

IDL: Interactive Data Language

User Information

Chris Rennie

March 2005

1 Introduction

IDL can be used to generate high-quality graphics, consisting of 2D plots (line plots) or 3D plots (surface plots), as well as for rendering images. It is roughly comparable to SuperMongo in being an independent application that can be used interactively or by running scripts. [If you need to create graphics from within your own program you should use the PGPLOT or NCARGraphics libraries.] IDL is able to take data files, transform the data in almost any way, and then to generate output either on screen, on paper, or to a file. It is a general-purpose programming language with an emphasis on data presentation.

There is on-line help available from within IDL, however it is not a satisfactory starting point from which to learn about IDL. Consequently this document aims to provide an overview of the essentials, and leaves detailed explanations to the manuals and on-line help.

2 Absolute essentials

IDL runs on most hosts in physics, but it might be necessary or desirable first to `ssh` to some host other than your local host. Then you can either type

```
idl
```

to run IDL through a traditional command line interface, or type

```
idlde
```

to run the graphical IDL Development Environment. The latter offers an integrated editor and easy display of variable values, but the former is better when simply running programs. The choice depends on circumstances.

In both interfaces there is a prompt

```
IDL>
```

at which one can type any of the hundreds of operations, declarations and expressions possible under IDL, and any of the executive commands including

<code>.compile</code>	or <code>.com</code>	Compile but don't run script
<code>.go</code>	or <code>.g</code>	Run previously compiled script
<code>.run</code>	or <code>.r</code>	Compile and run script
<code>.rnew</code>	or <code>.rn</code>	Clear memory, compile and run script
<code>?</code>		Start online help
<code>exit</code>		Exit IDL

Note that detailed on-line help can be obtained by entering '`?`', at the IDL prompt or by running the program `idlhelp` from the shell prompt.

The remainder of this overview section will revolve around the following commands, which can be entered at the IDL> prompt of either IDL or IDLDE,

```
IDL> x=findgen(100)
IDL> x=x/50
IDL> y=sin(3*!Pi*x)/EXP(x)
IDL> PLOT, x, y
IDL> wdelete
```

and which should result in a plot of a damped sine wave.

This simple example serves to demonstrate several general features of IDL. The first line contains a call to the function `findgen()`, which in this case generates an array of 100 floats that is assigned to the variable `x`. This useful function also initializes the array so that its elements are `x[0]=0.0`, `x[1]=1.0`, `x[2]=2.0`, ..., `x[98]=98.0`, `x[99]=99.0`. Note that data types are dynamic, so `x` will remain an array of 100 floats until there is a new assignment that changes the type or dimensionality of `x`.

The second line illustrates simple scaling of the array `x`.

The third line contains two other functions, `sin()` and `exp()`. The meaning is obvious, although it is notable that the functions have the array `x` as their argument, and so the result `y` will be an array with the same dimensions as `x`. The argument to `sin()` contains the predefined constant `!Pi`. This is an example of one of several system variables, which are distinguished user-defined variable by an initial exclamation mark. Note also that IDL is insensitive to case: `EXP` is identical to `exp` and `!Pi` is identical to `!pi`.

The fourth line plots the array `y` as a function of `x`. In the terminology of IDL, `plot` is a *procedure*. Procedures have a distinct syntax from that of functions: they have no brackets around the arguments, and generate no return value. IDL procedures are typically flexible in that often they can be called in alternative ways. For example,

```
IDL> plot, y
```

produces a similar plot, except that the abscissa is labelled with default values, 0 to 100 in this case. The various alternative forms for each procedure call are explained in the manual.

Also, procedures and functions usually allow a number of *keywords*. Keywords can appear anywhere after the procedure or function name. They might be optional parameters, for example

```
IDL> plot, x, y, thick=2.0
```

will cause the line to have double the usual thickness; or they can be switches, as in the example

```
IDL> plot, x, y, /xlog
```

which results in a log-linear plot. This last example generates a warning on account of `x[0]=0.0`, so this is a good opportunity to demonstrate how to specify a subscript range. The form

```
IDL> plot, x[1:99], y[1:99], /xlog
```

will skip the first element, eliminate the warning, and produce a plot with a more satisfactory scale. There is also a shorthand way of specifying the entire range of subscripts within one dimension, which is useful for example in extracting rows or columns from a multidimensional arrays:

```
IDL> plot, y[3,*]           ; plot 4th column of a 2D array
IDL> plot, y[* ,9]         ; plot 10th row of a 2D array
```

The fifth line in the example above, `wdelete`, is a procedure that requires no arguments.

Most data types are available within IDL, including the usual numerical types as well as complex and string data types and structures. Likewise all the usual operators are available (exponentiation uses `^^`) plus some unusual ones (`#` and `##` for matrix multiplication). Boolean operators follow Fortran nomenclature.

To view the contents of a simple variable or an array, use the `print` command, although `help`, `struct`, `\struct` is more suitable for structures, e.g.

```
print, y                   ; prints contents of variable or array y
help, !d, /struct          ; prints system structure !D
```

3 Scripts and complex statements

All but the most trivial plotting tasks are best handled with a script file. The following example shows a complete IDL script, and includes comments, continuations and a FOR loop.

```
; Example of FOR loop and array operations. Note also
; the comment (';') and continuation ('$') characters.
set_plot, 'x'                ; 'x' or 'ps' are common options
TotalPts=40
```

```

MaxWidth=16
; Create and plot a sequence of random numbers
plot, randomn(seed,TotalPts,/normal), $
           title='Smoothed Normal variate'
for n=4,MaxWidth,4 do begin
    Raw=randomn(seed,TotalPts,/normal) ; New sequence
    oplot, smooth(Raw,n,/edge_truncate), linestyle=n/4
endfor
end

```

If the above lines are in a file called `script1.pro`, then to run it type `.rnew script1` following the IDL> prompt. The result will be an X window containing one series of random numbers, overlaid with four increasingly smoothed series.

The command `.rnew` is like `.run` but additionally erases currently accumulated variables, so is preferred. It is also possible to type `.compile script1` and `.go` if you wish to compile and run separately. Note that if you use IDL's conventional file extension `.pro` for scripts then it is not necessary to specify it when running scripts.

One good reason for using script files is that complex control statements are possible. Although control statements have single line forms, they can appear as in scripts as multiline statements by combining them with `begin` and `end`. Both single and multiline forms are illustrated in the examples below.

<pre> for var=val0,val1[,incr] do begin statements end </pre>	<pre> for col=0,MaxCols-1 do begin av=total(arr[col,*])/100 arr[col,*]=arr[col,*]/av end </pre>
<pre> if expression then begin statements end </pre>	<pre> if f lt Nyquist then f=Nyquist </pre>
<pre> if expression then begin statements end else begin statements end </pre>	<pre> if Inverted then begin plot, Time, -Efflux end else begin plot, Time, Efflux end </pre>
<pre> repeat begin statements end until expression </pre>	<pre> repeat begin Pow=2*Pow end until (Pow gt val) </pre>
<pre> while expression do begin statements end </pre>	<pre> while (ans eq 'y') do begin print, systime() read,prompt='Again?',ans end </pre>
<pre> case expression of expression: begin statements end expression: begin statements end endcase </pre>	<pre> case style of 1: plot, y 2: plot, x,y else: begin print,'Illegal option' stop end endcase </pre>

4 Preparing the data for plotting

The first task of generating a plot is usually reading data files and getting the data into a suitable form. The ideal case is when you know beforehand the dimensions of the data. For example, if the file consists of three lines of header information, a ten element integer vector and a ten column by 100 row matrix of floats, then it can be read by

```

openr, lun, 'readings1.dat', /get_lun
header=strarr(3)
ivals=intarr(10)

```

```

data=fltarr(10,100)
readf, lun, header,ivals,data
free_lun, lun          ; close and free LUN
; plot data...
end

```

This example introduces `openr` and `readf`, both of which are very like the corresponding Fortran commands. Like Fortran, IDL has the concept of logical units for addressing I/O streams. They are addressed using arbitrary integers in some limited range, but IDL has the ability (illustrated above) of assigning LUNs automatically. There is no need to choose LUNs explicitly.

It often impossible to allocate array sizes correctly prior to reading the data. In such cases it is necessary to deal with the possibility of overflow, perhaps as in the following example. It is also usual practice to truncate an oversized array to eliminate unused elements. The following example shows how easy this is in IDL.

```

; Read file consisting of four columns of floats
openr, lun, 'readings2.dat', /get_lun
data=fltarr(4, 200)      ; 4 columns, 200 rows
temp=fltarr(4)          ; 4 elements
i=0
while (not eof(lun) and (i lt 200)) do begin
    readf, lun, temp
    data[*,i]=temp
    i=i+1
end
data=data[*,0:i-1]      ; discard unused elements
free_lun, lun          ; close and free LUN
end

```

For subtle reasons to do with the distinction between references and pointers, it is illegal to perform the following shortcut in the preceding example:

```
readf, lun, data[0,i], data[1,i], data[2,i], data[3,i]
```

The rule is that **you cannot read into a subscripted variable**.

The command `readf` is used much of the time as shown above, but also allows formats to be specified explicitly when necessary. This is useful when there are values to be skipped, or strings which need to be parsed explicitly. Some examples are

```
readf, lun, farray, iarray, format='(3f6.2,30x,10i5) '
readf, lun, str, farray, format='(a10,3x,4(e10.3,5x)) '

```

Opening a data files for writing is achieved with `openw`, and formatted writing is carried out with `printf`. There are also commands for reading and writing *unformatted* data, `readu` and `writeu`, and for reading and writing certain image formats like GIF, JPEG etc. Unformatted data produced by Fortran programs are catered for by the keyword `f77_unformatted`. For more, see under 'Input/Output Routines' in the online help.

5 Two dimensional plots

Before plotting it is usually necessary to define the destination for the output, and to specify the size of the plot area. Once that is done, subsequent plot commands are independent of hardware.

5.1 Output device definition

There are many possible output 'devices' in IDL, although the most important ones are an X window, a Postscript page and an Encapsulated Postscript figure. The following script fragments show how each can be set up in such a way as to result in basically similar output. Note the many small differences depending on the choice of output device. [The examples below assume that you have set the variables `x_cm` and `y_cm` to the required plot size in cm – although there are reasonable defaults already set.]

X window:

```

set_plot, 'x'
!p.font=-1          ; vector (Hershey) fonts
device, set_character_size=[7,12]      ; in pixels
window, /free, xsize=x_cm*!d.x_px_cm, ysize=y_cm*!d.y_px_cm
:                  ; create plot
wdelete, !d.window ; optionally delete window when done

```

PostScript file:

```

set_plot, 'ps'
!p.font=0          ; hardware (PostScript) fonts
device, /helvetica, font_size=10, filename='somename.ps', $
      ENCAPSULATED=0, xsize=x_cm, ysize=y_cm, $
      xoffset=(21.0-x_cm)/2, yoffset=(29.7-y_cm)/2
:                  ; create plot
device, /close_file ; close file - obligatory

```

Encapsulated PostScript figure:

```

set_plot, 'ps'
!p.font=1          ; TrueType fonts
device, /helvetica, font_size=10, $
      filename='somename.eps', /ENCAPSULATED, $
      xsize=x_cm, ysize=y_cm
:                  ; create plot
device, /close_file ; close file - obligatory

```

For the purposes of illustration only, all three legal values for `!p.font` are demonstrated above. The choice of font renderer may seem an arcane matter, but can be important if trying to create perspective effects, or consistent appearances on different output devices. In short: vector fonts can be used in both X windows and Postscript files but are ugly and idiosyncratic; TrueType fonts can be used in both X windows and Postscript files but are slow and bulky; Postscript fonts can only be used in PostScript files and can't be 'skewed', but have high resolution and don't add to the size of the file.

Technical aside: The majority of examples here and in the documentation tacitly assume the use of a PseudoColor (8-bit) display. Although IDL can be used on TrueColor (24-bit) displays in such a way as to exploit the greater colour depth, you may initially find it simpler to force 8-bit operation on TrueColor displays by adding `device, decomposed=0` when defining the X window. Or if TrueColor is important to you, then see the Physics IT support FAQ.

When outputting grayscale or colour to a PostScript file, it is necessary to enable colour maps in the printer, and to state the resolution with which colours are to be specified to the printer. Include something like

```
device, /color, bits_per_pixel=8.
```

5.2 Plotting

Having managed to read in the data and overcome the complications of output device configuration, it just remains to plot the data, and that can be as simple as

```
plot, y
```

which plots all values in the vector `y`, with the x-axis labelled 0 1 2... More often however, you will want to supply a vector of x-values so that the abscissa is more usefully labelled:

```
plot, x, y
```

This command by default plots the data points joined by a solid line, and adds axes on all four sides with tick marks pointing inwards. All this and more can be altered with keywords. The following keywords affect the overall appearance of the plot, or the way the lines are drawn,

BACKGROUND	Background colour index when erasing.
CHARSIZE	Overall character size.
CHARTHICK	Overall thickness for vector fonts.
CLIP	Coordinates of clipping window.
COLOR	Colour index for data, text, line, or polygon fill.
DATA	Set to plot in data coordinates.
DEVICE	Set to plot in device coordinates.
FONT	Text font index: -1 for vector, 0 for hardware, 1 for TrueType fonts.
LINESTYLE	Linestyle used to connect data points.
NOCLIP	Set to disable clipping of plot.
NODATA	Set to plot only axes, titles, and annotation w/o data.
NOERASE	Set to inhibit erasing before new plot.
NORMAL	Set to plot in normal coordinates.
POSITION	Position of plot window.
PSYM	Use plotting symbols to plot data points.
SUBTITLE	String for subtitle.
SYMSIZE	Size of PSYM plotting symbols.
T3D	Set to use 3D transformation stored in !P.T.
THICK	Overall line thickness.
TICKLEN	Length of tickmarks in normal coordinates. 1.0 produces a grid. Negative values extend outside window.
TITLE	String for plot title.
ZVALUE	The Z coordinate for a 2D plot in 3D space.

and the following affect the way the axes are drawn,

[XYZ]CHARSIZE	Character size for axes.
[XYZ]GRIDSTYLE	Index of linestyle to be used for tickmarks and grids.
[XYZ]MARGIN	Margin of plot window in character units.
[XYZ]MINOR	Number of minor tick marks.
[XYZ]RANGE	Axis range.
[XYZ]STYLE	Axis type.
[XYZ]THICK	Thickness of axis and tickmark lines.
[XYZ]TICKFORMAT	Allows advanced formatting of tick labels.
[XYZ]TICKLEN	Tickmark lengths for individual axes.
[XYZ]TICKNAME	String array of up to 30 labels for tickmark annotation
[XYZ]TICKS	Number of major tick intervals for axes.
[XYZ]TICKV	Array of up to 30 elements for tick mark values.
[XYZ]TICK_GET	Variable in which to return values of tick marks.
[XYZ]TITLE	String for specified axis title.

where '[XYZ]' should be replaced by 'X', 'Y', or 'Z'. The on-line help explains the use of these options more fully. These options apply not just to `plot`, but also to the many other IDL plot commands, including: `bar_plot`, `contour`, `errplot` for plotting error bars, `plot_field` for plotting vector fields, `plots` for plotting symbols, `vel` for plotting streamlines, and many more. See 'Plotting routines: Two dimensional and general' under 'List of routines by application' in the on-line help.

5.3 Contour plots

Data defined on a rectangular grid are easily plotted using `contour`. For example,

```
contour, dist(13,16)
```

generates a contour plot of the 13×16 array returned by the function `dist`. The more general form for calling this procedure is `contour, z, x, y`, which allows the grid to be specified (which can be unevenly spaced) and even allows points to randomly distributed in the plane. [But use `trigrd` when the points are distributed on a non-flat surface.] Much customization of contour plots is possible. In addition to the general keywords above, the keywords `levels`, `nlevels`, `fill` and `c_labels`, which are specific to `contour`, are commonly required. See under `contour` in the on-line help for several examples of this routine.

5.4 Multiple plots

A useful command to use in conjunction with `plot` is the command `oplot`. It is similar to `plot` except that it *overplots* an existing plot, retaining the same scale. It was demonstrated in Section 3.

Alternatively, if you wish to create several separate plots on the one page, the most convenient way is to set the value of the system variable `!p.multi`. For example, to create an array of three plots horizontally and two vertically, do

```
!p.multi=[0,3,2]
```

prior to plotting. Then successive calls to `plot` will fill the six plot regions from left to right, and top to bottom. [Alter the first value if you want to start plotting at a different point in the sequence.]

6 Three dimensional plots

Surface plots are an alternative to contour plots: compare the plots produced by `contour`, `dist(13,16)` and `surface`, `dist(13,16)`. However there are usually more complications with surface plots, the issues being orientation, perspective, colour and style of shading.

Orientation can be set with the keywords `ax` and `az` which specify a rotation around the X axis that is followed by one around the Z axis. [Both are in degrees and have default values of 30 degrees.] For more complete control of angles, perspective, scales and translations try the procedure `t3d`, which manipulates the general transformation matrix `!p.t`.

The choice of available colours is most conveniently made by typing `loadct, n`, where $n=0..40$. The result of this is a colour table (or gray table when $n = 0$), which can be viewed with `xpalette`, and can be used in plots of all kinds via the colour indices $0..255$. The default foreground and background colours are at the extremes of this range. You can also select a colour map using the `xloadct` widget, or specify your own colour map with `tv1ct` or `xpalette`.

IDL has four styles of shading. In the following examples `surf` is a 2-D array specifying the surface and `shade` is some other array **with the same dimensions**.

1. The surface is plotted as a wire-mesh with some specific colour.
`surface, surf, color=60`
2. The surface is plotted as a wire-mesh with the colour at each node specified by an independent 2D array.
`surface, surf, shades=bytsc1(shade, top=!d.table_size-1)`
3. A continuous surface is rendered, as if illuminated by a point source.
`shade_surf, surf`
4. A continuous surface is rendered, with the colours provided by an independent array.
`shade_surf, surf, shades=bytsc1(shade, top=!d.table_size-1)`

The function `bytsc1` simple rescales the values in `shade[...]` to be in the range $0..!d.table_size-1$, so that they correspond exactly to the range of available indices. The examples above show the general case where `surf` and `shade` are independent 2D array, but actually it is common for them to be identical: in this case the surface is colour coded according to height.

If you want to explore fancy effects, try the following:

```
; Combined plots in 3D
loadct, 3
shade_surf, dist(10), /save
contour, dist(10), /t3d, zvalue=1, /noerase, /noclip
```

and

```
; Simple animation
for j=0,19 do begin
  wait, 0.2
  shade_surf, dist(10), az=18*j, xstyle=4, ystyle=4, zstyle=4
end
```

Selecting a suitable colour map is best done with the widget tool `xloadct`. The widget `xpalette` is similar, and good for fine-tuning colour maps.

7 Images

Images are two dimensional arrays of pixels, the only complication is getting the colours right.

In the simplest case you have a JPEG, TIFF etc. file, in which the palette is provided. Then you can use functions with names like `read_jpeg` to read both the image and palette into arrays, load the colour table, and display the image. For example,

```
pic=read_tiff('my.tiff',r,g,b)
tv!ct, r,g,b
tv, pic
```

There are similar functions with names like `write_tiff` for writing images to files. Such cases are too simple to consider further.

More challenging is the case where you have an 2-D array that you want to represent as an image. We assume each pixel is represented by a single number. By scaling, offsets and clipping these raw values will have to be reduced to an array of **bytes**, each of which is in the range $0..!d.table_size-1$. Only after an image has been transformed into a suitably-scaled byte array can it be plotted using the command `TV`.

Actually there is a variant command `TVSCL` that scales integer or float data to the required range, although it is generally better to manipulate the raw image explicitly, as in the following example. If you want to image pixel values in the range 20..750, and to use the full range of available colours, then

```
NormalData=bytsc1(RawData,min=20, max=750, top=!d.table_size-1)
TV, NormalData
```

Those with TrueColor (24-bit) displays do not have to bother with scaling of course: if the pixels are read in as RGB triplets then `TV` can display the images directly. You just need to set `TV`'s keyword `true` to 1, 2 or 3 to tell it whether the values in the array are ordered $3 \times m \times n$, $m \times 3 \times n$ or $m \times n \times 3$.

Those with PseudoColor (8-bit) displays will have to select or create a colour map prior to calling `TV`. This is done with `loadct`, `xloadct`, `tv!ct` or `xpalette`. [This topic was touched upon in Section 6.]

When displaying images, the point (0,0) of an image will appear at the lower-left corner. If the window is big enough to display several images, then successive calls to `TV` will result in images being placed from left to right and top to bottom. The alternative to automatic placement is explicit placement. In the example below, a colour bar is placed to the left of a 256 x 256 image.

```
window, xsize=356, ysize=256
for i=0,19 do begin
    colour=(!d.table_size/20)*i
    tv, replicate(colour,30,10), 30, 10*i+30
end
tv, NormalData, 100,0
```

Image processing is one of IDL's strengths. See the list of functions in 'Array and Image Processing Routines' under 'Routines by Application' in the on-line help. There are function for resizing, filters, transformations and much more.

8 Running a script whenever IDL is started

You can set the environment variable `IDL_STARTUP` to equal a filename, and that file will be run whenever IDL is started.

This is useful, for example, if you want convenient access (whatever your current directory) to a set of IDL scripts located in a particular directory. One way to do this reliably is

```
if strpos(!path,'~/idl:') eq -1 then !path='~/idl:' + !path
```

(assuming the scripts are in `~/idl`), which prepends `~/idl` to IDL's path. You don't want to type this every time you start IDL, so this line can be copied into a file pointed to by the environment variable `IDL_STARTUP`. If the file is `~/idl/idl_startup.pro` then set the environment variable by adding the line

```
export IDL_STARTUP=$HOME/idl/idl_startup.pro
```

to `~/ .bashrc` if you use the `bash` shell, or the line

```
setenv IDL_STARTUP $home/idl/idl_startup.pro
```

to `~/ .cshrc` if you use `csh` or `tcsh`. Of course, any number of additional commands can be placed in the startup script.

9 Finally

This introduction has focussed on graphics and imaging, but IDL is a broad general programming language. Topics not covered include using IDL for curve fitting and optimization, user-defined functions and procedures, object oriented programming, and calling C routines from IDL. There also tools ('widgets') for developing your own graphical user interfaces, and numerous widget-based programs supplied: a particularly useful one is the graphical file selection routine,

```
InputFilename=dialog_pickfile(path='.', filter='*.dat')
```

and the previously-mentioned utilities `xloadct` and `xpalette`.

IDL programming will be even more enjoyable if syntax highlighting is used. This is built-in to IDLDE, but if you prefer to use your favourite editor, it requires you to install an add-on. Emacs users should go to

```
http://idlwave.org,
```

while nedit users can find an IDL pattern file at www.nedit.org.

If you are interested in getting the most out of IDL, there is `idldemo` and there are examples in subdirectories `lib` and `examples` of `/usr/physics/rsi/idl/`. The book *IDL Programming Techniques* is available from the Theory library, as are the IDL manuals.