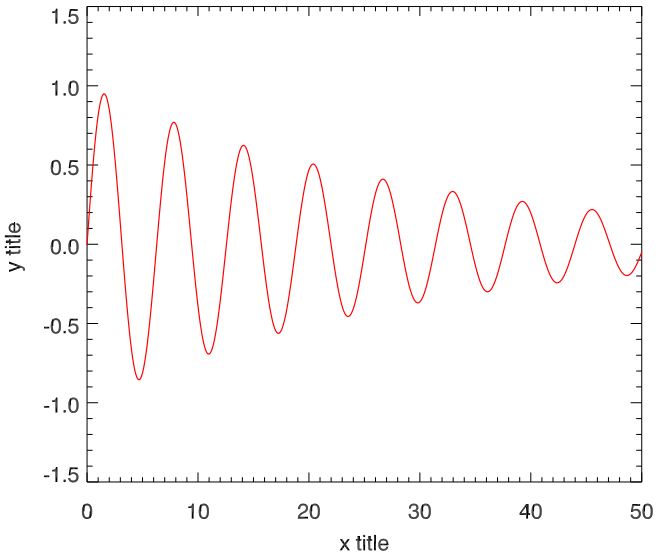


FIGURE 3.1— Simple plot in a PostScript file.

We define the independent variable and the function:
x = findgen(2001) * 0.1
u = exp(-x / 30.)
y = sin(x)

We define the axes:
cgplot, x, y * u, $
/noerase, /nodata, $
xrange = [0. , 50.], $
yrange = [-1.5, 1.5], $
xstyle = 1, ystyle = 1, $
color = cgcolor("black"), $
xtitle = "x title", ytitle = "y title" , $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We plot the function in red:
oplot, x, y * u, color = cgcolor("red")

We close the device:
device,/close

If instead of a PostScript file you prefer a PDF file, you may generate it automatically using the command

cgps2pdf, psname

EXERCISES
Exercise 3.1:
Plot the function $y = f(x) = exp(x) \cdot x^{-2}$ *and create a PDF file with it. Select the appropiate range in x and y.*

3.4 Plotting side by side figures

The following commands will create a PostScript file with two plots one beside the other as shown in Figure 3.2.

We set the PostScript environment:
set_plot, "ps"

We define the file name:
path = "/Users/your path here"

psname = path + "plot.ps"

We define the page size:
page_height = 27.94
page_width = 21.59

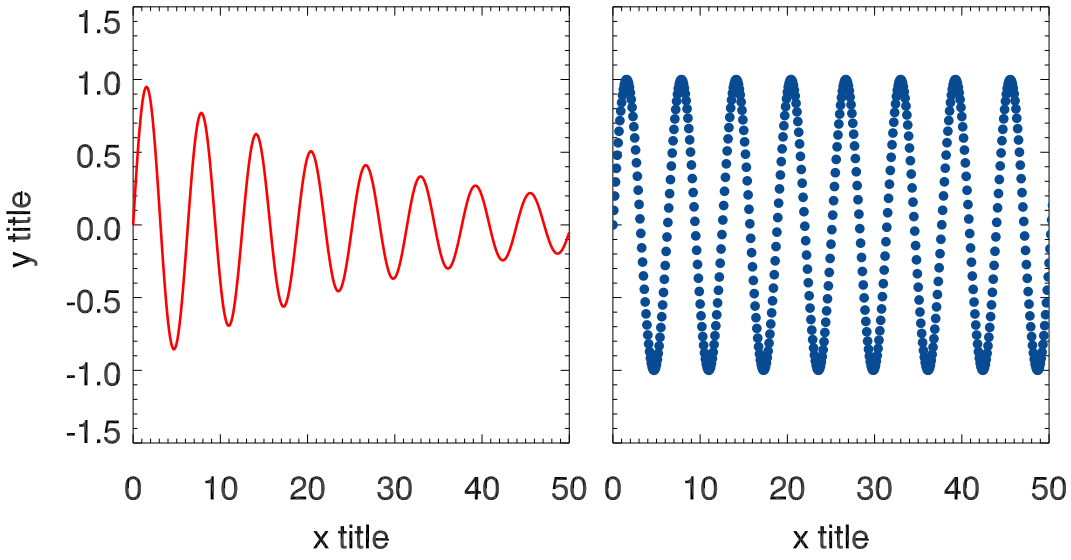


FIGURE 3.2— Side by side figures.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

PROCEDURE	USE
SYMCAT	Provides a symbol catalog for specifying a variety of plotting symbols.

We define the size and position of the left plot:
plot_left = 5.

```
plot_bottom = 5
xsize = 5.
ysize = 5.


We define the independent variable and the function for the left plot:
x = findgen(2001) * 0.1
y = sin(x)
u = exp(-x / 30.)

We define the axes:
cgplot, x, y * u, $
/noerase, /nodata, $
xrange = [0. , 50.], $
yrange = [-1.5, 1.5], $
xstyle = 1, ystyle = 1, $
color = cgcolor("black"), $
xtitle = "x title", ytitle = "y title" , $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom
+ ysize) / page_height]

We plot the function in red:
oplot, x, y * u, color = cgcolor("red")

We define the size and position of the plot on the right:
plot_left = 10.5
plot_bottom = 5
xsize = 5.
ysize = 5.


We define the axes:
cgplot, x, y, $
/noerase, /nodata, $
xrange = [0. , 50.], $
yrange = [-1.5, 1.5], $
xstyle = 1, ystyle = 1, $
color = cgcolor("black"), $
xtitle = "x title" , ytitle = "y title" , $
ytickname = replicate(" ", 60), $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom
+ ysize) / page_height]

We plot the function in blue:
oplot, x, y, psym = symcat(16), symsize = 0.5, color = cgcolor("blu7")
```

We close the device:
device,/close

EXERCISES
Exercise 3.2:
Plot three mathematical functions of your choice in three contiguous plots. Use different symbols for each plot.

3.5 Plotting with error bars

The following commands will create a simple plot (Figure 3.3) with vertical and horizontal error bars in each point.

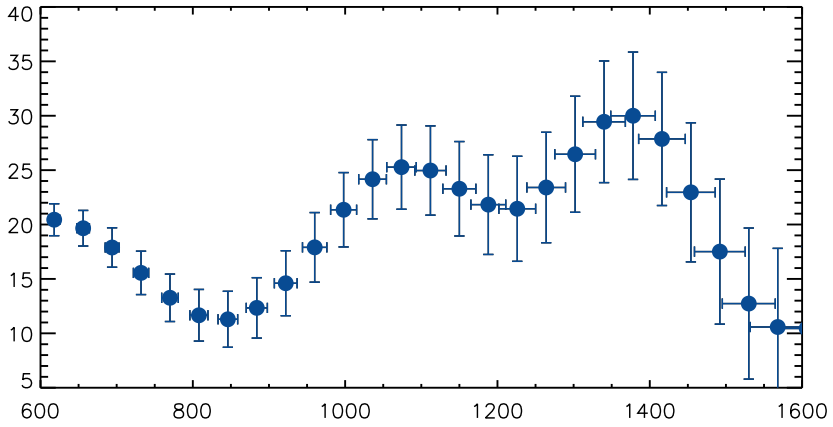


FIGURE 3.3— Error bars example.

We set the PostScript environment:
set_plot, "ps"

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and size of the plot: units are given in centimeters.
page_height = 27.94
page_width = 21.59
plot_left = 4.
plot_bottom = 5

```
xsize = 14.
ysize = 8.
```

We open the PostScript device:

```
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait
```

We read the input data from the supplied file "data.dat"

```
readcol, path + "data.dat", x, y
```

We define random errors:

```
yerr = 0.1 + findgen(101)*0.005*sqrt(x)
xerr = 5.0 * yerr
```

PROCEDURE	USE
PLOTERROR	Plots data points with accompanying X or Y error bars.

We create the plot:

```
ploterror, x, y , xerr, yerr, $
xthick = 4, $
ythick = 4 , $
xcharsize =1.2, $
ycharsize=1.2, $
thick = 5, $
/nodata, $
xrange = [600., 1600], $
yrange = [ 5, 40], $
xstyle = 1, ystyle = 1, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom
+ ysize) / page_height]

oploterror, x, y, xerr, yerr, psym = symcat(16), symsize = 1.5, color = cgcolor("blu7"), $
errcolor = cgcolor("blu7")
```

We close the PostScript file:

device, /close

3.6 Creating a simple histogram

The following commands will create a PostScript file with a histogram as shown in Figure 3.4.

We set the PostScript environment:
set_plot, "ps"

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and size of the plot:
units are given in centimeters.
page_height = 27.94
page_width = 21.59
plot_left = 3.
plot_bottom = 5
xsize = 10.
ysize = 10.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

We read the input data from the supplied file "histo.dat"
readcol, path + "histo.dat", vector

We create the histogram:
cghistoplot, vector, $
binsize = 1, $

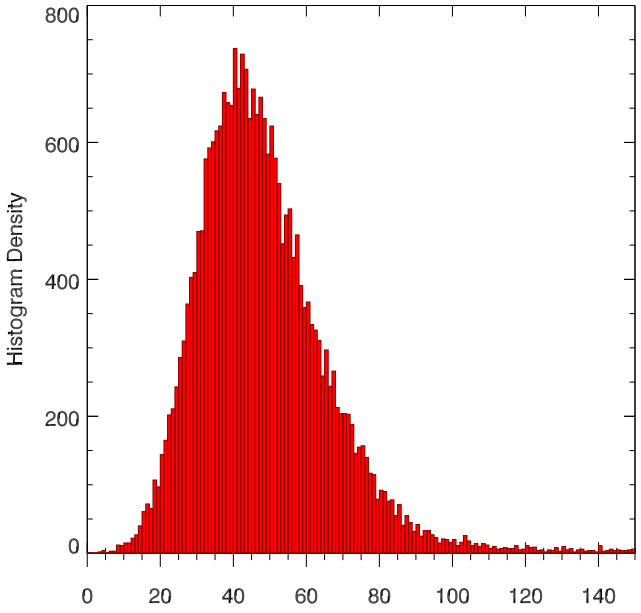


FIGURE 3.4— Simple histogram.

PROCEDURE	USE
CGHISTOPLOT	Plots a variety of histograms.

charsize = 0.5, $
xrange = [0,150.], $
yrange = [0, 800.], $
polycolor = "red", $
datacolorname = "dark red", $
/fillpolygon, $
xstyle = 9, ystyle = 1, $
xcharsize = 2.5, $
ycharsize = 2.5, $
/normal, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We close the PostScript file:
device, /close

EXERCISES
Exercise 3.3:
Read the first column in the file ngc4214_336.dat *and crate a histogram. Choose an appropiate bin size and do not fill the polygons.*

3.7 Plotting spectra

The following commands will create a PostScript file with the plot of a labeled spectrum as shown in Figure 3.5.

We set the PostScript environment:
set_plot, "ps"

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

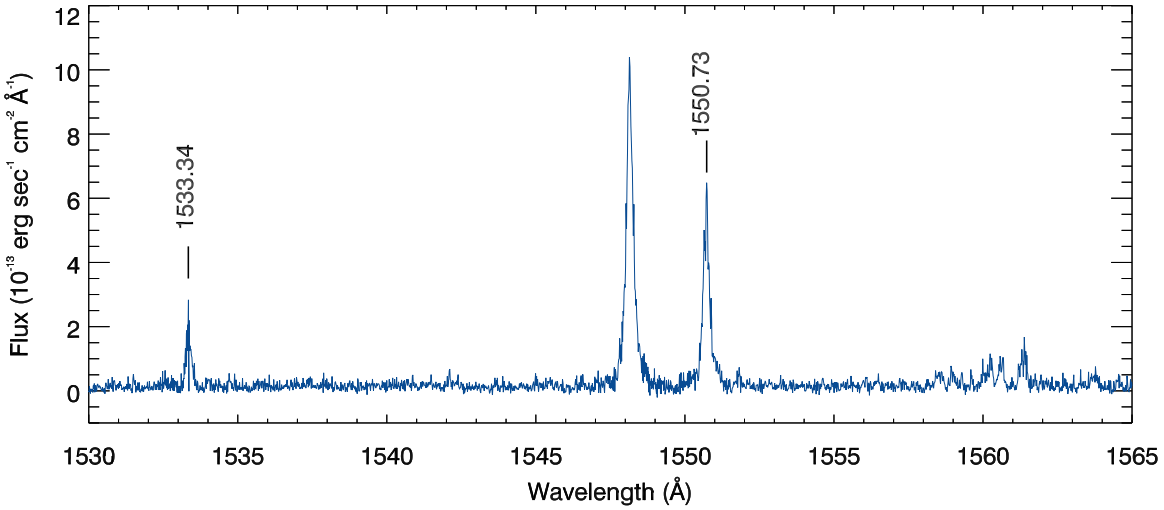


FIGURE 3.5— Scaled spectrum (flux as a function of wavelength). Two lines are marked and labeled.

We define the position and size of the plot: units are given in centimeters.
page_height = 21.59
page_width = 27.94
plot_left = 5.
plot_bottom = 5
xsize = 20.
ysize = 8.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = page_width, $
scale_factor = 1.0, $
/landscape

We read a COS spectrum
spec = mrdfits(path + "o5bn02010_x1d.fits", 1, hdr)

We assign the flux and wavelength to IDL vectors:
wave = spec.wavelength
flux = spec.flux

We sort the wavelength variable:
i = sort(wave)
swave = wave[i]
sflux = flux[i]

We scale the flux:
scaled_sflux = sflux / 1e–13

We define the axes for the plot:
cgplot, swave, scaled_sflux, $
charsize = 1.5, $
/nodata, $
yrange = [–1.0, 12], $
xrange = [1530, 1565], $
ystyle = 1, xstyle = 1, $
xtitle = "wavelength (!3" + string(197b) + ")", $
ytitle = "Flux (10!e–13!n erg sec!e–1!n cm!e–2!n !3 " + string(197b) + "!x!e–1!n)", $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We plot the spectrum in blue:
oplot, swave, scaled_sflux, color = cgcolor("blu7")

PROCEDURE	USE
XYOUTS	Draws text on the current graphics device starting at the designated coordinate.

We mark two lines:
oplot, 1533.34*[1,1], [3.5, 4.5], thick=3
xyouts, 1533.5, 5., strtrim(1533.34, 2), orientation = 90, charsize = 1.5
oplot, 1550.73*[1,1], [6.8, 7.8], thick=3
xyouts, 1550.8, 7.9, strtrim(1550.73,2), orientation = 90, charsize = 1.5

We close the device:
device, /close

3.8 Displaying a FITS image with annotations

The following commands will create a PostScript file where we display a FITS image and make some annotations as shown in Figure 3.6.

We set the PostScript environment:
set_plot, "ps"

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and size of the plot: units are given in centimeters.
page_height = 27.94
page_width = 21.59
plot_left = 5.
plot_bottom = 5
xsize = 14.
ysize = 14.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

We read the drizzled WFPC2 image:
image = mrdfits(path + "u9w10107m_drz.fits", 1, hdr)

PROCEDURE	USE
CGLOADCT	Loads one of 41 predefined IDL color tables.
BYTSCL	Scales all values of an array from one range to another range.
CGCOLORBAR	Adds a color bar to the current graphics window.

We select a sub-array and scale the image:
data = 255b – bytscl(image[100:800, 100:800], min = 1, max = 6)

We get the size of the image:
ss = size(data, /dimensions)

We load a color table:
cgloadct, 33, ncolors = 256, bottom = 0, clip = [0, 256], /reverse
tvlct, redvector, greenvector, bluevector, /get

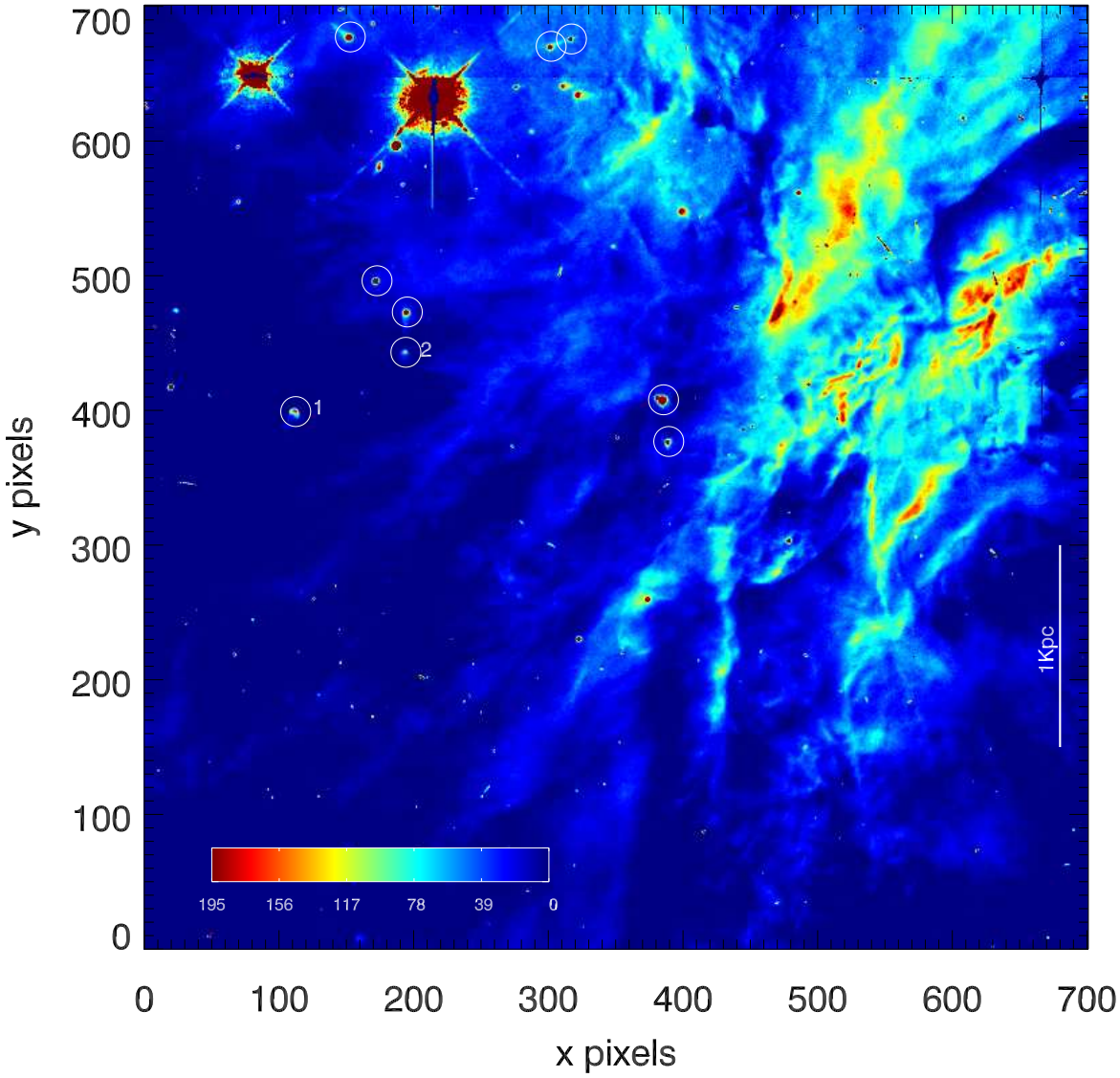


FIGURE 3.6— A drizzled WFPC2 image of Orion showing some proplyds. We inlcude some basic annotations and a colorbar to indicate the image intensity in arbitrary units.

We display the image:
cgimage, data, position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We load a color table:
cgloadct, 0
tvlct, redvector, greenvector, bluevector, /get

We draw the axes:
```
cgplot, [0], [0], $
xcharsize = 1, ycharsize = 1, $
thick = 2, $
xrange= [0, ss[0]], $
yrange= [0, ss[1]], $
xtitle = "x pixels", $
ytitle = "y pixels", $
xstyle = 1, ystyle = 1, $
/nodata, /normal, /noerase, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom
+ ysize) / page_height]
```

We define positions for several sources:
```
xc = [112.40, 194.21, 172.80, 385.57, 152.96, 302.10, 317.19, 195.28, 389.43]
yc = [ 399.14, 443.14, 496.3, 407.87, 677.33, 670.35, 675.82, 473.09, 376.93 ]
```

We draw white circles around the sources:
```
oplot, xc, yc, color = cgcolor("white"), psym = symcat(9), symsize = 2
```

We label two of the sources:
```
xyouts, 125.00, 397.0, "1", charsize = 1.0, color = cgcolor("white")
xyouts, 205.00, 440.0, "2", charsize = 1.0, color = cgcolor("white")
```

We draw the vertical line to show the image scale:
```
oplot,[680,680], [150,300], linestyle = 0, thick = 3, color = cgcolor("white")
xyouts, 675, 200, " 1Kpc", charsize = 1., color = cgcolor("white"), orientation = 90
```

We load a color table for the colorbar:
```
cgloadct, 33, ncolors = 256, bottom = 0, clip = [0, 256], /reverse
tvlct, redvector, greenvector, bluevector, /get
```

We define the position of the colorbar:
```
plot_left = 6
plot_bottom = 6
xsize = 5
ysize = 0.5
```

```
img = image[100:800, 100:800]
```

We draw the colorbar:
```
cgcolorbar, divisions = 5, range = [max(img), min(img)], xminor = 1, $
xticklen = 0.1, format = "(I3)", $
palette= [[redvector], [greenvector], [bluevector]], $
annotatecolor = "white", charsize = 0.8, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom
+ ysize) / page_height]
```

We close the device:
device, /close

3.9 Displaying a contour plot over a FITS image

The following commands will create a PostScript file where we display a FITS image as background and we over plot a contour as shown in Figure 3.7.

We set the PostScript environment:
set_plot, "ps"

We define the file name:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and plot size: units are given in centimeters.
page_height = 27.94
page_width = 21.59
plot_left = 4.
plot_bottom = 5
xsize = 14.
ysize = 14.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

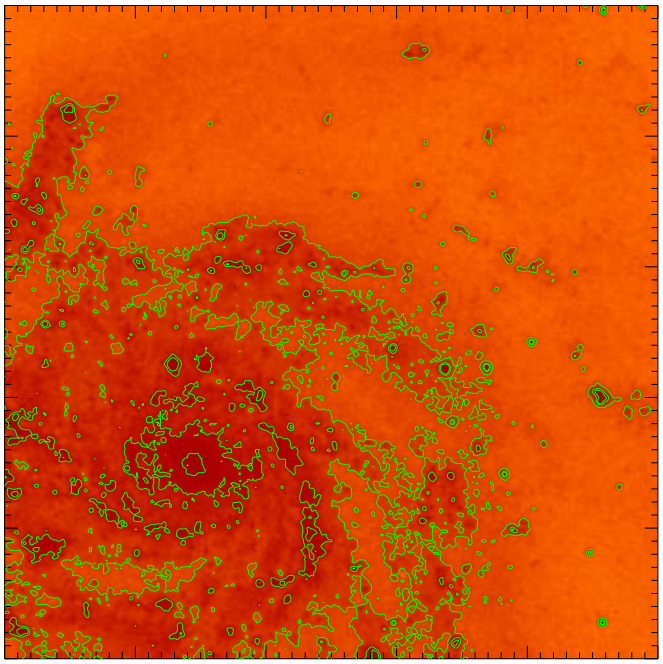


FIGURE 3.7— Contour plot over a FITS background.

We read a FITS file:
image = mrdfits(path + "m101_blue.fits", 0, hdr)

We select a sub-array and scale the image:
image = image[300:800,300:800]
data = 255b – bytscl(image, min = 0.0, max = 18000.)

We load a color table:
cgloadct, 3, ncolors = 256, bottom=0, clip = [90, 200]

tvlct,redvector, greenvector, bluevector, /get

We display the image:
cgimage, data, position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We get the size of the image:
tam = size(image, /dim)
contour_x = findgen(tam[0])
contour_y = findgen(tam[1])

PROCEDURE	USE
CGCONTOUR	Draws a contour plot from data stored in a rectangular array or from a set of unstructured points. Both line contours and filled contour plots can be created.

We create the contour plot:
cgcontour, image, contour_x, contour_y, $
levels = [8000, 12000, 14000], $
xstyle = 1, $
ystyle = 1, $
axiscolor = "black", $
label = 0, $
xtickformat = "(a1)", ytickformat = "(a1)", $
c_colors = cgcolor("green"), /noerase, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We close the device:
device, /close

EXERCISES

Exercise 3.4:
Create a plot that displays the same contour as in the example, but without the background FITS image. Play around with different intensity levels. Can you make it in different colors ?

4

Simple aperture photometry using IDL

4.1 Introduction

In this chapter we use very simple commands to perform aperture photometry of an ACS/WFC sub array. The purpose of this chapter is to show how basic aperture photometry can be achieved using IDL with procedures that were based on original IRAF codes. A real example of aperture photometry requires several corrections to the instrumental magnitudes which are usually time, position, and instrument dependent. The more general analysis is beyond the goal of this IDL tutorial.

Read through the whole chapter and then create your own procedure by copying and pasting the code lines.

4.2 Setting things up

We will perform aperture photometry on an NGC 104 image with ID j8hm01xaq_flt.fits. Place the supplied image inside a folder of your choice and define a path to the folder:

path = "/Users/your path here"

EXERCISES
Exercise 4.1:
Can you show how many extensions the FITS file has? and how many pixels per science chip?

Since the image is too big for a tutorial we will extract a 500×500 subarray from extension four, and we store the information in the variable "image500".

```
image = mrdfits(path + "j8hm01xaq_flt.fits", 4, hdr)
image500 = image[900:1399, 800:1299]
help, image500
```

EXERCISES
Exercise 4.2:
Can you tell when this image was created? Who is the PI ? In what physical units is this chip ? What is the exposure time ?

We now display the array to see the different sources.

We define the file name for the plot:
psname = path + "photometry_01.ps"

We define the position and plot size: units are given in centimeters.
page_height = 27.94

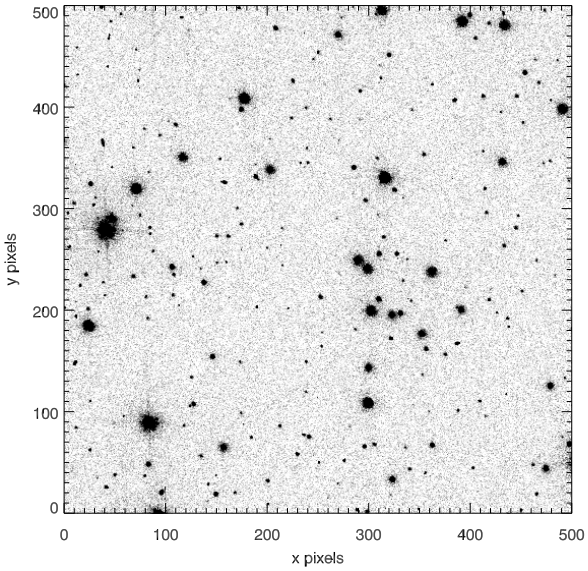


FIGURE 4.1— NGC 104 ACS/WFC subarray.

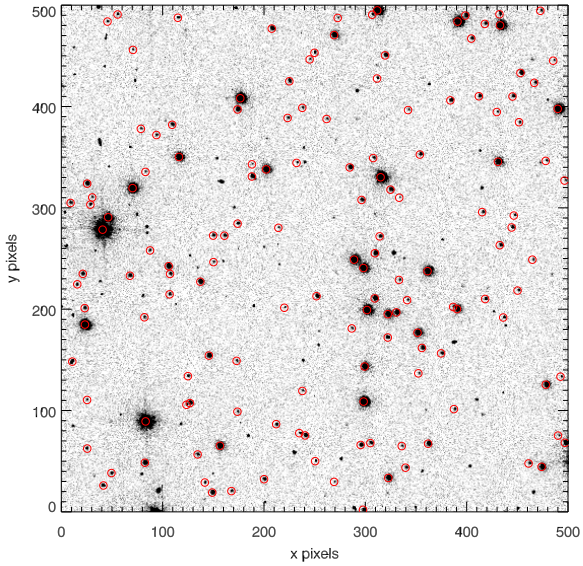


FIGURE 4.2— Star detections are marked using an open red circle.

page_width = 21.59
plot_left = 5.
plot_bottom = 5
xsize = 14.
ysize = 14.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

We scale the array to make the stars pop:
data = 255b – bytscl(image500, min = 1, max = 40)
ss = size(data, /dimensions)

We load a color table:
cgloadct, 0
tvlct, redvector, greenvector, bluevector, /get

We display the image:
cgimage, data, position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We draw the axes:
cgplot, [0], [0], $
xcharsize = 1, ycharsize = 1, $
thick = 2, $
xrange= [0, ss[0]], $
yrange= [0, ss[1]], $
xtitle = "x pixels", $
ytitle = "y pixels", $
xstyle = 1, ystyle = 1, $
/nodata, /normal, /noerase, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We close the device:
device, /close

Your plot should look like Figure 4.1. This is a not very crowded field in the proximity of 47 Tuc.

4.3 Source identification

We need to identify the sources for our photometry. One way of doing this involves using the FIND procedure which is based on the IRAF code called DAOFIND.

PROCEDURE	USE
FIND	Find positive brightness perturbations (i.e stars) in an image.

The subarray of interest is in the variable that we called "image500". We simply execute the code

find, image500, xcoo, ycoo

This code will prompt the user to input several parameters: first, it will ask for the approximate FWHM. Enter the value 2.0 and press return. Then it will ask for the minimum value above background for threshold detection in units of electrons. Enter the value 40 and press return. For the sharpness and roundness parameters, use the default values by pressing return. These values work just fine but you may want to play with them and experiment different outputs depending on how may stars you want to find and how many false detections you want to avoid.

The output of the FIND command gets stored into two variables: xcoo and ycoo. These are the source coordinates.

We can mark the stars that were found over our image:

EXERCISES
Exercise 4.3:
Create another PostScript file called "photometry_02.ps" *where we mark the stars using red open circles. You may want to use the following command:*

oplot, [xcoo], [ycoo], color = cgcolor("red"), psym = symcat(9)

There are many local maxima located above threshold. Notice that some bright sources are not marked. These are oddly shaped and are probably not stars. Other missed objects are too faint. Your plot should look like Figure 4.2.

4.4 Aperture photometry

Now that we have our source list, we can attempt to obtain a simple measurement of the stellar fluxes. We will achieve this goal by using the APER procedure

PROCEDURE	USE
APER	Computes concentric aperture photometry (adapted from DAOPHOT).

we use the following command:

aper, image500, xcoo, ycoo, flux, eflux, sky, skyerr, /flux

The input variables are the 2D array (image500), and the source coordinates that we obtained using FIND. The other variables are for the flux and sky output.

You will be prompted to enter the low and high bad pixel values. Use the default [none]. We will use an aperture radius of 3.0 pixels. If it asks for another star radius, leave this blank and hit return. Then it asks for the inner and outer radius for the sky. Use 5.0 and 8.0 pixels. Our image is in units of photoelectrons, so we will use the value 1.0 for the photons per ADU. There will be an output to the screen.

EXERCISES

Exercise 4.4:
What is the meaning of the /flux flag ? What happens if you omit /flux ?

Exercise 4.5:
Write a code that outputs the calculated flux for each star in the following format: xcoo, ycoo, flux, flux error, sky, sky error.

Exercise 4.6:
Create a figure that shows the flux error as a function of the flux. Use filled blue circles as symbols and label the plot accordingly.

5

Using IDL to study data cubes

5.1 Introduction

In this chapter we learn the basics of an important type of instrument in astronomy: the integral field spectrographs (IFS), which are spectrographs equipped with an integral filed unit (IFU). These instruments combine spectroscopic and imaging capabilities.

There are several of these instruments operating on the ground in observatories around the world, and they will be part of JWST.

Integral field spectroscopy is a technique to produce spectra over a contiguous 2D field, producing as a final data product a 3D data cube of the two spatial coordinate axes plus an additional axis in wavelength.

The purpose of this chapter is to show how to read a datacube, display its contents, and undestand the basics of this type of FITS files.

Initial data reduction to remove instrumental effects such as flat fielding, cosmic ray removal, and mapping between the 2D detector coordinates and the data cube, is highly instrument dependent. The initial data calibration is beyond the goal of this IDL tutorial.

5.2 Setting things up

We will study the datacube ngc4151_hband.fits provided by Tracy Beck. This is an infrared IFU obtained using the NIFS instrument at Gemini North. The source is NGC 4151 (the Eye of Sauron). Read more about this galaxy; it seems like an interesting object where lots of science is going on.

Place the supplied datacube inside a folder of your choice and define a path to the folder:

path = "/Users/your path here"

EXERCISES
Exercise 5.1:
Can you show how many extensions the FITS file has? What is the structure of the FITS file?

5.3 Datacube structure

This datacube is a 3D FITS file with the following structure: $[x, y, \lambda]$. The scale for all three axes is stored in the header using the standard FITS keywords:

CRPIX = reference pixel
CRVAL = value at reference pixel (either spatial coordinate or wavelength)
CDELT = dispersion in a given set of units per pixel.

We may start by reading the data set into an IDL variable called "cube":

```
cube = readfits( path + "ngc4151_hband.fits", hdr, exten = 1)
help, cube
```

This is an array with $59 \times 59 \times 2040$ pixels. This means that there are 2040 2D images (one for each wavelength). Each image is 59×59 pixels square.

In order to calibrate the λ axis, we retrieve the following header keywords:

```
crpix3 = sxpar(hdr, "CRPIX3")
cdelt3 = sxpar(hdr, "CDELT3")
crval3 = sxpar(hdr, "CRVAL3")
cunit3 = sxpar(hdr, "CUNIT3")
```

If you print this information, you will see that the first pixel is the reference pixel (crpix3), the wavelength associated with the first pixel is 14766.40 Å (crval3) and that the dispersion is $\sim$ 1.6 Å/pixel.

We can create a transformation that gives the third coordinate in units of Å instead of pixels. We first create an array that enumerates all the images:

```
pixel = findgen(2040) + 1.0
```

Then we create the transformation into wavelength:

```
lambda = crval3 + (pixel - crpix3) * cdelt3
```

EXERCISES

Exercise 5.2:
Repeat the same procedure with the other two dimensions in order to build two arrays: one for the x and one for the y. In what units are these axes given ?

5.4 Displaying the spectrum of a given IFU pixel

We can display the spectrum that is produced for a particular pixel in the IFU. We achieve this by simply selecting a pixel and plotting the cube value as a function of the wavelength.

Let us select the pixel with IDL coordinates $x = 30$ and $y = 30$. We will also select a range of wavelengths in order to avoid the xy planes at the beginning and end of the cube. The following commands will create the PostScript file shown in Figure 5.1.

We set the PostScript environment:

```
set_plot, "ps"
```

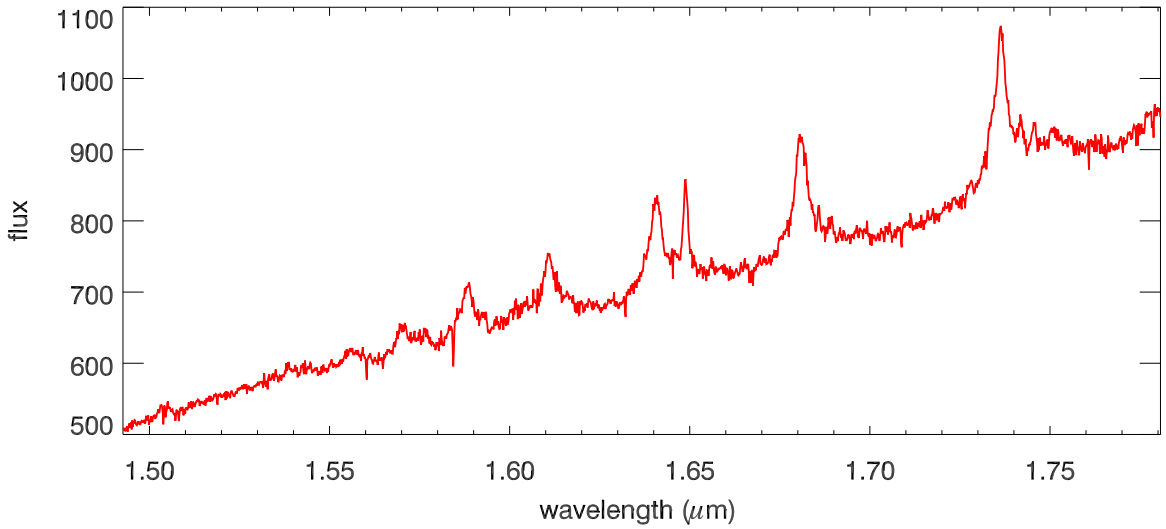


FIGURE 5.1— Spectrum of the pixel with IDL coordinates $x = 30$ and $y = 30$ in the NGC 4151 cube. The intensity is in arbitrary units and the wavelength is given in microns.

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and size of the plot: units are given in centimeters.
page_height = 27.94
page_width = 21.59
plot_left = 3.
plot_bottom = 4
xsize = 17.
ysize = 7.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $
scale_factor = 1.0, $
/portrait

We read the input cube:
cube = readfits(path + "ngc4151_hband.fits", hdr, exten = 1)

We select the pixel with IDL coordinates $x = 30$ and $y = 30$ and we reject some xy planes from the beginning and end of the cube.

image = cube[30, 30, 100:1900]
We translate the wavelength from Å to microns
lambda_microns = lambda[100:1900] / 1.e4

We define the axes of the plot:
cgplot, [0], [0], $
xcharsize = 0.8, ycharsize = 0.8, $
thick = 2, $
xrange= [min(lambda_microns), max(lambda_microns)], $
yrange= [500, 1100.], $
xtitle = "wavelength " + textoidl("(\mum)"), $
ytitle = "flux", $
xstyle = 1, ystyle = 1, $
yminor = 1, $
/nodata, /normal, /noerase, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We plot the spectrum:
oplot, lambda_microns, image, color = cgcolor("red")

We close the device:
device, /close

5.5 Displaying an *xy* 2D image for a given wavelength

We can also display one of the 2040 2D images. Let us select an image close to $\lambda = 1.65$ μm. The following commands will create a PostScript file shown in Figure 5.2.

We set the PostScript environment:
set_plot, "ps"

We define the file name for the output plot:
path = "/Users/your path here"
psname = path + "plot.ps"

We define the position and size of the plot: units are given in centimeters.
page_height = 27.94

page_width = 21.59
plot_left = 3.
plot_bottom = 4
xsize = 15.
ysize = 15.

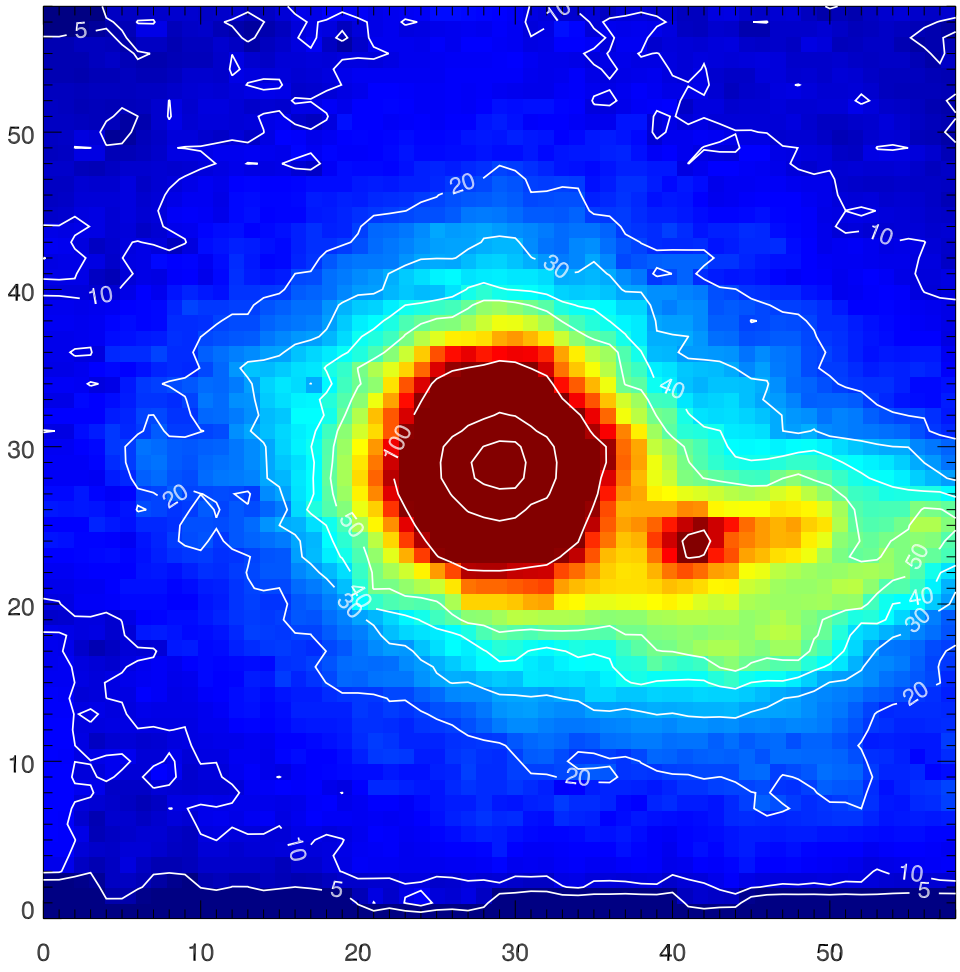


FIGURE 5.2— Image of the *xy* plane obtained with $\lambda \sim 1.65$ μm.

We open the PostScript device:
device, $
filename = psname, $
xsize = page_width, $
ysize = page_height, $
xoffset = 0, $
yoffset = 0, $

scale_factor = 1.0, $
/portrait

We load a color table:
cgloadct, 33, ncolors = 256, bottom = 0, clip = [0, 256], /reverse
tvlct, redvector, greenvector, bluevector, /get

We read the input cube:
cube = readfits(path + "ngc4151_hband.fits", hdr, exten = 1)

We select the plane with $\lambda = 1.65$ μm and scale the image:

image = 255b – bytscl(cube[*,*,1067.], min = 0, max = 100)
ss = size(image, /dimensions)

How do we know that plane 1067 corresponds to $\lambda \sim 1.65$ μm ?

We display the image:
cgimage, image, position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

contour_x = findgen(ss[0])
contour_y = findgen(ss[1])

We over plot a contour:
cgcontour, cube[*,*,1067.], contour_x, contour_y, $
levels = [5, 10, 20, 30, 40, 50, 100, 200, 600] , $
xstyle = 1, $
ystyle = 1, $
axiscolor = "black", $
xcharsize = 0.8, ycharsize = 0.8, $
c_colors = cgcolor('white'), /noerase, $
position = [plot_left / page_width, plot_bottom / page_height, (plot_left + xsize) / page_width, (plot_bottom + ysize) / page_height]

We close the device:
device, /close

EXERCISES

Exercise 5.3:
What is the wavelength range of this cube ? Repeat the procedure for another plane with an interesting IR wavelength.

6

IDL fitting routines

6.1 Introduction

In this chapter we introduce some fitting routines that come with IDL. Ceratinly the easiest functions to fit to a number of empirical points are polynomial functions . Astronomers also use more elaborate functions to fit their data, depending on the physical laws involved in the process, such as the Gaussian function, or combinations of many functions.

6.2 Fitting a polynomial to empirical data

Edwin Hubble recognized the relationship between a galaxy's redshift and its distance. This relationship is linear and there is a simple mathematical expression for Hubble's Law:

$V = H_0 \times D$

where:

V is the recessional velocity, typically expressed in km/sec.
H_0 is Hubble's constant.
D is the proper distance in mega parsecs (Mpc)

The provided file "hubble.dat" has some fake empirical data: first column is the distance in Mpc and the second column is the velocity in km / sec.

We will fit a first degree polynomial ($y = ax + b$) to the data and determine the slope of the line. In our context, the slope represents Hubble's constant.

We read the file "hubble.dat"
path = "/Users/your path here"
readcol, path + 'hubble.dat', distance, vel

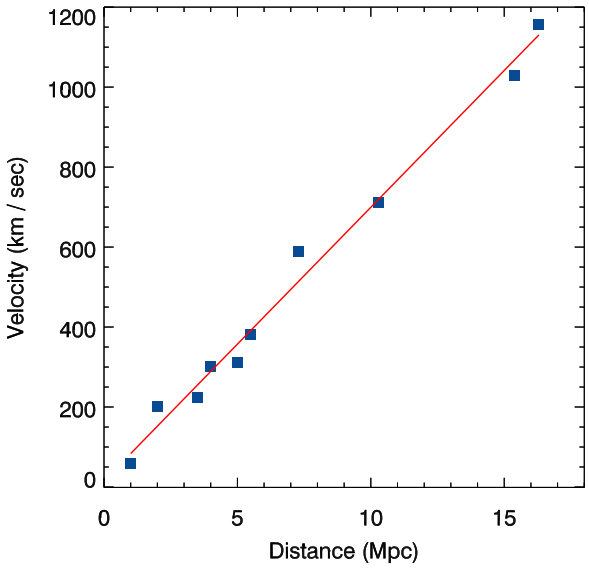


FIGURE 6.1— Hubble's law. The empirical points are represented with blue squares and the red line represents the line fit. The slope of the red line is the best fit to the Hubble constant.

PROCEDURE	USE
POLYFIT	performs a least-square polynomial fit with optional weighting and returns a vector of coefficients,

We estimate the coefficients a and b using POLYFIT
coeff = polyfit(distance, vel, 1)

```
print, "Hubble constant = " , coeff[1], " km/ (sec * Mpc)"
```

For this example we fit a simple first-degree polynomial, though any higher-degree polynomial can be generated as well. If the fit model is desired (rather than just the coefficients), setting the YFIT keyword will store the model (the value of the polynomial calculated at each of the input x values).

```
coeff = polyfit(distance, vel, 1, yfit=model)
```

EXERCISES

Exercise 6.1:
What values of the Hubble constant do you get? Can you figure out the error in H_0 ?
Exercise 6.2:
Produce an IDL code to plot the empirical data and the line fit as shown in Figure 6.1.

6.3 Fitting a Gaussian function to empirical data

In spectroscopy astronomers measure the line profiles by fitting functions to the stellar flux as a function of wavelength. A raw estimate of line properties such as the full-width at half maximum (FWHM), the line wavelength, etc can be achieved by fitting a Gaussian function:

$$f(x) = A \cdot \exp\left(-\frac{(x-B)^2}{2 \cdot C^2}\right)$$

This is provided by IDL in the GAUSSFIT function.

PROCEDURE	USE
GAUSSFIT	Computes a non-linear least-squares fit to a function $f(x)$ with from three to six unknown parameters. $f(x)$ is a linear combination of a Gaussian and a quadratic; the number of terms is controlled by the keyword parameter NTERMS

We will fit a Gaussian function to the most intense line in the spectrum provided in Figure 3.4.

We read the spectrum using:

```
spec = mrdfits(path + "o5bn02010_x1d.fits", 1, hdr)
wave = spec.wavelength flux = spec.flux
```

We assign the flux and wavelength to IDL vectors:

```
wave = spec.wavelength
flux = spec.flux
```

We sort the wavelength variable:
i = sort(wave)
swave = wave[i]
sflux = flux[i]

We scale the flux:
scaled_sflux = sflux / 1e–13

We select the wavelength region around the spectral line of interest (1547 Å$< \lambda <$1549 Å):

wline = swave[34610: 34900]
fline = scaled_sflux[34610: 34900]
fit = gaussfit(wline, fline, coeff, nterms = 3)

print, "A = " , coeff[0]
print, "B = " , coeff[1]
print, "C = " , coeff[2]

This is telling us that an estimate of the central wavelength for this particular line is 1548.15 Å.

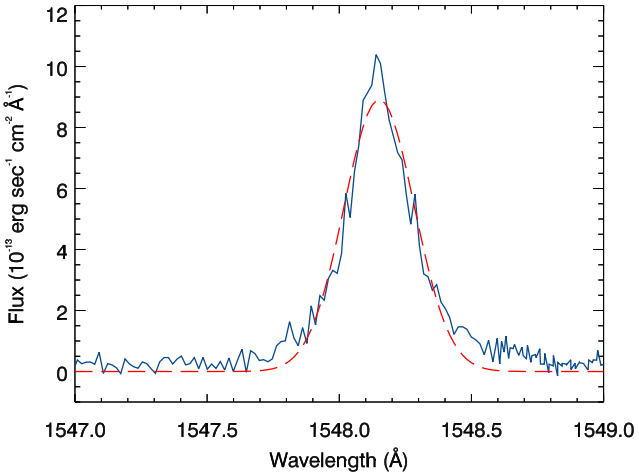


FIGURE 6.2— Gaussian fit to the spectral line with wavelength ∼ 1548.15 Å from the COS spectrum o5bn02010_x1d.fits. The scaled flux is shown in a blue line. The red dashed line is the Gaussian fit.

EXERCISES

Exercise 6.3:
Calculate the FWHM for this line, assuming a Gaussian profile.
Exercise 6.4:
Produce an IDL code to recreate Figure 6.2. Include a vertical line to show the position of the spectral line as shown in Figure 3.4.

6.4 Fitting user-specified functions to empirical data

In some cases polynomial and Gaussian functions are insufficient models for empirical data. This is often true in spectroscopy, where models are often complicated functions or even tabulated data measured in labs. There is some built-in functionality for models like this in IDL, though a more functional and robust library MPFIT has been written by Craig Markwardt. MPFIT is able to fit models with any number of free parameters in user-specified functions.
Download the library from https://www.physics.wisc.edu/ craigm/idl/fitting.html and add the library to your IDL path.

The provided file "cmb_spectrum.txt" contains some measurements of the CMB across various frequencies. The first column is the frequency (in Hz), the second is spectral radiance (in W/(m^2*Hz*sr)), and the third column in the one-sigma uncertainty of the radiance (in the same units). We will be fitting a curve given by Planck's law. Our free parameters will be the blackbody temperature and, for the sake of demonstration purposes, the speed of light.

We will be using the MPFITFUN routine to fit a user-defined function to the data.

PROCEDURE	USE
MPFITFUN	Computes a non-linear least-squares fit to a function $f(x)$ with any number of unknown parameters. $f(x)$ is a user-specified function.

To begin, we first need to create and IDL function that returns the model we want to fit, given the independent data points as well as values for the free parameters:

```
function blackbody, x, p
; x is the array containing the independent data points,
; p is an array containing values for the free parameters,
; p[0] is the value for the blackbody temperature,
; p[1] is the value for the speed of light.

; define Planck's and Boltzmann's constants
h = 6.626D-34 ; in units of m^2 kg s^-1
k = 1.38D-23 ; in units of m^2 kg s^-2 K^-1
; compute model
B = 20.*h*x^3/p[1]^2*(exp(h*x/(k*p[0]))-1)^(-1)
return, B
```

Save this function in your working directory as "blackbody.pro".

With our function defined, we now need to read in and fit the data.

We begin by reading the data:
path = "/Users/your path here"
readcol, path + 'cmb_spectrum.txt', freq, val, err

MPFIT requires starting guesses for the values of the free parameters, in general rough estimates for the parameters will do:

start = [4.0,1D8] ; in the appropriate units

The first element in the array start is our guess for the temperature, the second is our guess for the speed of light.

We then call the fitting routine:

res = mpfitfun("blackbody", freq, val, err, start)

The arguments for mpfitfun are: the name of the user-specified function, the independent data values, the dependent data values, the one-sigma uncertainties and the starting guess values. The result "res" contains the final calculated values for the free parameters.

The routine iterates a number of times to get the best fit and prints the statistical values indicating the goodness of fit. To construct the fit model from the calculated values of the free parameters we simply call our user-specified function, giving it the independent data values as well as the calculated parameters:
model = blackbody(freq, res)

MPFIT is a powerful library, which has uses far beyond the level of this toy problem. Tabulated spectra can be can be fit to empirical data by reading the tabulated data in the user-defined function, for example. It should be noted, however, that there are limitations. The most obvious one stems from the fact that MPFIT uses gradient-decent methods, which do not necessarily find the global minimum for the least-squares fitting, rather the local minimum in the parameter space. Therefore, it is important to check the fitting results to make sure a good fit is achieved.

EXERCISES

Exercise 6.5:
Calculate the temperature of the example CMB data, as well as the speed of light using MPFITFUN.
Exercise 6.6:
Produce an IDL code to plot the empirical data and overplot the fit model. Plot a vertical line through the peak wavelength of the data.

7

IDL resources

7.1 Useful IDL-related websites

The IDL Astronomy User's Library The IDL Astronomy Users Library is a central repository for low-level astronomy software written in the commercial language IDL.

http://idlastro.gsfc.nasa.gov/

Coyote's Guide to IDL Programming Coyote's Guide to IDL Programming is the personal web page for IDL consultant and trainer David Fanning.

http://www.idlcoyote.com/

Scientific Data Visualization Software from ITTVIS The developers of IDL.

www.exelisvis.com/idl/

Markwardt IDL Library Craig Markwardt has developed some excellent IDL codes that are made public.

http://www.physics.wisc.edu/ craigm/idl/idl.html

astroplotlib A library of IDL and Python templates that are useful to create figures of astronomical interest.

http://astroplotlib.stsci.edu