

QUICK IDL TUTORIAL NUMBER ONE

February 18, 2010

S. Stanimirovic, based on C. Heiles' notes

Contents

1	INTRODUCTION	2
2	THE VERY BASICS: Basic Commands, Constants and Arrays	2
3	THINGS THAT YOU REALLY WANT TO KNOW	4
3.1	Fundamental mathematical constants	4
3.2	The hypertext HELP facility	4
3.3	Batch files	5
3.4	I GOOFED!	6
4	MAKING PLOTS	6
4.1	Specifying data ranges, titles, etc	6
4.2	Overplotting	6
4.3	Making a Hard Copy on Paper	7
5	IDL SAVE FILES	7
6	OPERATORS	8
6.1	The < and > operators	8
6.2	Relational operators	8
6.3	The hugely important WHERE	8
7	FOR LOOPS: USING THEM AND AVOIDING THEM	8
8	PROCEDURES, FUNCTIONS, AND MAIN PROGRAMS	9
8.1	Procedures and functions	9

8.2	DOCUMENTING your procedures; and reading others' documentation	9
8.3	Main programs	9
9	ORGANIZATIONAL STRUCTURES	10
9.1	SCALARS	10
9.2	VECTORS	10
9.3	ARRAYS	10
9.4	STRUCTURES	11
9.4.1	Anonymous Structures	11

1. INTRODUCTION

This tutorial provides only those few things you need to get started with IDL and data reduction. IDL is far more powerful than you would guess from this tutorial and is widely used for scientific data analysis and presentation. As an example, IDL is used at the basic platform for handling all Arecibo observations including those from on-going large ALFA surveys.

In our tutorials, commands you enter are in **bold type**, our comments are in ordinary type, and when we refer to variables in our comments the variables are in *italics*. Go through the following steps.

2. THE VERY BASICS: Basic Commands, Constants and Arrays

print, 3*5 This command prints the integer 15, the result of 3 times 5. Integer numbers have no decimal point.

A = 3*5 Creates the variable *A* and sets it equal to 15.

help, a Tells you about the variable *A*. In IDL, uppercase and lowercase are identical, so *A* is the same as *a*.

HELP, A Identical to the lowercase command above—IDL doesn't care about case.

a = sqrt(a) & help, a Redefines *A* to be the square root of its previous value and also tell about *A*. Typing **&** allows you to put a second (or more) command on the same line. Note that, now, *a* has a decimal point. Numbers with decimal points are called *floating point* numbers.

a = 4.3 Defines *a* as a floating point number equal to 4.3.

a = 'Joe' defines a as a string variable—that is, ordinary text—and sets it equal to “Joe”.

a = [1,2,3,4,5,6] Make A a six-element array containing the integer values 1 through 6.

print, a, 2*a prints the array A and, also, two times A .

b = sqrt(A) Creates a new array, b , in which each element is the square root of the corresponding element in a .

c = a ^ 0.5 Creates a new array, c . The \wedge symbol is how you raise something to a power. We’d better have $b = c$; test it by typing

print, max(b-c) & print, min(b-c) Prints the maximum and minimum values of the array $b - c$. Both b and c are 6-element arrays, so their difference $b - c$ is also a 6-element array.

print, total(b) Prints the sum of all elements in b . You’ll need this for statistical analysis.

a = ftarr(100) Define a as an array of 100 floating point numbers, each element of which equals zero.

a = findgen(100) Define a as an array of 100 floating point numbers in which the values increase sequentially from 0 to 99.

print, a[0], a[99] Print the first and last elements of a . Elements of an array are designated by a numerical index. The index begins with zero; the last element of the array has index $n - 1$, where n is the number of elements in the array.

print, a[10:20] Prints a array elements $10 \rightarrow 20$. Such printouts are often handy but it’s hard to identify each array element with its index. To get a printout in *column format*:

for nr=0,99 do print, nr, a[nr] This uses a *for loop* to cycle through the indices numbered $0 \rightarrow 99$ and print a separate line with two numbers, the index number and the corresponding array element.

b = sin(a/5.)/exp(a/50.) Defines the new array, b , in which each element is related to the corresponding element in a by the mathematical expression. *Note* that when we divide anything by a number we *always express the denominator as a floating point number*. *Always do this* until you learn more.

plot, b Make a plot of b versus its index number.

plot, a, b Make a plot of b versus a , with a on the horizontal (x) axis and b on the vertical axis.

z = ftarr(3,7) Defines a new, array, z , as a two-dimensional array with 3×7 elements.

help, z Tells about z .

3. THINGS THAT YOU REALLY WANT TO KNOW

3.1. Fundamental mathematical constants

A number of mathematical constants are stored in IDL’s “internal variables”, which are characterized by the first character being “!”. Here’s just a few:

print, !pi (yep! this is just π)

help, !dtr (degrees times !dtr gives radians)

print, !radeg (radians times !radeg gives degrees)

There’s no variable for e , the base of natural logarithms; to get its numerical value, you have to take e^1 :

print, exp(1) (base of natural logarithms)

3.2. The hypertext HELP facility

Hypertext on-line documentation is provided by IDL’s *HELP* facility. To access *HELP*, type
?

and after some huffing and puffing a hypertext window will come up. Type in something under “Look For”.

For example, in the following pages we will use the random number generator *RANDOMU*. Type *randomu* into the Look For box and you will see several entries. To generate random numbers we will use the *RANDOMU function*, so single click on that; it will highlight “RANDOMU function”. Click on that.

In the documentation display, the first portion gives information about the function or procedure. Then it defines the required input parameters. **RANDOMU** needs two input parameters, the “seed” and the number of random numbers to generate. (All random number generators use an input number, called a “seed”, to begin the process of generating random numbers; they have to start somewhere!). In IDL, if you don’t specify the numerical value of the seed, it sets *seed* equal to the time from the system clock, which means the numbers differ each time you call **RANDOMU**. For example, with **RANDOMU** if you want 230 numbers distributed randomly between 0 and 1, type

output = randomu(seed, 230)

This generates a 230-element array called *output*.

help, output & plot, output

3.3. Batch files

Suppose you have entered a series of commands and want to repeat the series, perhaps after making either small or large modifications. More typing! But you *don't have* to do all this typing!

Create a file containing this series of commands. This is called a batch file, which is simply a file that contains the list of IDL commands you wish to run. Once you have generated the file with a UNIX text editor (e.g., `textedit` or `emacs`), you invoke it in IDL using the `@` symbol.

For example, consider the following series of commands:

```
original = sin( (findgen(200)/35.)^2.5)  
original = original + 2  
time = 3 * findgen(200)  
plot, time, original, xtitle="Time", ytitle="amplitude", $  
yrange=[0.5, 3.5], xrange=[0,600], ystyle=1, psym=-4
```

In the above, the dollar sign `$` is a continuation character, meaning that the line is continued on the next line.

Now, you could type each of these commands on the IDL screen. But you could also put them in a batch file called “`test.idlbatch.pro`”, or some other pet name¹. *DO THIS NOW!!!*

Then invoke the commands by typing in IDL (you don't need to type the “`pro`” at the end)

```
@test.idlbatch
```

and you can then edit the file and re-invoke it at your pleasure. Saves huge amounts of time!

One important point. As you write software you'll create dozens, if not hundreds, of files containing software. You need to annotate those files and explain what you've done so that, when you come back a day or week later, you can decipher what you've done. You can insert a comment in any IDL software file by preceding the comment with a semicolon. For example, you ought to insert at the very beginning of *test.idlbatch* the comment

```
;This file was made for tutorial number 1 at the  
;idiotic insistence of the professors involved
```

or something to that effect. You can also insert a semicolon anywhere in a line and the rest of the line will be ignored, e.g.

¹It's best to append the abbreviation “`pro`” to the name of any IDL procedure, function, run-time procedure, or batch file. The reason: IDL assumes that you do so, and it allows IDL to find your routines in the IDL path.

original = original + 2 ;We could instead have added 3

Heed the voice of experience: *You can't have too many comments!*

3.4. I GOOFED!

Sometimes we goof, and sometimes this puts IDL into some sort of bad state—usually doing something that seems to take forever. If you could only STOP it!

You *can* stop it. To interrupt any IDL command or program type

CTRL-c

which means: hold down the *Control* key and type the letter “c”. (This works on UNIX system commands, too). In IDL, after you’ve done this it sometimes leaves IDL within a procedure, and you need to get back to the main level by typing

retall

which means “return all”—get out of all procedures and go back to the main level. Whenever things look weird in IDL, type **retall**.

4. MAKING PLOTS

4.1. Specifying data ranges, titles, etc

It’s easy to make beautiful plots in IDL. First, generate the batch file as discussed above (§2.4) and run it; you see the plot. It has the x- and y-ranges you specified—titles too! Those aspects are specified by the keywords in the plot procedure. The **psym = -4** keyword puts each point on the plot as a diamond; try it with **psym = +4**, too—and **2** and **0**, too (**psym = 0** is equivalent to not specifying **psym** as a keyword). The plot procedure has *lots* of keywords. For the documentation, use the *HELP* facility! In addition to those we’ve introduced above, take a look at *linestyle*, *title*, and *ystyle*.

4.2. Overplotting

Often you want to plot two graphs on the same plot—comparing the data with a theory, for example. Just to illustrate this, suppose you want to compare your current plot of *original* with what you’d get by changing the 2.5 power to 2.0. You can do this by:

original_2 = 2 + sin((findgen(200)/35.)^2.0)

oplot, time, original_2, linestyle=2

which will overplot *original_2* using a dashed line.

4.3. Making a Hard Copy on Paper

You do this by creating and then printing a PostScript file. There are several ways to do this. We recommend the following procedure:

1. Using a batch file, edit it so that invoking it makes your plot look good on the screen.
2. Use the procedure `psopen` to open a PostScript file. This takes as an argument the name of the PostScript file; you can also specify the size of the plot (the default is 8.5 by 11 inches). For example, to make a 5 by 4 inch plot in the file called `plotfile.ps` you'd type

```
psopen, 'plotfile.ps', xsize=5, ysize=4, /inches
```

3. Invoke your batch file. Because you called `psopen`, your plot will be written to the postscript file instead of to the screen.
4. Close the poscript file by typing `psclose .`
5. It looks a bit different on the paper than on the screen. Before wasting a piece of paper, look at the ps file to make sure you like it. You can use one of two programs for this (from the UNIX prompt):

```
xv plotfile.ps
```

or

```
gv plotfile.ps
```

6. Once you like it so much you are willing to spend a sheet of paper, print it from the UNIX prompt with the command `lp plotfile.ps .`

5. IDL SAVE FILES

You can easily write IDL variables to disk and retrieve them during a later session by using the `save` and `restore` commands. See IDL's online help..

6. OPERATORS

6.1. The < and > operators

Suppose **a** and **b** are two arrays. The statement **c=(a < b)** sets the new array **c** equal to either **a** or **b**, whichever is smaller. Ditto for **>**, except it's whichever is larger.

Example: suppose you want to plot an array **a**, but restrict the range to the range $(-1 \rightarrow +1)$. You could either use the **yrange** keyword or you could type **plot, a < 1 > (-1)**.

6.2. Relational operators

Suppose **a** is an array. The statement **c=(a eq 5)** sets the new array, *c*, equal to 1 for those elements where **a** is equal to 5 and zero elsewhere. In place of **eq**, you can write **ne**, **lt**, etc. See IDL's **?relational operators** help.

Why would you want to do this? Suppose **a** consists of angles that are in the range $0 \rightarrow 2\pi$ and you want to put them in the range $-\pi \rightarrow \pi$. The easy IDL command is **c = a - 2*!pi*(a gt !pi)**

6.3. The hugely important WHERE

Suppose you have an array **a** and you want to identify the indices of that array for which the elements exceed 10, say. They are given by **indices = where(a gt 10)**. Then **b=a[indices]** contains only those elements. This is great for finding bad data points!

7. FOR LOOPS: USING THEM AND AVOIDING THEM

For loops are handy because you can repeat things easily and automatically; same for While loops. However, they are painfully slow. You should avoid them when possible. And IDL provides some very powerful tools to replace their use; specifically, the operators discussed above allow you to avoid the for/if combination that is so often a part of Fortran and C.

However, sometimes you really do need to use loops. The example below is a case in which a loop was *not* necessary; one could simply write **a=indgen(6)** and **b=indgen(6)^2**. To learn how to use loops, see IDL's **?for** help. Contrary to popular misconception, you can use a **for** loop in a batch file, but you have to put the special characters **&\$** at the end of each line to tell IDL that the statements are in a group. Example:

```
for n=0 to 5 do begin &$
```

```
a[n]=n & $  
b[n]=n^2 & $  
endfor
```

8. PROCEDURES, FUNCTIONS, AND MAIN PROGRAMS

8.1. Procedures and functions

Often you find yourself invoking a specific calculation again and again. In this case, you should define a *procedure* or a *function*. We cannot overemphasize the importance of breaking down your code into small segments, each of which is defined by a procedure or function that resides in a separate file, with each one thoroughly checked so that there is absolutely no doubt about its reliability. This is called *modular programming*, and unless you get into the habit you'll find yourself dealing with undocumented, unreadable, unmodifiable software files that are hundreds of lines long.

8.2. DOCUMENTING your procedures; and reading others' documentation

IDL provides an easy way for you to document any procedure or function that you write. To see how, look for **doc_library** under IDL's hypertext help. This command also gives you the documentation for any procedure for which documentation has been provided; for example, all of the Goddard library's procedures are documented in this way.

If you write a procedure and don't document it, you might as well forget it—because you *WILL* forget it!

8.3. Main programs

A main program is exactly like a procedure, residing in a separate file, except that there is no procedure statement. This makes it very much like a batch file; you invoke it from the keyboard by typing **.run batchfilename** (instead of using the @ symbol). However, there are crucial differences: the main program doesn't need any special symbols in loops, and it must have an *end* statement. When developing a procedure, it is often handy to work with it as a main program.

9. ORGANIZATIONAL STRUCTURES

9.1. SCALARS

A scalar is just a single number. For example, a string scalar is `joename= 'joe'`.

9.2. VECTORS

A vector is a one-dimensional array. For example, a three-element vector of names is `three-names = ['joe', 'ivan', 'mark']`.

9.3. ARRAYS

IDL handles arrays up to 8 dimensions, i.e. with 8 subscripts. Arrays with two subscripts can be mathematically treated as matrices using the `#` and `##` operators, and various matrix manipulation routines; see IDL help under **matrices** and **matrix operators**. You create vectors and arrays using, for example, the `fltarr` or `findgen` commands (for floating point numbers; equivalent commands exist for all variable types). You populate them as appropriate, but try to avoid using for loops; instead, use **where**, appropriate use of the `*` operator, etc.

IDL provides a great deal of flexibility in using subscripts to address particular array elements, and this flexibility is what makes IDL so useful. For example, consider a two-dimensional array `a=findgen(100,100)`. Then:

```
b = a[ 23:25, 67:69]
```

makes `b` a 3×3 2-d array equal to `a`'s array elements in the little box specified. The combination

```
indx = where( a gt 10.)  
b = a[ indx]
```

makes `b` a 1-d array equal to the elements of `a` that are larger than 10. The combination

```
indx = where( a gt 10.)  
jndx = where( a[ indx] le 100.)  
b = a[ indx[ jndx]]
```

shows that you can subscript arrays with other arrays, and makes `b` equal to a 1-d array equal to the elements of `a` that are both larger than 10 and less than or equal to 100.

9.4. STRUCTURES

Structures are immensely useful for any project in which data of different types are related. For example, if you have a catalog of stars with positions and reddenings, you can put the whole catalog in a structure array in which each element of the array contains many quantities such as the name and position. Arecibo data files contain data but also detailed information about telescope pointing, frequencies, etc. and are read into IDL structures. So, we will be working with structures a lot!

There are two types of structure, *anonymous* and *named*. In our experience, anonymous structures are usually preferred.

9.4.1. Anonymous Structures

Our abovementioned star catalog has several different data elements. We group them all into a *structure*; each data element is called a *tag*. Let's call the structure *A*. Then we define the structure tags with the statement

```
A = {name: 'alpha ori', ra:5.3345, dec:-7.6568,reddening:fltarr(12)}
```

We just made one structure, *a*. Now if you type `help,/struct,a` or `help,/st,a`, you will see on the screen

```
** Structure <752268>, 4 tags, length=72, data length=72, refs=1:
NAME          STRING    'alpha ori'
RA            FLOAT      5.33450
DEC          FLOAT      -7.65680
REDDENING    FLOAT      Array[12]
```

The number <752268> is arbitrary and will change from one anonymous structure to the next. The fact that a *number* appears instead of a name means that *a* is an *anonymous* structure. It has four tags: the name, the two positions, and 12 different measurements of reddening. In IDL you refer to a particular tag, e.g. *dec*, by typing `a.dec`:

```
print, a.dec
```

You could change things by typing

```
a.dec = 5.5
a.reddening[3] = 0.7
```

Or you could could populate the 12 reddening measurements by typing, for example,

```
a.reddening = [1.2, 1.4, 1.3, 1.6, 1.3, 1.4, 1.3, 1.3, 1.6, 1.3, 1.4, 1.3,]
```

You can redefine the tag names and number of tags in an anonymous structure—but not for a named structure...