A (Very) Quick Guide to IDL
Version 1.1

Chris North[1]

November 28, 2005

[1]Department of Astrophysics, Oxford Univeristy. Email: cen@astro.ox.ac.uk

1   Introduction

This guide will introduce some useful things to know about IDL. It assumes a basic knowledge of programming and concepts such as conditional statements and variable types. In §2 I will discuss the mainstay of IDL programming: procedures and functions. In §6 I will discuss executive commands, which are used at the IDL prompt. §3 will describe basic syntax, while §8 will suggest and breifly outline some common or particularly useful functions[1]. Many of the procedures and functions do much more than what I have described here, and there are also other ways in which IDL can be used, such as object-oriented programming (the closest to that I've described here is the structures data type). These are described in detail in the Online Help.

I cannot possibly describe every function and procedure in its entirety; there's a very thick manual to do that (with over 4000 pages of procedure and function descriptions and a 160-page index!). This is merely intended to explain what is possible and how to perform basic tasks. For more information and more formal definitions, you should consult the IDL manual. This can be easily accessed by typing *?* at the IDL prompt [2]. To jump straight to help on a particular procedure/function, type *? procedure_name*. The help guide is very formal, which sometimes gets in the way of saying what it means. It uses a few conventions, such as sqaure brackets around optional arguments, which I have tried to use consistently here.

If at any time the explanations here and in the help guide are not useful (which I hope should be rarely), or even just to clarify what has been said, then the best advice is to simply try it out for yourself at the IDL prompt. The beauty of an interpreter is that you can type things line by line, which is useful for playing around and working out what functions do.

If there are any errors in the document, or any other procedures that you think should be described or discussed, then please do not hesitate to contact me. Revisions of this document will probably be "published" as the I learn more about IDL myself

---

[1]The meaning of "useful" is determined by me. . .

[2]If it comes up with an error about the Acrobat Reader not being installed, try again. It should work the second time. Don't ask.

2 Procedures and Functions

Any IDL code you write will be stored in procedures and the functions. While it is possible to type commands into the IDL command line (see §6) one at a time, problems often arise.

Procedures and functions are stored in text files, generally with the file extension *.pro*. The first non-commented line of a procedure/function should be the procedure/function definition; i.e. a procedure should begin with *PRO* followed by the procedure name, while in a function it should be *FUNCTION* followed by the function name. The last non-commented line of either must be *END*. More than one procedure or function (including a mixture if necessary) can be stored in one file, but each must start with a definition and end with an *END* statement. When then file is compiled, it will compiled up to (and including) a procedure/function with the same name as the file (see §6). If required, the procedure/function name can be followed by a list of procedures and keywords, separated by commas (see §2.1). E.g.:

> **PRO** *procedure_name, parameter1, parameter2, . . . ,parametern,*
>   *keyword1=keyword1, . . . , keywordn=keywordn*
> **FUNCTION** *function_name, parameter1, parameter2, . . . ,parametern,*
>   *keyword1=keyword1, . . . , keywordn=keywordn*

Procedures are called with:

> *procedure_name, parameters, keywords*

Functions are called with:

> *Result=function_name(parameters, keywords)*

To tell a function which variable to return as its result, the line *Return, variable_name* should be used. *Result* will then become *variabel_name*. The return statement is described in §8.

2.1 Parameters and Keywords

There are two types of arguments: parameters and keywords. Any parameters have to be given first, and must be in the correct order. Any keywords come after parameters. When the procedure is called, the objects in the call statement are assigned to the corrsponding parameters defined in the program definition, regardless of name, data type, size etc. If there are fewer parameter objects in the call statement than in the definition, then no error

occurs bye default, but the remaining parameters are not defined, which may cause problems later.

To check whether a parameter (or any variable, for that matter) has been defined, the *defined*() function can be used. E.g.:

> **IF defined(***parameter***) THEN** ...

The number of defined parameters can also be checked, with the *n_params*() function. E.g.:

> **IF n_params() NE** *number* **THEN** ...

Keywords are called by name, and can either be set equal to a value (or other object) in the call statement, or simply set to true. E.g.:

> *PROCEDURE_NAME, keyword=value*

sets the object *keyword* equal to *value*. To check whether a keyword has been defined in this way, the *defined*() function can be used as it was for parameters.

> *PROCEDURE_NAME, parameters, /keyword*

sets the keyword *keyword* to true. To check whether a keyword has been set, then the *keyword_set*() function can be used. E.g.:

> **IF keyword_set(***keyword***) THEN** ...

When debugging a program which has halted due to an error, typing *help* at the IDL prompt will bring up a list of defined variables as well as compiled functions and procedures.

2.2   Filenames

When a procedure or function is called, IDL looks in the follwing places.

1. Currently Compiled Procedures

2. Implicit IDL routines

3. *.pro* files with the same name in the current working directory

4. *.pro* files with the same in any directory in the path

5. If the function cannot be found, then IDL assumes it is an array reference.

It should be noted that when IDL looks for *.pro* files on a Linux system, **it only looks for entirely lowercase filenames**.

This list also has implications for array references. If an array is referenced with parentheses i.e. *Array(i,j)*, then problems will occur if a function is created with the same name, as IDL will find this before it even considers looking for an array. The ways around the problem are:

- Do not give functions common array names e.g. *value* or *result*.

- Only refer to array elements with square brackets[3].

3   General Syntax Guide

The general systax of IDL is similar to most other programming languages, and the basic structure is the same (it has data types, modules, etc.). As with, for example, FORTRAN, code such as *If* statements, *FOR* loops and *WHILE* loops, can be used. However, the exact syntax may differ. One difference is that IDL is effectively an interpreter, rather than a compiler. This means that, for example, an *IF. . . THEN* statement will assume that it operates on a single line of code unless it know otherwise.

Variable names follow the following rules (most of which are normal):

- Variable names must not start with a number.

- IDL is case insensitive, so capitalise however you like.

- Variable starting with a ! are system variables, and so are available to ALL procedures and functions (see §7).

- Procedures and Functions can only use variables which are passed when they are called (although the local and global names can differ). All other variables must be defined within the procedure/function.

Declarations are not normally necessary in IDL. Where they are necessary (e.g. when reading in a value which is not a float—see §3.9), the variable

---

[3]The IDL compiler can be setup to not accept array references using parentheses

can be defined simply by setting it equal to a value of the appropriate data type (see §3.7. E.g. *A=0* would mean *A* is an integer, *B=0L* would mean *B* is a long integer, *C=0.* would mean *C* is a floating-point number, *D=0.d0* would mean *D* is a double-precision floating-point number, and *E="* would mean that *E* is a string.

3.1   Blocks

To run a statement, such as *IF. . . THEN* or *WHILE*, on a block of code, rather than a single line, the *Begin* and *END* commands should be used. For example:

> **IF** *condition* **THEN Begin**
> > *. . . block of code*
>
> **ENDIF ELSEIF** *other condition* **THEN Begin**
> > *. . . more code*
>
> **ENDELSE**

Note that if an *Else* is required, the initial *If* has to be ended before the *Else* can begin. If only one line of code is required, then the following systax can be used (all on one line):

> **IF** *condition* **THEN** *something* **ELSE** *something else*

The same thing can be used with *For*, *While* etc. (see §3.3).

3.2 $ and &

Multiple lines of code can be written on one line of text using the & operator. For example: *A = B + C & Print, A* would add *B* and *C* together into *A* and then print *A* to the screen (although in this case, *Print, B + C* would have the same direct result. This is useful, for example, when two short commands are required for an *If. . . Then. . .* statement.

The $ operator is the counterpart of &. It tells IDL that the next line of text is actually part of the same line of code. This is useful when writing long individual lines, such as *PRO* or *FUNCTION* statements with a long list of parameters and keywords, or for a long *If. . . Then. . . Else. . .* statement (though *Begin...End* blocks probably make more sense here).

3.3 Conditional Statements and Loops

The *If. . . Then. . . Else. . .* statement was introduced above. The other main type of conditional statement is the *Case*, which has a syntax of:

CASE *expression* **OF**
    value1: statement
    value2: statement
ENDCASE

Each statement can be a single line or a block of text. If required, a case of *Else: statement* can be added in at the end.

The available loops are *For. . . Do*, *While. . . Do* and *Repeat. . . Until*. They have the syntaxes:

FOR *integer=low,high* **DO** *expression*

OR

FOR *integer=low,high* **DO Begin**
    block of code
ENDFOR

WHILE *condition* **DO** *expression*

OR

WHILE *condition* **DO Begin**
    block of code
ENDWHILE

REPEAT *statement* **UNTIL** *condition*

OR

REPEAT Begin
    block of code
ENDREP UNTIL *condition*

3.4 Conditional Operators

The condition statements described above rely on the conditional operators. The main logical operators are: EQ (equal to), NE (not equal to), LT (less than), GT (greater than), LE (less than or equal to), GE (greater than or equal to), AND, OR and NOT.

If the condition is testing whether a variable or result of a function is true
or false, then it is not necessary to use an operator. A good example is the
Keyword_set() function described above.

3.5   Array Operations

In IDL, arrays are indexed by column number then row number (and then
by any other dimensions which are not possible to describe simply), e.g *Array[x,y]*, which is intuitive when using 2D images. Arrays can be created
manually, e.g.:

   $Result = $ [[1,2,3],[4,5,6],[7,8,9]] would create a 3 by 3 array looking like:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

More levels of brackets would create more dimensions, though it ends up
being tedious. Creating arrays is more efficient using the functions described
in §8.

To refer to an element *[i,j]* of a two-dimensional array *Array*, use the syntax
Array[i,j]. To refer to a range of elements, *[i,k]–[j,k]*, separate the high and
low indices with colons, i.e. *Array[i:j,k]*. An asterisk will refer to an entire
row or column e.g. *Array[*,k]* refers to the whole of the kth row of *Array*,
while *Array[*,*]* refers to every element of *Array*.

Operations on whole arrays are also straightforward. To add a fixed value, *c*,
to every element, then use either *Array = Array+ c* or *Array[*,*]=Array[*,*]
+ c*.

Matrix Multiplication is possible via the # and ## operators. The former
multiplies columns by rows, the latter rows by columns.

The sum of all (or some of) the elements in an array can be acheived using
the *Total* function. TOTAL(*Array*) would return the sum of all the elements
in *Array*, while, for example, TOTAL(*Array[*,k]*) would return the sum of
the kth row of *Array*.

3.6   Mathematical Operators

The basic mathematical operators in IDL are:

=	Assignment
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
++	Increment by 1
−−	Decrement by 1
MOD	Modulo operator
<	Minimise operator
>	Maximise operator

They are all self explanatory, except possibly for the last two. These are used to limit the range a numeric variable can have. For example, $A=A<6$ would leave A alone, unless it is less than 6, in which case it would be set to 6. The same works for arrays: $Array=Array<100$ would set any element of *Array* which is over 100 to 100 and leave the rest alone.

Common mathematical functions are *EXP, SQRT, ALOG, ALOG10, SIN, ASIN, SINH* and so on. All are fairly self-explanatory (asin=arcsin), though it should be noted that all angles are dealt with in RADIANS, which makes the *!DTOR* and *RADEG* system variables very useful (see §7).

3.7   Data Types

As with most languages, every IDL variable has a data type. Each data type has a numeric code which can be viewed with, for example, the *Size* funtion (see §8). The basic data types, and their numeric codes, are:

Code	Type Name	Description
0	Undefined	Undefined variable (i.e. has been referred to but never defined)
1	Byte	8-bit integer in the range 0–255
2	Integer	16-bit integer in the range $-32,769$–$+32,767$
3	Long	32-bit integer in the range ~ -2billion–$+2$billion
4	Float	32-bit floating point number in the range $\pm10^{38}$ with 6 or 7 decimal places
5	Double	64-bit floating point number in the range $\pm10^{308}$ with around 14 decimal places
6	Complex Float	Real-Imaginary pair of floating-point numbers
7	String	List of between 0 and ~2billion characters treated as text
8	Structure	See below
9	Complex Double	Real-imaginary pair of double-precision floating-point numbers
10	Pointer	Not discussed here
11	Object Reference	Not discussed here
12	Unsigned Integer	16-bit integer in range 0–+65,535
13	Unsigned Long	32-bit integer in range 0–~4billion
14	64-bit Long	64-bit integer in range $\sim \pm9 \times 10^{18}$
15	Unsigned 64-bit Long	Unsigned 64-bit integer in range 0–$\sim 18 \times 10^{18}$

Type conversion is possible with the functions *Byte, Fix, Long, Float, Double, Complex, String, UInt, Ulong, Long64, Ulong64*, which are fairly self-explanatory given the data type names above. The oddity is *Fix*. This function converts to integer type, cutting off the decimal points. The keyword *Type* can be set to a type code to store the integer in a different precision variable, if required.

3.8 Structures

Structures are complex data types. A structure has a number of fields, each of which has a unique name. The fields can have different data types. There are two ways to create a structure, a simple definition and the *Create_Struct* finction. E.g.:

$Result = \{$*[Name,]* Tag_1:$Value_1$,*ldots* ,Tag_n:$Value_n$, *INHERITS structure*$\}$

OR

$Result = $**Create_Struct**$(Tag_1, Value_n, \ldots, Tag_n,value_n$ *[, Structures, Name=*$)$

The values can be any data type, including existing structures (which means they become sub-structures). To inherit the tags and values of an existing structure, then the *INHERITS structure* command can be used in the first example, or the inherited structure can simply be inserted as a paramter in *Create_Struct*. If a *Name* (which does not have to be the same as *Result*) is given, then the defined structure (in the eamples above, *Result*) becomes a named structure, with the tags and value data types being unchangeable. With no name given, the structure is "anonymous" and the tags and value types of *Result* can be changed.

To refer to *Value1* of *Result*, then use *Result.Tag1* (although you've hopefully chosen more imaginitive names!). To refer to a named structure definition, then the structure name should be put in curly braces. For example, to give *Result2* the same tags and value types as named structure *Structure1*, then use *Result2 = {Structure1}*.

3.9 Input/Output

When opening a file in IDL, the purpose of opening it has to be known. The procedure OPENR, *unit, filename* will open *filename* (a string variable, including the path) and assign it the code *unit* (an integer) and make it read-only. The command OPENW, *unit, filename* works in the same way, but with write-access enabled. Unless the */Append* keyword is set, then any existing file withn the same path and filename will be erased. The command OPENU, *unit, filename* will open a file for reading and writing. In this case, the current position is at the start of the file, so if anything is printed, it will overwrite what is currently there.

If a variable name is used for the unit, and the keyword *Get_Lun* is set, then
the variable will be assigned the value of the next unused unit number. When
the file is closed, it is often useful to use Free_Lun, *unit* to free up whichever
number is currently stored in *unit*, where *unit* is the variable used above.

The command READ, *var[, Prompt=string]* will read whatever is typed at
the prompt into *var*, with an optional prompt *string*. If it has not previsouly
been defined, the variable *var* is assumed to be a floating-point number. It
cannot be an element (or range of elements) of an array. If *var* is an (entire)
array, then values will be read into each array element until the end of the
array is reached.

To read from a file assigned to *unit*, the command READF, *unit*, $Var_1[,\ldots,$
$Var_n]$ should be used. This read the first item in the file into Var_1, the
second into Var_2 and so on. At the end of the command, the current line of
the file is moved down by one. An error will occur if the end of the file is
reached, so it is useful to know that the function EOF(*unit*) will return 1 if
the end of file assigned to unit *unit* is reached.

To print to the screen, then the command Print, $Value_1[,\ldots,Value_n]$ should
be used. Each $Value_i$ can be any variable type, or the result of a function.

To print to a file, us command PrintF, *unit*, $Value_1[,\ldots, Value_n]$ should be
used. This will print the values to the assigned to *unit*, with a default width
of 80 characters. This means that a long list of values will extend to the next
line, which sometimes causing problems when reading the information back.

When a file is finished with, the command, CLOSE, *unit* can be used with
as many different units as is required. CLOSE, */all* will close all open files.

4   Configuring IDL

Various things may need to be done to get IDL to work for you. For a
start, you will want the appropriate directories in your path. The System
Variable !PATH (see §7) contains the list of directories. A useful function is
Expand_Path. This accepts a single parameter argument: a string containing
a path definition. It returnsa string containing the expanded path. Different
directories should be separated by ':', while putting a '+' infront of each
directory path tells IDL to expand it to include all sub-directories containing
.pro or *.sav* files. E.g.:

!PATH = !PATH + Expand_Path(+/home/cen:+/usr/local/packages/healpix)
would add the directories /home/cen/ and /usr/local/packages/healpix, along with all their subdirectories, with the !PATH system variable.

Another useful command is *Setenv*, which will set environment variables for use later on. E.g:

Setenv, 'Temp=/home/cen/temp'
will create an environment variable $Temp which points to the directory named. Note that there can be no spaces either side of the '=' in the string argument.

Both the *Expand_Path* and *Setenv* commands can be put into a file which is run on startup. Either put them in *.idl_start* or, at the UNIX prompt type *idl path/file.pro*, which will start IDL and immediately run the given file. The latter is useful if several different startup sequences are needed. It should also be noted that *file.pro* should NOT have the intial *PRO* or final *END* lines if is to be run from the UNIX prompt.

5 Using Online Help

There are several types of online help which can be used. As descibed in §1, the *?* command can be used to get help with intrinsic procedures. The procedure *Doc_Library* can be used to get help with user-written files. *Doc_Library* searches the source for a line starting with ';+' (without the quotes). It then displays every commented line (i.e. starting with ';') until it reaches a line starting with ';-'. E.g. for a file containing the lines:

;+
;Some Help
;With Using
;This Program
;-

Doc_Library would output:

Some Help
With Using
This Program

A graphical version of *Doc_Library* is *XDL*. Help on using both of these can be found in the online help (or by using *Doc_Library* or *XDL*!)

6   Executive Commands

Executive commands can only be used at the IDL command prompt.

6.1   .COMPILE and .RUN

IDL will automatically compile a procedure/function the first time it is called, providing it is in the current directory or path. However, it is often necessary (especially when writing or debugging code) to recompile a procedure/function. If the file is in the current directory or path, then the command .COMPILE *procedure/function_name* will (re)compile the function. To compile more than one procedure/function/file, then separate the names with spaces.

There may also be occasions when the filename is either not in the path, or has a different name to the procedure/function called (meaning that IDL will not be able to find it). In this case, *.compile PATH/filename* will compile the entire file.

The .RUN command is very similar in function to the .compile command and works in exactly the same way. However, it has a few extra features available. Adding the modifier *-T* will cause the contents of the file (with line numbers added) to be printed to the terminal before compilation. The modifier *-L listfile* will print the same thing to a given filename.

6.2   .continue, .skip, .step and .stepover

When debugging a program, it is sometimes useful to put the line *STOP* in the code at a key point. This will stop execution and return to the IDL prompt. All variables will remain defined, so it is quite a useful way of checking what's going on. The same thing will happen if an error occurs or a keyboard interrupt (Ctrl-C) is encountered. At the IDL prompt, the command .CONTINUE will continue running the stopped code from where it left off.

The .SKIP *[n]* command will tell IDL to skip *n* lines and continue running. If a number is not given, then only one line is skipped.

The .STEP *[n]* command tells IDL to run the next *n* commands (or just the next command if the number is omitted). If a statement calls a routine, then the counting continues throughout the called routine. The .STEPOVER *[n]* command does the same, but treats calling statements as single lines, and continues couting at the end of called routine.

6.3   .return and .out

To run a program until a *Return* statement is encountered, use the .RETURN executive command. The .OUT command will simply run the current program until it reaches the end.

7   System Variables

System variables have names beginning with a '!', e.g. !VERSION. They have a predefined type and structure which cannot be changed. New system variables can be defined with the *DefSysV* procedure. The default system variables govern a variety of things:

7.1   Constant System Variables

constant system variables are read-only. They include such things as !PI, !DPI (double precision π), !DTOR (conversion factor from degrees to radians) and !RADEG (conversion factor from radians to degrees).

7.2   Error Handling

Variables such as !ERROR_STATE and !EXCEPT tell IDL what to do if an error occurs. !MOUSE contains information about where, when and how (as in which button) the mouse was last clicked.

7.3   IDL Environment Variables

System variables such as !VERSION, !DIR and !PATH govern the particular system environment in which IDL is installed. In these cases, !VERSION contains information on the Operating System and the IDL release, !DIR contains the directory in which IDL is installed, while !PATH contains the current path (see §).

7.4   Graphics System Variables

Various properties of the display are controlled with system variables. !DEVICE contains information about the current device (X-window, PostScript, etc.), !D contains information about the display, while !P, !X, !Y and !Z contain information pertaining to plotting to the device.

8   Some Useful Functions and Procedures

The following are a few procedures which are useful to know about. While all this information is in the IDL help guide

8.1   FltArr, DblArr, etc.

FltArr($D_1[,\ldots,D_8]$) will return an array with up to eight dimensions, of sizes D_i, containing real, single precision floating-type data type, initially set to zero. The optional keyword *NOZERO* will prevent the zeros being inserted and make the function run faster.

Similar functions for different data types are: DBLARR, COMPLEXARR, DCOMPLEXARR, INTARR, LONARR, STRARR etc.

8.2   Findgen, Indgen, etc.

Findgen($D_1[,\ldots,D_8]$) will return a floating-point array of the specified dimensions. Each element contains the value of its one-dimensional subscript.

For example: *Result=*Findgen(3,3) will return *Result* containing:

0	1	2
3	4	5
6	7	8

8.3   Where

The *Where* function returns the one-dimensional indices of the elements of a given array which satisfy a condition. E.g:

$$Result = \text{WHERE}(\textit{Array GT 100 [, count]})$$

will result in an array, *Result*, containing the indices of all elements of *Array* which are greater than 100. The optional parameter *count* will contain the number of such elements. If *Array* has more than one dimension, then the one-dimensional index is the index found by counting along each row from top to bottom (and then moving along in the 3rd, 4th, etc. dimensions if required).

8.4   Size

Size(*Var*) will return a list of numbers in the following format:

$$N_{dim} \qquad [D_1 \qquad \ldots \qquad D_8] \qquad Type_n \qquad N_{elements}$$

Where:

N_{dim} is the number of dimensions (0 for a scalar, 1 for vector, $>= 2$ for an array)

$D_1 \ldots D_n$ are the sizes of the dimensions (absent for a scalar)

$Type_n$ is a number representing the datatype (see §3.7)

$N_{elements}$ is the total number of elements

Various keywords can be used to only return certain information. E.g. */Dim* only returns the number of dimensions, */TName* only returns the name of the data type (in English, not some numeric code).

8.5 Replicate

Replicate(*X, $D_1[,\ldots,D_8]$*) will return a vector or array of dimensions D_i with all elements equal to *X*.

8.6 String

String(*X*) will return a string containing the literal text of *X* i.e. the way it is printed on the screen, complete with and redundant leading spaces (see *StrCompress* below).

8.7 StrCompress

StrCompress(*String*) will return the string *String* with all redundant spaces removed, i.e. the string 'two spaces' would become 'two spaces'.

     Result=StrCompress(*String*, /REMOVE_ALL) will perform the same function, but ALL spaces will be removed, i.e. the string 'one space' would become 'onespace'.

8.8 Return

In procedures, the *Return* statement tells the procedure to return to the program level above, i.e. leave the current procedure at whatever point it's at and continue from where it was called in another program. This is useful if a circumstances mean that there is no point in going any further in the procedure.

In functions, the command Return, *variable* is used when the function has finished. This tells the function to return control to the program which called it, and return *variable*. The last line of any function (apart from the *END* statement) will normally be of this form, though there may be other lines within conditional blocks further up.

8.9 Message

To print a message to the screen in the case of an error or for information, the **Message**, *[Text], [/Continue], [/Informational]* procedure can be used.

The command will look at the current error state and print the appropriate
message, unless the sting *Text* has been specified. IDL will then perform
whatever action is required by the error, which will often result in stopping.
If the keyword */Continue* is set, then IDL will keep on running, regardless
of the error message. If */Informational* is set, the message will be merely
informational, and nothing will be done about errors. The advantage of this
over a simple *Print* statement is that *Message* also displays the procedure
name by default.

9   Plotting

Plotting graphs in IDL is reasonable striaghtforward. The procedure *Plot,
[X,] Y* will plot the vector *Y* against either the vector *X* (if given) or simply
the element index. Various keywords can be given, such as *Title*, to add
things to the graph. The keyword */NODATA* will simply plot the axes.

To plot another line on the graph, then the command *OPLOT* works in the
same way, but does not wipe the original graph.

To plot multiple graphs on one "page", the system variable !P.MULTI must
be changed. !P.MULTI=$[0,a,b]$ will put a grid of *a* by *b* graphs on one "page",
and set the "current" graph to the top left. When a *PLOT* command is
used, the "current" plot steps along the rows from left to right, then top to
bottom. See the IDL Help guide for much, much more information about
plotting, inlcuding using colour (sorry, color) tables.

To change what is being plotted to, use the *Set_Plot* procedure. For example,
SET_PLOT, *'X'* and SET_PLOT, *'PS'* will set the plotting device to the X-
terminal or PostScript respectively. When writing to PostScript files, the
Device command can be used to set the filename, among other things. E.g:
       Device, filename=*filename*[, /color]
will set the filename to write to, with color postscript being enabled by using
the optional */color* keyword.

Other graphics commands, such as *Write_JPEG*, *Write_PNG*, etc. are de-
scribed in detail in the IDL documentation. The procedure *XYOuts* will
allow you to place text in an image. Again, consult the help guide for further
information.