Observational Astronomy
ASTR 310
Fall 2007

Introduction to IDL

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.

IDL can be used to write programs to calculate things, just like FORTRAN or C. In this area, its strength is that whole arrays of numbers can be represented by a single name and can be manipulated without any explicit reference to the indices of individual elements of the array. This feature is based upon the APL family of languages (APL2, J, etc.) It is also like the APL languages in that it is *interactive*, i.e., you type in an expression and it is executed as soon as you hit "return". (In languages like FORTRAN or C you must *compile* a program before running it.)

Finally, there are a wide variety of built-in commands in IDL for plotting data, displaying images, and printing out the results. Thus you can go back and forth between looking at a data set or image, and making alterations and transformations to that data.

2 Logging on and Starting IDL

Look to see if the Dell computer is running (Mandrake) Linux. (If it is running windows, close all the applications that might be running and reboot the computer – Ctrl-Alt-Del if all else fails. When the machine comes back up, (quickly) select "Linux", using the arrow keys.) Enter your account name and password to login.

- If you have entered a valid name and password, you will be in, but you still must enter **startx** to start up X-windows. If one or more terminal windows do not open up, click the terminal icon to open one. Linux is basically a Unix operating system, and understands Unix (not DOS, etc.) commands. For example, type "pwd" (print working directory). It will respond with your current directory. Or type "ls" to get a listing of files (the Unix equivalent of the DOS "DIR").

- In one of the windows, type "idl" (lower case – Unix is case sensitive, but IDL is not!). You will eventually get some information about the IDL version number, etc., followed by the "IDL>" prompt. You're now running IDL.

  You can get help with function names, etc., at any time by typing "?". A window will pop up with various options.

  You can see some nice demos by typing "demo".

- When you are done, type "exit" to leave IDL. **Then type "Ctrl- Alt-Backspace" to leave X-windows, and "logout" to log off the machine – otherwise you are leaving your account open to anyone who happens by.** Have a nice day :-).

3 An IDL test drive

At the >IDL prompt, try typing "print, 4 + 3". (Without the quotation marks!)

The response "7" will be returned.

Now type "x=3 & y=5". (The "&" symbol separates statements on the same line.)

Nothing is printed, but the variables x and y have been defined and set to the values 3 and 5. To see this, type "print, x, y,(x+y)". The response is "3 5 8".

All the usual arithmetic operations are defined. Type "print, x+y, x-y, x/y, x*y, xˆy". IDL returns "8 –2 15 0 243". Why is x/y (3 divided by 5) given as 0? Because IDL has different data types: integers, floating point, double-precision, complex numbers, etc. Since x & y were entered without a decimal point, they were implicitly defined as integers, and the integer part of 3/5 is zero. To get the usual result, enter "x=3.0 & y=5.0". Then retype "print, x+y, x-y, x/y, x*y, xˆy". (In IDL, you can bring back an earlier command by using the up-arrow key. Try it.) The result is now "8.00000 –2.00000 15.0000 0.600000 243.000". Also, mathematical functions like SQRT (square root), ALOG (natural logarithm), SIN (sine - the trig functions expect the arguments to be in radians) are there: Type "print, SQRT(2), ALOG(10), SIN(!pi/4)" and see the result: "1.41421 2.30259 0.707107". The quantity "!pi" is the predefined constant $\pi = 3.141592654$.

At this stage, IDL looks like nothing but a hand calculator, but there is much more. First, a single variable name can represent a vector, or a two-dimensional array of numbers – a matrix. (And an image is nothing but an array of numbers ...) To see how this works, enter "x = [1, 3, 7, 10, 200]". Then look at the following:

"print, x" → 1 3 7 10 200

"print, x+2" → 3 5 9 12 202

"print, 5*x" → 5 15 35 50 1000

Next, try this (the $ sign is used to continue an expression onto the next line):

```
a=[[2,3,0,6],[7,10,1,4], $
   [-5,5,1,8],[13,-2,17,9]]
```

Then "print, a" gives

```
 2   3   0  6
 7  10   1  4
-5   5   1  8
13  -2  17  9
```

So a is a 4×4 matrix. If you want the inverse of the matrix, just type "ai=invert(a)" and "print, ai". You will see

```
 0.126447   0.0181374   -0.109982   0.00540262
-0.145614    0.108310    0.0666324  -0.0102907
-0.218292   0.0307435    0.0901724   0.0517108
 0.197324  -0.0602007   0.00334448  0.00334448
```

In the same way, a could be a 500×500 array of pixels – i.e., it could be an *image*. Then, if you wanted the logarithm of the image, you would just write "b=ALOG(a)". No need to reference the individual elements (pixels)! Of course, sometimes you need to reference a particular element of an array. In IDL, square brackets are used for this. **Note that in IDL the first index of an array is always zero, not one!** Try the following:

"print, a[2,3], a[3,1], ai[2,1]" → 17 4 0.0666324

Notice that for arrays with two dimensions, the first index refers to the column and the second refers to the row of the array.

A very important feature is the ability to cut out a *range* of elements; you need this, for example, to extract a specific part of an image. In IDL, if you enter say, "20:35" as an index within the square

brackets, you will get all the elements with the indices $i = 20, 21, 22, ..., 34, 35$. For example, it you enter "b=a[0:2,1:3]", then "print,b" will yield

```
 7   10    1
-5    5    1
13   -2   17
```

3.1 Plotting your data

We have seen how IDL can manipulate arrays of numbers. But that is only a small part of its utility. Let's see how it can plot your data. First, let's create something to plot. One useful function in IDL is **findgen** (i.e., **f**loating **ind**ex **gen**erator). The function "findgen(n)" creates an array of dimension *n*, with values 0.0, 1.0, 2.0, ... (n-1). So let's use this as follows: Note that the expression "findgen(n+1)/n" will divide the interval from zero to one into n equal parts. Thus the expression

x=4*!pi*findgen(401)/400

generates an array, x, of 401 numbers between 0 and 4π. (Try "print,x" to see these values.) Now let's create a function to plot. How about sines and cosines? Enter this:

y=sin(x) & z=cos(x)

Now, enter

plot,x,y

Voila! a window opens with the plot of x vs. y. Next, try the "oplot" command (i.e., overplot - plots over the previous plot without erasing it):

oplot,x,z,linestyle=2

The "linestyle=2" part gives a dashed instead of a solid line, so we can tell which is which. There are lots of options with the plot command that let you do just about anything you want, but there are default values so you can keep it simple when you are just starting out.

Now, let's look at the display of images. First, we must consider how we get the data into IDL – we can hardly enter a few hundred thousand numbers by hand! IDL has the ability to read many image formats, but the one most used in astronomy is the FITS format (Flexible Image Transport System). Files in this format end in **.fts** or **.fits**. In your directory is an image of the Sombrero Galaxy in the FITS format called "sombrero.fts". To get it into IDL type these statements:

im=readfits('sombrero.fts',hd)
help, im
im=rotate(im,3)
help, im

The first statement reads in the file and stores it as an array called "im". If there is a header file (which contains text, such as a description of the image), it will be stored in the variable called "hd". The second statement just tells about the variable **im**: in this case "Array[384, 576]" of integers. "help" is very useful – if you type help without any name, it will tell you about *all* the variables, functions, etc. that are defined. Since the image looks better the long way horizontal, "rotate(name,3)" switches rows and columns. Then "help, im" shows you that it is now [576, 384].

So how do we look at this image? Just type

tvscl, im

The **tvscl, im** command opens a window and displays the the variable whose name is **im**. The "scl" part of the name stands for *scale*, which means that the image is scaled so that the pixels with the highest value come out white, and the lowest are black. Thus, if you display (5000*im), it will look exactly the same. This image is not very impressive, because the nucleus of the galaxy is bright and this makes all the faint outer parts look almost the same, i.e., black. One way to represent a wider range of values is to use *color*. (Sometimes called *false color*, since the colors only represent different levels of intensity at the same wavelength, and have no relation to the actual colors of the object.) There are an infinite variety of mappings from intensity to color (called *color tables*). Here is one; type

loadct, 15

This loads color table 15, STERN SPECIAL, (my personal favorite). Now you can see a bit more detail away from the nucleus. (That's just a star near the top.) If you want to look at the full range of color tables, you can type "xloadct", and a window will pop up with all sorts of choices. "loadct, 0" gets you back to plain black & white.

Still, there is a lot more to see in this image. One problem that often happens is that a couple of pixels with anomalous values will throw the scaling way off. And you often get odd values at the very edge of the image, etc. To see the range of values that *tvscl* is scaling to, just type

print, minmax(im)

This function returns the minimum and maximum values in any array. You will see that result is "384 4742". But 384 is an anomalous low value. You can examine the values from point-to-point in the image by typing

curval, im

Now, as you move the cursor with the mouse over the image, you see listed in the IDL window four values: X, Y, Byte Inten, Value. X and Y are just the indices of the pixel your cursor is at. *Value* is the value of that pixel (i.e., it is the number im[X,Y]). Finally, *Byte Inten* is the scaled number, and integer between 0 and 255, that is used to choose the display color from the color table. As you move the cursor around, you see that all the numbers are over 900. There is one low pixel hiding in a corner somewhere that is fooling tvscl. What can we do to ignore the extreme values in the array? In IDL it is simple. First, you need to understand how IDL evaluates and expression of the form $a < b$ or $a > b$. Lets generate a string of numbers:

x=indgen(21)

This generates the string of integers 0, 1, 2, ... 20. Type "print, x" to verify this. Now, type

print,x<11

You see that all the numbers in the array *x* that are greater than 11 have been replaced by 11. Likewise, type

print,x>4

and see that all the values of *x* less than 4 are replaced by 4. And you can combine the two:

print,x>4<11

and only the values of x between 4 and 11 are left.

Now you can understand that the expression **im>900** will replace all values of **im** that are below 900 with that minimum value, and **tvscl** will work much better. Type this:

tvscl,im>900

Now that's a lot better! You can see the bulge of the galaxy and the dust lane that runs beneath the nucleus. Still, the core of the galaxy is so bright, we still aren't seeing what's going on in the faint regions. We can disregard the *highest* values also (this will "burn out" the nucleus of course) and use the full range of the color table on the lower intensities by using the $<$ relation:

tvscl,im>960<1400

Finally, we begin to see what's out there! Another trick to get both high and low values is to take the square root or the logarithm of the image. Try these:

tvscl,sqrt(im>900)
tvscl,alog(im>900)

By now you can begin to see the power of IDL for making mathematical transformations of your data to bring out "hidden" detail. You should also try the **zoom** command. When you type that, and left-click on some part of your image, a new window opens showing that region in detail. If you click the center mouse button, a little zoom factor window opens – try clicking the maximum factor of 20, then left-click the image again. Now you can clearly see each individual pixel of your region. Clicking the right button exits zoom.

You may have noticed that the window with the image is actually larger than the image itself. You can always control the size of the window directly, with the command "window,xsize=nx,ysize=ny", where *nx* is the number of columns in the image and *ny* is the number of rows. In this case it would be 576 and 384:

window,xsize=576,ysize=384

There are lots of other ways to display this image. For example, try these:

shade_surf,(im>960<1400),az=80,ax=20
contour,(im>960<1400)

3.2 Getting printed output from IDL

At some point, you may produce a plot or an image that you want to print out. The best way to do this is to switch into the *Postscript* mode and make a postscript file. Then you can send the file to the printer. Lets go back and generate a function to plot like before. An interesting function we will encounter in this course is sin(x)/x. Start by making a thousand points between 0 and 10π, and then evaluate the function at those points:

x=10*!pi*findgen(1001)/1000
sn=sin(x)/x

You got an error message:
"% Program caused arithmetic error: Floating illegal operand"
To see why, look at the first few values of **sn**:

print,sn[0:4]

You will see "NaN 0.999836 0.999342 0.998520 0.997370"

What's this NaN? It means "Not a Number", because the first x value is zero, and we were trying to compute 0/0, which is not defined. Perhaps you know that sin(x)/x does have a value at x=0, and that value is 1. So we can fix that one bad value of the **sn** array like this:

sn[0]=1.0

Now, lets look at this "damped sine wave" function:

plot,x,sn

Now to make a plot we can print out, follow these steps:

1. Switch to postscript;
2. give a name to the postscript file we will create;
3. tell IDL to plot it sideways (/landscape, the default is upright);
4. do the plot (i.e, enter here whatever command(s) you would use to make a plot on the screen);
5. close down Postscript; and
6. go back to writing to the screen (X-windows).

Here are the exact commands that do each of these steps:

set_plot,'ps',/copy
device,filename='snplot.ps'
device,/landscape
plot,x,sn
device,/close
set_plot,'x'

It's now time to quit IDL, and see about that plot. Type **exit**. You are back to Unix. Enter the command "ls" and see that the file "snplot.ps" is in your directory. You can view this file and see that is what you intended by invoking a program called "ghostview":

ghostview snplot.ps

If everything is OK, leave ghostview (Click quit, which is under "file"). Now, send the file to the printer by typing "lpr" (for "line printer", even though it's a laser printer):

lpr -Plabs snplot.ps

Now you can go to the printer in the corner and get your printout. While it is possible to make color prints, we will not do this, since they have to be sent to a special printer and are quite expensive.

4 Other Stuff

4.1 When IDL crashes ...

Sometimes when IDL is running a command, it may hit an error and then it looks like some of your images or other variables have vanished. When IDL crashes like this, you should type **retall**. The problem is that you are still stuck in the routine that crashed and you need to get back to the main command level.

4.2 Reading and writing text files

To write out an array of numbers:

openw,1,'file'
printf,1, array_name
close,1

To read in an array of numbers:

openr,1,'file'
a=fltarr(columns,rows)
readf,1, a
close,1

Read colum N of a table of numbers in a file:

x = rdcol('file',n1,n2,N)

where n1 = first row, n2 = last row, N = column to be read. This is very useful, since if there is text in the file before you get to the table, you just set n1 to the row where the numbers start, and IDL will not try to read the text. Also, you don't have to know in advance how long the column of numbers is: you just set n2 to a very big number (i.e., 99999) and rdcol will stop when it gets to the end of the file.

4.3 Running commands from files

Instead of typing in commands while we are running IDL, we can also read them in from a file. Before starting IDL, you can use any text editing program (such as "PICO") to type IDL commands into a file which you then save with some name, say "myfile". Then, after we start IDL, we can just type

@myfile

and IDL will run the commands. This is really useful if you are working with some images, and you always have to start by reading them in, rotating them, etc. Just put those commands in a file and each time you start IDL, you can read in your data effortlessly.

4.4 Functions and procedures

This is not a course in IDL programming, but it is useful to know how you do more complicated or repetitious tasks. You do this by writing **functions** and **procedures**. The argument(s) of a function is placed in parenthesis (separated by commas if there is more than one); the result just replaces the function name in the expression. Thus

b = sin(a)

which sets b to the sine of a, is a function. *In IDL you can define your own functions.* Thus the statements

function pythagorean,a,b
c = sqrt(a*a+b*b)
return, c
end

define a function **pythagorean**, which will calculate the hypotenuse of a right triangle with sides a and b. Type these statements into a file and save it with any name, say 'pythagorean'. Then when you are running IDL, you *compile* the function with the statement

.run pythagorean

After that, you can use it anywhere, just as you would use sin. Thus if you type

print, pythagorean(4,3)

you will get "5.0000".

A function returns only one value. More complicated tasks are done with a *procedure*. To invoke a procedure, you type its name, followed by the list of it's arguments, separated by commas. Some of the arguments may be inputs, some may return results, some may do both. We have been using procedures all along, such as

tvscl, im

where **tvscl** is the name of the procedure and **im** is its argument. Just like functions, we can define our own procedures (which can of course call other functions and procedures). To see what the syntax looks like, consider the following:

```
pro mean2, array, numb, mean, sd, sdm
; procedure to give the number of elements, the mean, the standard
; deviation, and the standard deviation of the mean, of an array.
numb=n_elements(array)
mean= total(array)/numb
delta=array - mean
sd=sqrt(total(delta*delta)/numb)
sdm=sd/sqrt(numb-1.0)
return
end
```

The lines beginning with semicolons are comments (anything after a semicolon is not read by IDL). You type these statements into a file that you could call mean2.pro. Then, running IDL, you would compile it like a function: **run. mean2**. After that, if you have defined some array of numbers called **a0**, you can call it with the statement

```
mean2, a0, n, av, sig, sig_m
print, n, av, sig, sig_m
```

and IDL will print out the number of elements in the array, the mean of those elements, the standard deviation of the values about the mean, and the the standard deviation of the mean. Note that the names of the arguments don't have to be the same as in the definition of the function, it's just their position that counts.

4.5 Libraries of functions and procedures

In addition to procedures and functions built into IDL (and for which you can find on-line documentation by typing "**?**" while running IDL) there are libraries of functions and procedures contributed by the user community. The most important of these are the ones from Goddard Space Flight Center, which include all the DAOPHOT routines we will use in the observing projects. These are in a directory where IDL can find them; if you want to look at the IDL code – which always contains a description of how to call the procedure – look in the directory

/local/pkg/rsi/idl/user_contrib/goddard/pro/

Another useful collection, which contains some things we will use, such as the function **mean()**, is the one from Johns Hopkins Applied Physics Lab:

/local/pkg/rsi/idl/user_contrib/jhuapl/

You can bring up an interactive window with information on the user contrib functions by typing

widget_olh

4.6 Plotting in more than one window

Sometimes you want to have two images displayed at the same time, for example, when you are comparing star fields to see if they overlap. You can do this easily. When you open a window of a given size, you can type

window,n,xsize=nx,ysize=ny

where **n** can be 0, 1, 2, etc. (When you don't include n explicitly, it assumes it is n=0). After you've put up window **n**, if you want to go back to a previous window **m**, you just type

wset,m

4.7 Saving work in progress

If you have to quit and want to save the images and variables that you have defined, it is possible to do this by typing **SAVE**. You can then log off, and when you next start IDL, you can get all your stuff back by typing **RESTORE**. But be careful with this, because the file of saved data which appears in your directory (called "idlsave.dat") can be huge! Don't try to do a save if you have lots of images.

4.8 The Mouse Buttons

Some interactive procedures make use of the left, middle and right mouse buttons. The Dell computers in CSS 1220 have two buttons and a wheel. The middle button in this case is simulated by clicking the wheel.

5 A handy procedure for getting a hardcopy

I've hacked together the following IDL procedure to make it easy to send your plots, etc. to the printer. All you need to do is type **hardcopy** and the contents of the current window are sent to the printer "labs" in CSS 1220. Here's the code that does it:

```
pro  hardcopy
; produce b&w printout from screen plot or image
; resize to fit 8.5" x 11" page with 0.5" margins
im=tvrd( )  ; grab the screen image
sz=size(im)
; if there are three color planes, flatten:
if sz[0] gt 2 then begin
  sum= im[0,*,*]+im[1,*,*]+im[2,*,*]
  im=bytarr(sz[2],sz[3])
  im[0:(sz[2]-1),0:(sz[3]-1)]=sum[0,0:(sz[2]-1),0:(sz[3]-1)]
endif
low=min(im) & hi=max(im) & mid=0.5*(low+hi)
av=total(im)/n_elements(im)
; if most pixels are dark, reverse image:
if av lt mid then im=(hi+low)-im
sz=size(im)
; rotate image if width greater than height:
if sz[1] gt sz[2] then im=rotate(im,3)
sz=size(im)
sx=1.0*sz[1] & sy=1.0*sz[2]
width=7.5 & height=width*(sy/sx)
if height gt 10.0 then begin
  height = 10.0 & width = height*(sx/sy)
endif
dely=0.5*(11.0-height)
delx=0.5*(8.5-width)
set_plot,'ps',/copy,/interpolate
device,/portrait
device,yoffset=dely,xoffset=delx,/inches
device,filename='temp.ps'
tvscl,im,xsize=width,ysize=height,/inches     ;write Postscript file temp.ps
device,/close
set_plot,'x'
spawn,'lpr -Plabs temp.ps'       ; Unix command to send temp.ps to printer
spawn,'rm temp.ps'               ; and then delete it.
return
end
```