# QUICK IDL TUTORIAL NUMBER ONE
**January 28, 2011**

Carl Heiles

**Contents**

This tutorial provides only those few things you need to get started and prepare your first lab report. IDL is far more powerful than you would guess from this tutorial, and as the course evolves you will experience some of this power.

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.

## 1. THE VERY BASICS

### 1.1. Basic IDL 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 ∧ 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, array_equal(b,c)** Prints 1 if the arrays are equal and 0 if they are not. For success, the arrays must not only have equal elements, but equal length.

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

**a = fltarr(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. Why doesn't the index run from $1 \rightarrow 100$ instead of $0 \rightarrow 99$? You'll appreciate why... later in the course.

**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 = fltarr(3,7)** Defines a new, array, $z$, as a two-dimensional array with $3 \times 7$ elements.

**help, z** Tells about $z$.

### 1.2. Matrix operations

A matrix is just one step away from a 2-d array. IDL provides all of the standard, and many sophisticated and advanced matrix, operations. To multiply two matrices you use the # operator. Thus, the matrix product **C** of two matrices **A** and **B** is **C** = **A # # B**—or **C** = **A # B** depending on...a tricky little point about matrices having to do with row-major or column-major formats. You need to be aware of this if you will use matrices; see §10.

## 2. THINGS THAT YOU REALLY WANT TO KNOW

### 2.1. 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 $\pi$)

**help, !dtor** (degrees times !dtor 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)

### 2.2. 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*.

Next in the documentation comes the list of keywords, which are optional. With **RANDOMU** you can not only generate numbers that are distributed uniformly, but also with other distributions. For example, you can get a normal (also called "Gaussian") distribution by setting the *normal* keyword equal to unity:

**output = randomu(seed, 230, normal=1)**

or, equivalently when a keyword is equal to unity,

**output = randomu(seed, 230, /normal)**

After all this, the help facility provides an example, and then it gives related items that you can click on if you wish.

So why are you just sitting there? Generate an array of random numbers, as in the last line above for example, and plot them with

**help, output & plot, output**

## 2.3. Command-line editing

By now you might be sick of typing. Typing involves making mistakes and then retyping the whole line with yet different mistakes; or perhaps wanting to enter a command that differs just a bit from a previous one. The most important editing commands are listed when you start IDL with our startup file:

| | |
|---|---|
| **arrow keys** | move the cursor as you'd expect |
| **Ctrl-d** | deletes the character under the cursor. |
| **backspace** | deletes the character behind the cursor. |

| | |
|---|---|
| **Ctrl-e** | moves the cursor to the end of the line. |
| **Ctrl-a** | moves the cursor to the beginning of the line. |
| **Ctrl-k** | deletes the the rest of the line. |

Sometimes, when command-line editing, you inadvertently hit **Ctrl-s**; this prevents the cursor from responding to your keystrokes. If you encounter this condition, type **Ctrl-q**, after which things will work normally again.[1]

****Important Caveat!!!**** **Ctrl-d** will knock you out of IDL unless you redefine its meaning with the statement

**define_key, /control, '∧D', /delete_current**

You should put this statement in your idl startup file. If you use my startup file, it's done; see §4.

Finally, a very useful feature is the caret key (∧). If you press this, IDL will ask you for a *String To Match.* Type in a string and IDL will find the previous command in its memory buffer in which this string appears. This is great if you need to get a command that you entered a hundred lines ago.

### 2.4. 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

[1]This behavior is the default in UNIX and Linux; it allows you to start and stop output to your workstation by pressing **Ctrl-s** or **Ctrl-q**, respectively. Most people find this pretty annoying and if you'd like to turn this "feature" off, just include this line in your `.cshrc` file: `stty -ixon`

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 name[2]. *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.

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

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

## 2.5. 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

[2]It'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.

things look weird in IDL, type **retall**.

## 3. MAKING PLOTS

### 3.1. 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*.

### 3.2. 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. If you want to plot just a single point, you can use **plots**.

### 3.3. 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 (which is described in more detail in our memo *Making, Printing, and PowerPointing PostScript Files of Graphs and Images*):

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
   ```

One important option for PostScript files is using PostScript fonts, which are really nice; and, also, using thicker lines. To accomplish this the easy way, replace `psopen` with our wrapper procedure `ps_ch`:

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

3. Invoke your batch file. Because you called `psopen` or `ps_ch`, your plot will be written to the postcript file instead of to the screen.
4. Close the poscript file by typing `psclose`; or, if you used `ps_ch`, type `ps_ch, /close`.
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` .

## 4. STARTUP FILE

The IDL startup file enables you to customize IDL by invoking a set of IDL commands when you begin your IDL session. If you have a good startup file, you can do things like type

**plot, x, y, color=!red**

and the plot will come out in red. If you use my startup file, you have these features and the colors include red, green, blue, cyan, magenta, yellow, white, and black—plus orange, forest, purple, and gray. If you're on the *astro* network, to use my startup file define this alias in UNIX (put it in your `.cshrc` or `.idlenv` file):[3]

```
setenv CARLPATH  /dzd2/heiles/
setenv IDL_STARTUP   /dzd2/heiles/idl/gen/idlstartup_ay204
```

[3]The `.cshrc` file is like the IDL startup file, but for Linux/UNIX: it enables you to invoke a set of Linux/UNIX commands when you open a terminal window.

and if you're on the *ugastro* cluster then it's

```
setenv AY121PATH    /home/global/ay121/
setenv IDL_STARTUP   /home/global/ay121/idl/gen/idlstartup_ay121.pro
```

What's happening here: The first statement creates a Linux/UNIX environment variable that names the origin of the tree that contains the IDL procedures; this environment variable is used in the IDL startup file. The second statement tells the path and name of the IDL startup file. Using environment variables is a powerful way to have the same IDL environment on more than one computer.

## 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. READING AND WRITING FORMATTED DATA ONTO DISK

Formatted data files are text files of the sort made by a text editor. You can read such files that you typed in by hand; you can generate such files in IDL.

### 6.1. Writing a file

Suppose you want to save the above arrays *original* and *original_2* by writing them into a disk file. This is a three-step process:

**openw, 1, 'original.dat'** Opens logical unit number 1 for writing (the "w" in "openw" means "write") and equates it to the filename "original.dat".

**printf, 1, original, original_2** *Printf* is like *print*, but the "f" on the end tells IDL to write the data to logical unit number 1 in exactly the same format you'd see on the screen with *print.*

**close, 1** Closes logical unit number 1.

To test this, make a directory from the UNIX prompt; you should see the file name *original.dat.* Also, print the file from the UNIX prompt by typing

*more original.dat*

You see all those numbers...first the 200 numbers of *original*, then the 200 numbers of *original_2*.

Quite a jumble, and you can't easily compare the two variables or read off what the value of a particular array element is. It's cleaner to write the numbers in *column format* using a *for* loop, by substituting for the above **printf** statement the following:

**for nr=0,199 do printf, 1, nr, original[nr], original_2[nr]**

Try it!

## 6.2. Formatting print statements

**print** and **printf** don't necessarily give you what you want: you might want more or fewer decimal points, for example. You can use these in conjunction with the **format** keyword; see IDL's **?explicitly formated I/O** help.[4]

## 6.3. Reading a file IN COLUMN FORMAT

When you enter data by hand or write them out with a *for* loop as above, you have *column format.* There's an easy way to read column-formatted data. In the above example, we have three columns. Suppose we want to read these columns into arrays named $a$, $b$, and $c$. Then just type

**readcol, 'original.dat', a, b, c**

Try it!!! This is not an IDL routine, but rather exists in the Goddard library. To get documentation on properly-documented non-IDL routines, type

**doc_library, 'readcol'**

`doc_library` is a native IDL procedure; we recommend usinag our `doc`, which is better (as well as being quicker to type!):

**doc, 'readcol'**

Perhaps you are interested in looking at the code, or in importing it to your own directory so you can make some changes for yourself. You can find where the procedure is located by typing

**which, 'readcol**

This works for all code that is written in IDL, including IDL-supplied procedures. But it does not work for IDL-supplied procedures that are *not* written in IDL (such as **plot**, for example).

[4]Instead of typing **?** and working with IDL's help windows for info on **topic**, you can just type **?topic**.

### 6.4. Reading a file in yourself, without using readcol

This is similar to writing a file and requires three statements. You can use free-format or specify the format. See the documentation by typing **?read** and then click on *READF procedure.*

## 7. OPERATORS

### 7.1. 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)**.

### 7.2. 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)**

### 7.3. 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!

## 8. 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
```

## 9. PROCEDURES, FUNCTIONS, AND MAIN PROGRAMS

### 9.1. 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.

### 9.2. 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!

### 9.3. 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.

## 10. ROW- vs. COLUMN-MAJOR: AN IMPORTANT DETAIL ABOUT MATRICES

IDL covers matrix operations completely, from elementary to sophisticated. Before you forage in this territory, you must understand the following details.

In a computer, a multidimensional data set can be indexed in two ways, the *column-major* and *row-major* formats. IDL uses the row-major format, as does Fortran; the other major language, C, uses column-major. Suppose you have a $2 \times 2$ matrix called $A$. In IDL's row-major format, when you type **print, A** IDL prints

$$\begin{bmatrix} A_{0,0} & A_{1,0} \\ A_{0,1} & A_{1,1} \end{bmatrix}, \tag{1a}$$

which is different from (i.e., it's the *transpose* of) what you are used to seeing in standard matrix notation which is the column-major format

$$\begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix}. \tag{1b}$$

We ourselves use the row-major convention such that when displayed in a standard IDL *print* statement, they look correct. That means that our definition is the transpose of the standard one.

There are some matrix operations for which the difference is important. This includes not only multiplication, but also some other operations such as **invert** and **svsol**. IDL almost always assumes that the inputs to these other operations follow our row-major convention.

If you want to be a purist and define the matrices in the standard column-major manner, then go ahead and do so. You then need to do three things. First, if you want to see the matrix displayed in the usual way, then print its transpose by typing **print, transpose(A)**. Second, in all our IDL matrix equations, replace ## by #. Third, check any IDL procedure having a matrix as input to see what it assumes (the default is almost always row-major).

To be specific: if you follow our row-major convention, which is the transpose of the standard one, then the matrix product must be written

$$\mathbf{C} = \mathbf{A} \ \#\# \ \mathbf{B} \tag{2a}$$

while, in contrast, if you follow the standard column-major one then you must write

$$\mathbf{C} = \mathbf{A} \ \# \ \mathbf{B} \tag{2b}$$

Why does IDL do this nonstandard thing? It's because it's more straightforward for image processing, in which traditionally the images are scanned row-by-row (as in a TV set) instead of column-by-column. And IDL's origins are image processing, not matrix math. You might find IDL's convention annoying when you're doing matrix math, but this is more than compensated for by the intuitive feel you gain when doing image processing.