Introduction to Astronomical Computing with IDL

Version updated 25 August 2016 by G. Chartas. Adapted from versions by J. Carson and J. Neff.

Exercise #1 – Getting Started With IDL

This is an exercise to get you started using IDL, one of the most common languages used by astronomers. It was written by scientists for scientists, so the syntax is fairly easy to master, and it provides a platform to apply your scientific creativity to make the computer do what you want. Compared to other programming languages and software packages, it is very efficient at handling large grids of numbers, which is the underlying format of any digital image. If you don't have programming experience, some of the early learning curve might seem obscure, but it will be time well spent. Unlike the case for pre-built software programs like Photoshop or Powerpoint, where you have a constrained menu of options from which you must select, in a programming language you are in charge. There are no limits or restrictions on what you can make.

I. The Nature of IDL; Starting Up IDL; Getting Help

IDL is several things at once. First and foremost, it is a high-level programming language. It is also an INTERACTIVE operating environment. Finally, it is a collection of an almost unlimited library of useful tasks and programs, mostly written by other scientists. You can be issuing commands interactively, compiling and running programs, and calling library routines all at the same time.

There are two ways you can startup IDL.
One approach is to open a terminal window cd to the directory where you would want to run IDL and type idl. You should see some information regarding the IDL version number and whether your license file is installed correctly. The command line prompt should say 'IDL>'.

A second way of running IDL is through a graphical interface. To open the graphical interface find the IDL app in the `/Applications/exelis/idl` directory and drag it to the DOC. Double click on the IDL icon to launch IDL. If everything is running correctly, you should see a graphical interface with a text editor in the upper right and a command line prompt toward the bottom. The command line prompt should say 'IDL>'.

You can use either approach.
The first IDL command you should type at the IDL prompt is **HELP**. This command tells you what is in IDL's "active" memory. There is a memory level (it should say $MAIN$), variables (you won't have any yet), compiled procedures (probably

ASTROLIB and $MAIN$), and compiled functions (you shouldn't have any of these yet either). Type HELP regularly as we go along. You'll quickly develop a feel for what's going on. One reason IDL is so flexible is that it operates primarily in the RAM (temporary memory). When you start it up, it allocates a fixed amount of space. $MAIN$ is the top-level in the memory tree, kind of like C: is generally the top level in the Windows file system. Most of the time, you will be interacting on the $MAIN$ level, but sometimes you get stuck in a subroutine. If that happens, IDL will stop and allow you to fix things and move on. In that case, when you type HELP, it might appear that IDL has lost everything you thought it knew. Have no fear, it's still there. HELP just shows you the active part of the memory. This might sound like a bug right now, but you'll soon see it to be a valuable feature.
If you type a **?**, you should get a separate window that provides access to all of the on-line documentation. If it doesn't work for some reason, just google IDL help. There's more documentation elsewhere on the web. You can leave this help window up all the time if you wish.

II. Variable Types: Special Symbols

Unlike a calculator, where a number is just a number, IDL can store a single number with various levels of precision. This fact can be a bit annoying, but when you are applying a mathematical function to a billion different numbers for instance, which we sometimes do in astronomy computing, it is important to not use more precision than you need. In IDL, *Integer* variables store numbers from -32767 to +32762. *Long-word Integers* are used for the largest numbers. Most of the time, we will be using *Floating point* variables, which can go up to fairly high values, and have a precision of 0.00001. *Double precision* numbers are the most precise. They take up the most memory and have a precision of 0.0000001. Remember that IDL works in the RAM (temporary memory), so the more space your variables take up, the less free memory is available to crunch numbers. The lab computers have a fair amount of memory, so I don't think this will be an issue, but for some customized processes it might.

IDL typically stores text with the *STRING* variable. For example, you could type **silly='Hello there'** (followed by enter) and then **print,silly**. Go ahead and try it. Now type **a=2** then type **b=2L** then type **c=2.0** then type **d=2D**. Type **HELP** and see if it all makes sense.

One powerful aspect of IDL compared to other languages is that you do NOT have to explicitly tell it what kind of variable you want to store your value in. It will chose the easiest one by default. Unlike most other languages, it can also operate on variables of different types. For example, let's operate on your numbers: **e=a*b f=a*c g=a*d** (hit the enter key between each command) then **print,a,b,c,d,e,f,g** and then type **HELP**. Can you follow what's going on? What will happen if you type **print,a*silly** ? How about **print,1/a,1/c** ? Try them out and see if it makes sense. IDL usually converts the variable types before operating on them

in order to maintain the highest precision, but there are exceptions. If you give it all integers, it will not convert them to floating point before operating on them. But if one of your numbers is floating and the rest are integer, it will convert all of them to floating before the operation. Note that it doesn't change your variables in its memory permanently, just long enough to get the job done (i.e. type HELP and you'll see that a is still an integer).

IDL also provides commands to convert a variable of one data type to another. The function FIX turns a floating point or string (under certain conditions) into an integer [type **print,fix('2'),fix(1.49), fix(1.51)**]. The function FLOAT turns an integer or string into a floating point. The functions LONG and DOUBLE convert integers and floating point to double precision.

There are very few rules regarding variable names, but there are special symbols that should never be used. Any variable that begins with **!** is a "system variable". These are variables whose value is known to IDL no matter what level or subroutine you are in. Otherwise, when control passes to a subroutine, all the "main" level variables are temporarily forgotten until the subroutine is finished. You have a few system variables set up already.

Try typing **print,!path**. The resultant printout should show you where IDL looks for programs in the cases where you enter a command that is not a standard IDL procedure. IDL searches for a file that has the same name as your command, but ends with a .pro extension. Then it will compile and immediately execute that program. Part of your search path is procedures that came with IDL, and part of it is stuff that I loaded that is relevant to astronomical data analysis. It should also show the paths designated by your startup file. When you get a chance, start snooping around in your search path to find, read, and try to interpret some real IDL programs.

Don't put $ in a variable name. Also, NEVER use spaces in a variable name or a procedure name. Comment lines in IDL programs start with a semicolon, so never use one of those in your variable and procedure names. Don't use **.** in the name either. IDL has another (actually several more) variable type called a structure. These are VERY powerful, but they have **.**'s in them. For example, **print,!d.name** and it should tell you what kind of device IDL is communicating with. Some special commands begin with a "dot" (e.g. .run), so just don't use it in variable names (your procedure names will be called *something.pro*, but to IDL you will just call them *something;* for example, a program called *juldate.pro* would be run using the IDL command *juldate*).

Dashes (-) are not a good idea in variable names, because they might get interpreted as a minus sign! If you need some kind of delimiter, use an underscore (_). If you read an introductory IDL manual, you will come across several more special symbols, but the ones I mention here are the main culprits.

Sometimes you might have more than one version of a program stored on the computer. To determine where IDL is grabbing a program from, type for example, **findpro,'readfits.pro'**. What information is outputted by the which command? Pay attention to this command. The **findpro** command can be very important if you have more than one version of a script on your computer. **findpro** will tell you which version is being accessed by IDL, and where other versions lie.

Finally, note that IDL is not case sensitive, *but filenames are*. In IDL, **print,a** is the same as **PRINT,A** or **print,A**.

III. Vectors and Arrays

Quite possibly the single-most powerful aspect of IDL, from the perspective of a scientific programmer, is the way it handles arrays of numbers. Simply put, it treats an array as a single variable. If you want to divide one image by another, you just type result=image1/image2 and IDL figures out the rest. This may not sound so impressive to you if you haven't struggled through writing loops in other languages to operate on each number in the array one at a time and then put the resulting numbers in the right spot in the resulting array (after declaring all the variable names and data types). All of the data types that apply to single variables also applies to arrays. A 1-D array (i.e. a column of numbers) is called a "vector". Mostly in astronomy we deal with 1- and 2-dimensional arrays. A spectrum, for example, is a vector, and an image is a 2-d array (which we'll just call an "array" from here on, because everything you'll learn also applies to arrays of any dimensions, but we seldom use them).

You can also string multiple operations together in a single command line; you don't have to define intermediate variables or read and write disk files. Of course, you can if you want to; IDL has the power. But if you don't need to, it uses less memory and runs faster to operate on your variables inside a command line. You'll see some examples below.

If a vector called charlie is 25 elements long, the first element is charlie(0) and the last is charlie(24). In 2-dimensional arrays, you can think of the index values as rows and columns. For example, in a 1024 x 1024 image that you called fred, fred(512,736) is the value of the pixel in the 512th row and 736th column. Each row and column of the 2-dimensional array can be treated as a vector by giving a wildcard to the other dimension (e.g. fred(512,*) is the entire 512th row of fred).

First, quit IDL and restart it. This is the only way to completely clear all the junk out of its memory. Then start **IDL** back up and try out the following commands.

```
fltvec=fltarr(360)
help
fltvec=findgen(360)
help
print,fltvec
plot,sin(fltvec*!pi/180.)
silly=sin(fltvec*!pi/180.)
plot,silly,xtitle='theta',ytitle='sin(theta)'
plot,silly,xtitle='!7h !5(degrees)',ytitle='sin(!7h!5)',CHARSIZE = 2.
plot,20*sin((fltvec-.2)*15/180.)
oplot,[0,360],[-.2,-.2]
intvec=[23,97,111,45,-19]
print,intvec
intvec=intarr(10)
read,intvec
 ...  type in 10 random integers
print,intvec
help
for i=0,9 do print,i,'my random number is ',intvec(i)
a=findgen(100)/100.
b=fltarr(100,100)
for i=0,99 do b(i,*)=sin(a*2*!pi)
surface,b
contour,b
```

atv.pro: An interactive display tool for astronomical images written in IDL

Now install `atv.pro` on your laptop. First install the `coyote` idl routines.
I have placed the coyote library and `atv.pro` and `cmps_form.pro` programs on OAKs. Unzip the coyote library in an appropriate directory in the Application folder. Move the `atv.pro` and `cmps_form.pro` programs into the coyote directory.

You now have to let IDL know where to find these programs. You can edit the file startup.pro on your laptop to indicate the path to these new files. Finds the path to startup.pro using `Spotlight` on your laptop.

```
cd /path to startup.pro/
emacs startup.pro

add the following line to startup.pro

!PATH = '/path to the coyote directory/coyote:' + !PATH
```

save and exit the file statup.pro.

Once you have installed it try in IDL:

atv,b
atv,b,min=0,max=2
atv,rebin(b,500,500)
Spend some time here and try out the different features on the atv display. You will likely be using these later on.
print,max(b),min(b),mean(b),median(b)
plot,b(50,*)
plot,b(*,60)
plot,b(*,60),yrange=[-1,0],psym=2

Now try to make up some commands on your own, based on what you've learned so far. Some will work, and some will bomb. Think of something simple you'd like to calculate or plot, and make it work.

Save a digital copy (postscript file) of your graph (or graphs).
Example:
set_plot,'ps'
device,filename='graph1.ps'
surface,b
device,/close_file

Or you can use atv to save the displayed image.

Make sure that you also record in a file called `readme.txt` the commands you used to generate the graph.

IV. Simple IDL Programming

There are multiple ways to "drive" IDL, and that applies to writing programs, too. You can write a one-time-only program interactively. Try this example....

.run
moron=fltarr(50)
for i=0,49 do moron(i)=3*i^2-14*i+14.5
plot,moron
end

The end command is necessary. When IDL encounters it, the program will be compiled and then run line-by-line. If there's an error, it will stop at that line. A program may stop several levels deep in subroutines. To clean up the mess of a crash, and return to $MAIN$ level, type **RETALL**.

Now try running the above commands as a simple script. To do this, use the text editor in the upper right to create a text file with a name convention something like filename.pro. Copy the aforementioned lines into the text file, but replace "**.run"** with "**pro filename".** Make sure to save the file.

Now, from the IDL prompt, you can either type **.run filename** (without the .pro) or just type the filename. They do slightly different things**. .run** always recompiles the program. If you've made changes to it since it was last compiled (e.g. if you fixed an error), this will compile and execute the new version. If you want to compile it the first time, all you have to do is type the name without the .run.

After the pro line, it is customary to include a comment section starting with a line that has just a **;+** on it and ending with a line that has only **;-** on it. All the lines in between should have **;** at the beginning so that IDL knows not to treat them as commands. You can see an example of this in the juldate.pro program that we will be examining soon. The aforementioned section is just comments, but if you type doc_library,procedurename, those comments will print out on the screen. Many procedures are written flexibly, so that if you call them wrong or with the wrong number of parameters, they print out instructions on what to do and die gracefully. As you'll soon see, there are many ways to send parameters and variables between programs. Some parameters are required, some are optional. Some are global (the system variables) and some are only known to the subroutine. Some are changed in the subroutine then passed back to the calling routine. I know this sounds obscure now, but we'll look at a few examples and you'll soon get the hang of it.

Julian date

Find where the program juldate.pro is located on your laptop using findpro. The look at the program juldate.pro using the IDL gui by clicking open and selecting it from the directory you just found using findpro or you can type

more /path_to_juldate/juldate.pro

What is the julian date right now? See if you can figure out how the program works, and calculate a number. What was the Julian date when you were born? How many days have your lived. Do all this in IDL. Record your answer, to be submitted later.

Even when you have careful records in your science notebook, you will inevitably at some point come across a folder on your computer where there are programs that

look useful to you, but you can't remember what they are used for. To avoid this confusion, write a text file called readme.txt and place it in the folder with your programs. The readme.txt file should include your name, the date, and a brief description of the juldate.pro script, the program file you created before that, and anything else you feel is relevant. This text file serves as an explanatory note that will help keep your programs organized and clear. For the purposes of this assignment, please record there the answers you concluded from using the juldate.pro program, as well as the commands you used to generate your previous plot.

RGB "true-color" images

Downlowd FITS files of an astronomical object taken through red, green and blue filters. Here is a suggestion, but you can use a different object:

https://www.spacetelescope.org/projects/fits_liberator/antennaedata/

ATV can make RGB "true-color" images in a rudimentary way. To do this, first set the colormap to grayscale, and don't invert the colormap. Load your "R" image and get the color stretch just the way you want it, and then save it in blink channel 1 with "SetBlink1". Do the same for your G image in blink channel 2 and your B image in blink channel 3. Then, just select "Blink->MakeRGB" and ATV will make a truecolor RGB image using the 3 blink channels. The color image is not adjustable in any way: if you want to change the color balance, the only way to adjust it is to re-load in your original images, change the color stretch, and then save them to the blink channels again.
If you like how your RGB image came out, you can save it to a file using "File->WriteImage" to save it to a jpg, png, or tiff image.

The RGB image will stay in the ATV display window until you do anything that changes the display settings, like zooming or changing the color stretch. As soon as you do that, ATV goes back to its normal display mode with the last image that was loaded.

Assignment 2

2.1
Write an IDL program on your own that calculates some function of your choice and make a graph (a postscript file) of that function with labels and units. Include in your readme.txt file information about the function and idl code. Please upload your idl program, graph and readme.txt file to the OAKS Dropbox folder labeled Assignment-2. The Dropbox selection can be found on the OAKS course page under the 'Grades' menu option.

2.1
Upload your idl program that calculates the days since you or someone else was born, and your readme.txt file (only use one readme.txt for question 2.1,2.2, and 2.3) in the OAKS Dropbox folder labeled Assignment-2.

2.3
Upload to OAKs your RGB true color image of an astrophysical source observed through red, green and blue filters and include in the readme.txt file information about the object and a brief summary of how you created the RGB image.