

An Introduction to IDL

Hugh C. Pumphrey

September 13, 2005

Contents

1	Lecture 1: Getting Started	1
1.1	So, what is IDL? What can it do for me?	1
1.2	Starting IDL.	2
1.3	A First Plot	2
1.4	Importing data	3
1.5	Printing your graph.	4
2	Lecture 2: Programming in IDL	5
2.1	Variables, data types	5
2.1.1	arrays	5
2.1.2	Structures	6
2.2	Flow Control	6
2.2.1	The IF statement	7
2.2.2	The FOR loop	7
2.2.3	The WHILE loop	8
2.2.4	Other flow control statements	8
2.3	Array operations	8
2.4	Defining functions and procedures	9
3	Lecture 3: More graphics	10
3.1	Surface plots	11
3.2	Contour Plots	11
3.3	Colours in IDL	12
3.4	Maps in IDL	12
4	Lecture 4: Images in IDL	13
4.1	Greyscale images	13
4.2	Colour images	14

Abstract

These lectures are intended to provide you with an introduction to the IDL (Interactive Data Language) scientific graphics package. They cover, in four short sessions, all that you need to know to get yourself started using IDL and to begin producing useful results. They certainly will not teach you everything that IDL can do – that would take a whole term or more. It is important to remember that learning any computing language or package is largely a matter of practice. You will not learn much by just listening to me. I can show you the sort of things that can be done and I can show you how to get started. Becoming proficient is up to you – I hope these lectures stimulate you to go away and do just that. These lectures do not assume any knowledge of IDL at all. They do assume that you are comfortable using the Sun workstations and that you can program in some procedural programming language (*e.g.* C, Fortran, pascal).

1 Lecture 1: Getting Started

1.1 So, what is IDL? What can it do for me?

IDL is a command-line driven scientific graphics package. “Command-line driven” means that you use it by typing instructions at a prompt. “Scientific graphics package” means that its purpose in life is to turn your dull files full of data into exciting-looking, informative, illuminating pictures.

1.2 Starting IDL.

To start IDL, type “idl” at the prompt in a terminal window. You should get a response like this:

```
humilis.hcp>idl
IDL Version 6.1, Solaris (sunos sparc m64). (c) 2004, Research Systems, Inc.
Installation number: 3451.
Licensed for use by: Edinburgh Meteorology Department
```

```
IDL>
```

(There may be a warning message which you can ignore.) If you get that, you are ready to go. If not, then there is one thing you can check on. The file `.bashrc` in your home directory may need to contain the line:

```
source /usr/local/rsi/idl/bin/idl_setup.bash
```

If it does not, edit `.bashrc` to add this line. Then either open a new terminal window or type the above line at the prompt in an terminal window that you already have open. IDL should then start if you type “idl” – if it doesn’t, go and ask for help.

Lets assume now that IDL is working properly. Type “`idlinfo`” — this tells you several things:

1. IDL has a graphical programming tool called `idlde`. This is not covered in this course. You may find it useful for more complicated projects.
2. IDL has some demonstrations built in which you can see by typing “`demo`” at the IDL prompt.
3. The online manual may be started by typing “`?`” at the IDL prompt (or `idlhelp` at the unix prompt).

The last item is by far the most important. The on-line manual has details of every aspect of IDL. If you don’t like on-line manuals, there are manuals on paper in room 7307. (Don’t take them out of room 7307, please.) There is also a book called *IDL Programming Techniques* by David Fanning. (It’s white and has a big black plastic spiral binding which may help you find it if it has been left at the bottom of a pile. Don’t take it out of 7307 either.)

1.3 A First Plot

Now that we have IDL working, let’s make a plot with it. Let’s suppose that we have several points (each of which has horizontal and a vertical co-ordinate) and that we want to plot them on a graph. We’ll be traditional and call the horizontal co-ordinates x and the vertical co-ordinates y . We can enter the data into IDL like this.

```
IDL> x = [1, 2, 4, 5, 6.7, 7, 8, 10 ]
IDL> y = [40, 30, 10, 20, 53, 20, 10, 5]
```

We now have two array variables called x and y , each containing eight numbers. To make a plot of x against y , we just type this:

```
IDL> plot,x,y
```

and there it is. It is a bit plain, but we have visualised our data. No-one else is going to know what it means unless we label the axes, like this:

```
plot,x,y,title='Should he resign?',xtitle='weeks',ytitle='Popularity'
```

We suppose that the data is the popularity ratings of a politician as determined by a polling organisation. Notice that there are two different ways in which we have provided instructions to the ‘plot’ routine. The data, x and y , are provided as positional parameters. The order they come in matters. Here, the first parameter is the horizontal co-ordinates of our data, the second is the vertical co-ordinates. Optional things like the labels are passed as keyword parameters. An important IDL routine like `plot` usually has many of these. You can supply them in any order you like.

Now let’s suppose that our politician’s popularity is being measured by two polling organisations. We put both sets of measurements on the same plot by using `plot` for the first set and `oplot` for the second.

```
IDL> y2 = [30, 28, 8, 19, 50, 22, 12, 6]
IDL> oplot,x,y2,linestyle=2
```

Note how we use the `linestyle` keyword to make the second line different from the first.

This is getting to the stage where it might be tedious to re-type everything to correct a mistake we made earlier. We can avoid this by putting a list of IDL commands into a file to make a program. As an example, use your favourite text editor to create a file called `dubya.pro` and put the following lines in it:

```
; **** Here are the data. ****
x = [1, 2, 4, 5, 6.7, 7, 8, 10 ] ; horizontal co-ordinates
y = [40, 30, 10, 20, 53, 20, 10, 5] ; vertical co-ordinates (1st set)
y2 = [30, 28, 8, 19, 50, 22, 12, 6] ; vertical co-ordinates (2nd set)

; **** make a plot of the first set of data ****
plot,x,y,title='Should he resign?',xtitle='weeks', $
    ytitle='Popularity', psym=-2

; --- Note how we use $ in the above command to continue onto the
; --- next line

; ++++++ Add the second set to the plot ++++++
oplot,x,y2,linestyle=2, psym=-6

; End of the program - don't forget this!
end
```

Note that the semicolon is used to indicate a comment – IDL ignores everything on a line after the first semicolon. As an aside, if you use `xemacs`, you will be pleased to know that it has an IDL mode. To get it to recognise your file as IDL, edit `.xemacs/init.el` and add these magic lines:

```
; Tells EMACS to assume that files ending in .pro are IDL
(setq auto-mode-alist (cons '("\\.pro\\'" . idlwave-mode) auto-mode-alist))
```

To run the program, type

```
IDL> .run dubya
```

It is important that the file name ends in “.pro” or IDL may not find it. The file also has to be in the directory from where you started IDL. If you have never used IDL before, why not take this program, run it, and then make some changes to it. Read up on the ‘plot’ routine in the manual and find out how to do things like change the axis ranges, use logarithmic axes, plot the lines in colour etc. etc.

1.4 Importing data

You now know how to make a line plot of some data. In most cases you will have far too much data to want to type it in. Typically the data will be in a file and you will want IDL to read it in and then make a plot of it. As an example, we will use some numbers generated by the MODTRAN radiative transfer package. The instructions for one of the MSc practicals show you how to run MODTRAN and how to remove the header lines from the file. This leaves you with a file containing 14 columns of numbers, like this:

```
14000. 0.714 7.97E-31 1.56E-26 2.10E-07 4.12E-03 4.11E-08 1.18E-07 2.32E-03 2.94E-08 3.29E-07 6.44E-03 1.64E-05 0.3950
14100. 0.709 4.95E-31 9.84E-27 2.13E-07 4.24E-03 4.15E-08 1.18E-07 2.35E-03 2.90E-08 3.31E-07 6.59E-03 4.96E-05 0.3940
14200. 0.704 3.05E-31 6.15E-27 2.11E-07 4.25E-03 4.17E-08 1.15E-07 2.32E-03 2.81E-08 3.26E-07 6.58E-03 8.22E-05 0.3887
14300. 0.699 1.84E-31 3.77E-27 1.90E-07 3.90E-03 4.06E-08 1.03E-07 2.11E-03 2.56E-08 2.93E-07 6.00E-03 1.12E-04 0.3734
14400. 0.694 1.13E-31 2.35E-27 1.87E-07 3.88E-03 4.05E-08 9.90E-08 2.05E-03 2.46E-08 2.86E-07 5.93E-03 1.40E-04 0.3664
14500. 0.690 6.53E-32 1.37E-27 1.60E-07 3.36E-03 3.74E-08 8.57E-08 1.80E-03 2.17E-08 2.46E-07 5.17E-03 1.65E-04 0.3446
.
.
.
33900. 0.296 0.00E+00 0.00E+00 2.64E-10 3.01E-05 2.62E-10 2.91E-18 3.33E-13 1.99E-20 2.64E-10 3.01E-05 3.27E-03 0.0001
33900. 0.295 0.00E+00 0.00E+00 1.68E-10 1.93E-05 1.67E-10 1.52E-19 1.75E-14 1.05E-21 1.68E-10 1.93E-05 3.27E-03 0.0000
34000. 0.294 0.00E+00 0.00E+00 1.82E-10 2.10E-05 1.80E-10 1.83E-20 2.11E-15 1.26E-22 1.82E-10 2.10E-05 3.27E-03 0.0000
```

The columns are 201 rows long. Let us suppose that we want to plot the 12th column (which is total radiance) against the 2nd column (which is wavelength). Here is a short IDL program which will do this.

```

;;;;; Set up array to contain the data ;;;;;;;;;;;;;;
modtrandata=fltarr(14,201)

;;;;; Open the file, attach it to unit no. 1 ;;;;;;;;;;
openr,1,'/home/hcp/wrk/idlcourse/modtran/pldat.dat'

;;;;; read the data from the file attached to unit 1 ;;;;;;;;;;
;;;;; into the array modtrandata
readf,1,modtrandata

;;;;; Close the file ;;;;;;
close,1

;;;;; Now make the plot ;;;;;;
plot,modtrandata(1,*),modtrandata(11,*), $
  xtitle='Wavelength / microns',$
  ytitle='Radiance in Watts/(cm!U2!N Steradian Micron)'

;;;;; Don't forget the 'end' statement ;;;;;;

end

```

Note that we have to know in advance how many rows and columns there are in the file. We'll look at how to get around this later in the course. Note also that for an IDL array with n elements, the individual elements are numbered from 0 to $n - 1$, just as in C or Java. They are not numbered from 1 to n as in Fortran.

1.5 Printing your graph.

It's great to have plots on the screen, but you will often need to get them on paper to include in reports etc. IDL treats the screen as one device and a postscript file (which you can print) as another. You can switch devices with the `set_plot` command:

```
IDL> set_plot,'ps'
```

will switch to postscript output and

```
IDL> set_plot,'x'
```

will switch back to the screen. The `device` command allows you to do various things to control which postscript file your output goes to, how big your plot will appear, and so on. Here is the program to plot modtran output, with extra lines added so that it will put the plot in a postscript file called modtran.ps

```

;;;;; Set up array to contain the data ;;;;;;;;;;;;;;
modtrandata=fltarr(14,201)

;;;;; Open the file, attach it to unit no. 1 ;;;;;;;;;;
openr,1,'/home/hcp/wrk/idlcourse/modtran/pldat.dat'

;;;;; read the data from the file attached to unit 1 ;;;;;;;;;;
;;;;; into the array modtrandata
readf,1,modtrandata

;;;;; Close the file ;;;;;;
close,1

;;;;; switch to postscript output ;;;;;;;;;;
set_plot,'ps'

```

```

;;;;; set the name of the output file (if you dont set it, it is
;;;;; called idl.ps)
device,file='modtran.ps'

;;;;; Now make the plot ;;;;;
plot,modtrandata(1,*),modtrandata(11,*), $
  xtitle='Wavelength / microns',$
  ytitle='Radiance in Watts/(cm!U2!N Steradian Micron)'

;;;;; Close the postscript file (important. The last bit of your plot
;;;;; wont appear unless you do this
device,/close

;;;;; switch back to X Window system ;;;;;;
set_plot,'x'

;;;;; Don't forget the 'end' statement ;;;;;;

end

```

And that's it for the first session. You can now make most of the 2-D graphs that you will need. In the next session we'll look at some more sophisticated programming techniques.

2 Lecture 2: Programming in IDL

This lecture covers the nuts and bolts of IDL programming. There won't be much graphics but we will look at many things which will help you to make good use of IDL. This is the bit of the course where I assume that you can program in C or Fortran. I will, for example, explain what sorts of variables there are in IDL, but I won't explain what a variable is in any detail. If you don't like the way I present the material, the place to look it up is the section of the on-line manual called "Building IDL Applications".

2.1 Variables, data types

As with most languages, IDL has several types of variables. I list some of them here:

type	range	bytes	to define	to convert
byte	0 to 255	1	b=15B	b=byte(x)
integer	-32768 to +32767	2	i=15	i=fix(x)
long	-2147483648 to +2147483647	4	j=long(15) j=147483647	j=long(x)
floating pt	$\pm 10^{38}$, 7 sig figs	4	y=1.7	y=float(x)
double prec.	$\pm 10^{308}$, 14 sig figs	8	y=1.7d0	d=double(x)
complex	two floating point no.s	8	z=complex(1.2,0.3)	z=complex(x)
string	(used for text)	0-32767	s='blah'	s=string(x)

Variable names can have letters, numbers and underscores in them. They are NOT case-sensitive: foo, Foo and FOO are all the same variable.

Note that the default type of integer is only a 2-byte integer. If you want a long integer you have to remember to ask for it.

IDL is a dynamically typed language. That means that you don't have to define your variables or say which variable is which type at the start of your program. You can say `gak=37.5d0` at any point in your program and a double precision variable called `gak` will spring into existence and take on the value 37.5. If you had a variable of a different type called `gak` at some earlier point in your program, it will vanish when you create the double precision variable with the same name.

2.1.1 arrays

In addition to single, scalar variables, IDL has arrays. An array is a numbered group of variables, all of the same type. We used arrays in the first lecture to hold the data for line plots. You can define an array by putting the elements between square brackets, like this:

```
IDL> an_array=[3, 5, 6, 2.5, 100, 27.7]
```

Note that we have mixed up floats and integers in this expression; IDL will coerce all the integers to floats if there are any floats at all. If you define an array using integers only then IDL will leave them as integers.

You can refer to individual elements of the array like this:

```
IDL> print,an_array[0],an_array[2]
      3.00000      6.00000
```

Note that the first element of the array is numbered 0 as in 'C' (not 1 as in FORTRAN). Note also that you can use () or [] for indexing arrays. The square brackets are a recent addition to the language, but are the wiser choice because the parentheses () have other uses. You can refer to a subset of the array like this:

```
print,an_array[3:5]
      2.50000      100.000      27.7000
```

You can define a 2-dimensional array like this:

```
IDL> array_2d= [[ 2,3,4],[10,20,30]]
```

... and refer to parts of it like this:

```
IDL> print,array_2d[0,1]
      10
IDL> print,array_2d[1:2,1]
      20      30
IDL> print,array_2d[2,*]
      4
      30
```

Note how an asterisk, *, is used to represent an entire column or row of the array and how two integers and a colon are used to select part of a row or column.

Finally, you can set up an array to use later like this:

```
IDL> foo=fltarr(100,100)
```

will make foo a 100 by 100 element floating point array with all elements initialised to 0.0 .

2.1.2 Structures

For some purposes, arrays are not flexible enough. One might want a variable to contain elements of different types. If, for example, you wanted to keep information about a person in one variable, you might want to record his or her name (a string), age (an integer), height (a float). You can set up such variables, called structures, like this:

```
IDL> elvis={person,name:'Elvis Aaron Presley',age:64,height:1.7}
IDL> bill={person,name:'William Jefferson Clinton',age:58,height:1.85}
```

You can then access the different elements of the structures like this:

```
IDL> print,elvis.name
Elvis Aaron Presley
IDL> print,bill.age
      58
```

Giving your newly-defined type of variable a name ('person' in this case) is optional.

2.2 Flow Control

IDL has most of the constructs that you would expect for arranging loops, conditions and the like.

2.2.1 The IF statement

This is used when you want to do something if a condition is true and something else otherwise. The statement looks like this:

```
if condition then statement 1 else statement 2
```

This will execute statement 1 if the condition is true and statement 2 if the condition is false. (Note that in these descriptions of what a statement does I use **typewriter font** to show what you actually type and *italics* to indicate where you have to put in your own stuff.) Here is an example if an 'if' statement:

```
if i eq 2 then print,'I is two' else print, 'i is not two'
```

Notice how the logical operators you need to set up the condition look like the FORTRAN ones (but without the dots), not like the C ones. Here is a table of relational and boolean operators which you can use in the condition of an if statement.

Purpose	IDL	C	FORTRAN
Relational Operators			
Equal to	eq	==	.EQ.
Not equal to	ne	!=	.NE.
Less than or equal to	le	<=	.LE.
Less than	lt	<	.LT.
Greater than or equal to	ge	>=	.GE.
Greater than	gt	>	.GT.
Boolean Operators			
And	and	&&	.AND.
Not	not	!	.NOT.
Or	or		.OR.
Exclusive OR	xor		

If you want statement 1 and / or statement 2 to consist of more than one statement, then the if construct looks like this:

```
if condition then begin
statement 1a
statement 1b
statement 1c
endif else begin
statement 2a
statement 2b
statement 2c
endelse
```

The statements between a **begin** and an **end** are called a block of statements. The **begin** and **end** are analogous to the curly brackets in 'C' or Java except that the end statement has to match the thing that comes before the begin *e.g.* an **if . . . then begin** has to be matched with an **endif** and a **else begin** with an **endelse**. Blocks of statements are used within programs, not at the IDL prompt.

2.2.2 The FOR loop

If you have a statement or statements that you want to repeat a number of times, you can use the FOR statement to do so. It looks like this:

```
for variable = start_value , stop_value [ , increment ] do statement
```

(I use [] to indicate that *increment* is optional.) The *variable* will start at *start_value* and *statement* will be executed over and over again, with *increment* being added to *variable* each time, until *variable* reaches *stop_value*. If you miss out *increment* it is assumed to be 1.

Try these examples at the IDL prompt.

```
IDL> for j=0,5 do print,j
IDL> for j=0,6,2 do print,j
```

Just as a warning, the loop variable takes its type from the start value. This is a (short) integer in these examples. This means that **for j=0,40000 do . . .** will NOT do what you expect, but **for j=long(0),40000 do . . .** will be OK.

2.2.3 The WHILE loop

If you need a loop for which you don't know in advance how many iterations there will be, you can use the 'while' statement. It works like this:

```
while condition do statement
```

Again, if you are using this in a program you can replace *statement* with a block of statements – the block must end with an **endwhile**. You can construct *condition* in the same way as for an if statement.

Here is an example. It is the modtran plotting program again, but this time we don't need to know the number of lines in advance. The program reads a line at a time, stopping when it reaches the end of the file. We put the data into an array which starts off with more lines than we are likely to need. (Don't make the array big enough for ten million lines, though – you'll fill up the computer's memory!) Once we reach the end of the file, we truncate the data array to the length of the file.

```
;;;;; Set up array with far too many rows to contain the data ;;;;;;;;;;;;;;
modtrandata=fltarr(14,5000)

;;;;; Open the file, attach it to unit no. 1 ;;;;;;;;;;
openr,1,'/home/hcp/wrk/idlcourse/modtran/pldat.dat'

inputline = fltarr(14)           ; set up a variable to hold one row of the file
linecount=0                     ; variable to count lines

;;; start while loop to read in the data
while not eof(1) do begin       ; eof becomes true when we reach the End Of
                                ; the File.
    readf,1,inputline
    modtrandata(*,linecount)=inputline
    linecount=linecount+1
endwhile
modtrandata=modtrandata(*,0:linecount-1) ; truncate array to actual length
                                         ; of file.

;;;;; Close the file ;;;;;;
close,1

;;;;; Now make the plot ;;;;;;
plot,modtrandata(1,*),modtrandata(11,*), $
    xtitle='Wavelength / microns',$
    ytitle='Radiance in Watts/(cm!U2!N Steradian Micron)''

;;;;; Don't forget the 'end' statement ;;;;;;

end
```

2.2.4 Other flow control statements

IDL also has a Repeat loop (like a while loop, but the test is at the end), a case statement (for multiple-choice situations) and a goto statement (as in "considered harmful"). We don't have time to cover these – check the manuals if you are interested.

2.3 Array operations

IDL allows you to do many operations on whole arrays. Suppose, for example, you wanted to make an array whose elements were the squares of the elements in another array. You could use a for loop, like this:

```
xsquared = fltarr(n)
for j=0,n-1 do xsquared(j) = x(j) * x(j)
```

This is how you would have to go about it in C or Fortran. In IDL, however, you can do this:

```
xsquared = x*x
```

This will clearly make your code shorter and clearer but it will also make it *much* faster. IDL is not a true compiled language like C. When it tells you it has

```
\% Compiled module: $MAIN$.
```

it has checked the syntax and converted it to an internal form which is more efficient than interpreting what you typed in directly. (Perl wizards will know what I mean.) It does NOT compile the program into machine code like the C compiler does. The upshot of this is that an IDL program with too many ‘for’ and ‘while’ loops, ‘if’ statements etc. can be very slow. If you can replace the loops and ifs with array operations, the program will be a LOT faster – as fast as a C program in the best cases.

2.4 Defining functions and procedures

Our examples so far have been too short to break up into sub-programs. However, any sizeable IDL project will be easier to deal with if each specific task is put into its own sub-program, which your main program can call. Sub-programs in IDL come in two flavours: functions and procedures. The main difference is that functions return a value and procedures don’t. (This will be familiar to old FORTRAN hackers like me – C people are used to having only functions.) Procedures are defined like this:

```
pro procedure_name, arg1, arg2..... , keyword1=keyword1....  
statement  
statement
```

```
.
```

```
.
```

```
.
```

```
end
```

and are called like this

```
procedure_name, argval1, arg2..... , keyword1=value1....
```

Functions are defined like this:

```
function function_name, arg1, arg2..... , keyword1=keyword1....  
statement  
statement
```

```
.
```

```
.
```

```
.
```

```
return, return_value
```

```
end
```

and are called like this:

```
foo = function_name(argval1,....., keyword1=value1.... )
```

You don’t have to have any positional arguments or any keywords. If your function or procedure is called ‘smeg’ you should put it on its own in a file called `smeg.pro` in the same directory from where you started IDL. As you would expect, all the variables you set up within a procedure or function are local to that sub-program. If IDL has a problem and stops within a sub-program, you will find that the variables you can look at or print are those belonging to the sub-program. You can leave the ‘scope’ of the sub-program and return to the main level by typing

```
IDL> retall
```

Here is an example of a function. This calculates $\sin(x)/x$ – note the use of `if` and `while`.

```
function sinc,x  
; This idl function calculates the function sinc(x) = sin(x)/x.  
; ( Beware: some authors define sinc(x) = sin(Pi x)/(Pi x) )  
; sinc(0)=1, but you cannot calculate this directly as sin(0)/0.  
; This routine uses the power series  
; sin(x) / x = 1-(x^2)/3! + (x^4)/5! - (x^6)/7! for small values of x  
; This function only works for scalars at the moment  
  
if abs(x) gt 0.1 then begin  
    s=sin(x)/x                ; Use obvious formula if |x| > 0.1  
endif else begin  
    j=0  
    s=x*0+1.0                ;make s double if x is double  
    term=s                    ;term same type as s
```

```

;; Calculate power series - stop after 20 terms or if term gets
;; very small
while j lt 20 and abs(term) gt 1.e-12 do begin
    j=j+2
    term=-1.0*term*x*x/(j*(j+1))
    s=s+term
endwhile

endelse

return,s

end

```

To use this, put the text into a file called sinc.pro. If you type

```
IDL> .run sinc
```

IDL will inform you that it has compiled the function successfully. If you then type, for example,

```
IDL> print,sinc(0.05)
```

then the value of $\sin(0.05)/0.05$ will be printed out.

To summarise, we have learned about IDL's programming features and that we should use array operations instead of loops where that is possible. In the next lecture we will learn how to display two-dimensional data sets and how to make use of colour in IDL.

3 Lecture 3: More graphics

In this lecture we will learn how to display data that are a function of two variables. An example would be the height of the ground above sea level as a function of distance East and distance North. There are no figures in these notes – if you want to see what the plots look like, start IDL up and run the examples. We will construct an example data set using this short program:

```

nx=21                ; Number of points in x direction
ny=31                ; Number of points in y direction

xmin=-2             ; x axis goes from xmin ...
xmax=2              ; ... to xmax

ymin=-2             ; y axis goes from ymin ...
ymax=2              ; ... to ymax

; make 1-d arrays containing x axis and y axis values
x=xmin + findgen(nx)*(xmax-xmin)/(nx-1)
y=ymin + findgen(ny)*(ymax-ymin)/(ny-1)

; z is a 2-d array which will contain the data values
z=fltarr(nx,ny)

; x2d and y2d are arrays the same size as z. They contain the x values
; at each point and the y values at each point. This is done so that
; we can calculate z without using nested for loops.
x2d=z
y2d=z
for j=0,nx-1 do y2d(j,*)=y
for j=0,ny-1 do x2d(*,j)=x

; calculate z.
z=exp(-(x2d^2 + y2d^2)) + 0.6*exp(-((x2d+1.8)^2 + (y2d)^2))

```

```
; don't forget the 'end' statement
end
```

If you type this program into a file called `example2d.pro` (or, if you are looking at the HTML version of these notes with a web browser, cut and paste it) and then type:

```
IDL> .run example2d
```

you will be left with two 1-D arrays called `x` and `y` and a 2-d array called `z`. You can imagine that `x` is distance East, `y` is distance North and `z` is height above sea level. We will investigate several ways of displaying these data.

3.1 Surface plots

Perhaps the most immediate way of displaying a data set like our example is as a surface plot. Type

```
IDL> surface,z,x,y
```

and look at the result. Depending on your terminal, the axis labels may be a bit small. You can use the `'charsize'` keyword to make them more legible. Just as with the line plots we saw in the first class, you can label your axes: here we can label the `z` axis too. Try:

```
IDL> surface,z,x,y,charsize=2.5,xtitle='East',ytitle='North',ztitle='Alt.'
```

Like most IDL procedures, `surface` has lots of keywords to enable you to tweak the plot to look how you want it to look. Note particularly the `ax` and `az` keywords, which let you look at the plot from different angles. Try these examples:

```
IDL> surface,z,x,y,ax=5
IDL> surface,z,x,y,ax=85
```

You can also do:

```
IDL> surface,z,x,y ,bottom=20
```

to colour the lower surface in a different colour.

An alternative to the wire frame is a shaded surface. This is produced by the `shade_surf` command:

```
IDL> shade_surf,z,x,y
```

3.2 Contour Plots

It is less visually striking, but sometimes more useful, to display 2-d data as a contour plot. For a basic contour plot, do:

```
IDL> contour,z,x,y
```

It's a start, but there are not very many contours and we don't know what each contour represents. You can use the `levels` keyword to specify which contours you want:

```
IDL> contour,z,x,y,levels=findgen(21)/10,/follow
```

Note that `findgen(n)` returns an `n`-element array with the elements set to `[0, 1, 2, ...n-1]` so the above example will produce a plot with 21 contours going from 0 to 2 with a spacing of 0.1 between the contours. Mysteriously, the `follow` keyword makes IDL label the contours.

Plain contours are not very striking – it looks more eye-catching if we fill the spaces between the contours with colours, like this:

```
IDL> contour,z,x,y,levels=findgen(21)/20, /cell_fill
```

That makes the general form of the data clearer as you can see immediately where the highs and lows are, but we have lost the detailed information that comes from the labels. We can put it back like this:

```
IDL> contour,z,x,y,levels=findgen(21)/20,/follow,/overplot
```

Note how the `overplot` keyword allows you to put the contour lines on top of the fill.

3.3 Colours in IDL

So far, everything that we have seen has come out in grey. Now we'll look at how to get things to be in colour. Before looking at colours in IDL, we'll cover briefly how colours are done on computer monitors in general. Each pixel on the screen is made up of three colours: red, green and blue, each of which is represented by one byte and can therefore take on values between 0 (black) and 255 (as bright as possible). There are therefore $256^3 = 16777216$ possible colours. Some monitors (hopefully all of the ones you will use) have three bytes (24 bits) of video memory for each pixel - on such a monitor you can make any pixel any one of these 16777216 colours any time you like. A few older monitors may have only one byte of video memory per pixel, which means that there are only 256 colours available. You can make a pixel any of the 16777216 possible colours BUT you can only have 256 of those colours on the screen at any one time. Hopefully you won't encounter one of these. However as a result of this history, a lot of colour management in IDL is done via tables of (up to) 256 colours.

IDL is not brilliant at doing the right thing on systems with different colour abilities. In your case, you will almost certainly need to do

```
IDL> device,true_color=24,decomposed=0,retain=2
```

before it will work properly.

IDL has a selection of about 40 built-in colour schemes. You can load the fifth one of these, for example, by doing:

```
IDL> loadct,5
```

To return to the default grey scale, do:

```
IDL> loadct,0
```

Most of these built-in schemes have colour 0 as black and colour `!D.Table_size` as white. The default behaviour for most plotting commands is to draw in colour `!D.Table_size-1` but there is nearly always a colour keyword so that you can change the colour to one that you prefer. For example:

```
IDL> x=findgen(101)/100
IDL> plot,x,x*x
IDL> oplot,x,x^3, color=!D.Table_size*0.7
IDL> oplot,x,sin(!Pi*x), color=!D.Table_size*0.3
```

If you really want to control the exact values of each colour, you can do this with the `tv!ct` command. This:

```
IDL> red= [0,1,0,0,0,1,1,1]*255
IDL> green=[0,0,1,0,1,0,1,1]*255
IDL> blue= [0,0,0,1,1,1,0,1]*255
IDL> tv!ct,red,green,blue
```

will set colours 0 to 7 to be black, red, green, blue, cyan, magenta, yellow and white, in that order. (The other colours will be unaffected).

3.4 Maps in IDL

IDL has reasonably good facilities for drawing maps and for displaying data on top of them. To begin drawing data on a map, you must first use the `map_set` command:

```
IDL> map_set, /continents, /mercator, /isotropic
```

The mercator projection is just one of a selection available – check the section on the `map_set` command in the manual for more details. `Map_set` establishes a geographical co-ordinate system - most of the plotting commands work on top of the map. They assume that your x co-ordinate is longitude and your y co-ordinate is latitude.

For example: after the `map_set` command above, try this:

```
IDL> map_set, /continents, /mercator, /isotropic
IDL> longitude=findgen(360)-180
IDL> latitude=60*sin(longitude*!Pi/180.)
IDL> oplot,longitude,latitude
```

Contour plots, including filled ones, will work on top of a map, too.

To sum up, we have seen a number of ways of plotting a 2-dimensional data set and shown how colours are used in IDL and taken a quick look at IDL's mapping abilities. In the next section, we will look at how IDL can be used to display and process images.

4 Lecture 4: Images in IDL

To most computer systems a grayscale image is just a 2-D data set such as we looked at in the previous lecture, and IDL is no exception. The differences between images and other data sets are simply that:

- Images are (usually) bigger, perhaps several hundred data points in each direction.
- Images are (usually) of type byte. This means that each pixel can have only 256 values.

...and that's all. You can display and process greyscale images just like any other 2-dimensional array. You will find that contour or surface plots of an image of any size are not very useful and take a very long time to draw. In this lecture we look at ways of reading, displaying and processing images in IDL.

4.1 Greyscale images

IDL has routines to read in most of the standard image formats:

File type	Extension	IDL routine
Microsoft Bitmap	.bmp	read_bmp
Portable Network Graphics	.png	read_png
JPEG	.jpg or .jpeg	read_jpeg
Macintosh PICT	.???	read_pict
Portable Bit/grey/pixelmap	.pbm .pgm .ppm	read_ppm
Sun Raster format	.???	read_srf
Tagged Image File Format	.tiff or .tif	read_tiff
X11 Bitmap files	.xbm	read_x11_bitmap
X Windows Dump (from xwd)	.???	read_xwd

¹ For greyscale images, most of these require a file name and return a 2-d array of type Byte, but the syntax is not the same for all of these routines – see the manual for details. As an example, try this:

```
IDL> meteosat=read_png('~hcp/wrk/idlcourse/met.png')
```

This has read in a greyscale image from the file met.png and put it in a variable called meteosat. If we do

```
IDL> help,meteosat
```

we find that meteosat is a 640 X 400 array of type byte. The basic command in IDL to display an image is tv. If we do this:

```
tv,meteosat
```

then we can see the image, but there are a couple of problems. The image is not the same size as the window and it looks a bit washed out. We can fix the first problem by opening a window of the right size – you can open a 300 X 200 pixel window like this:

```
IDL> window,0,xsize=300,ysize=200
```

This provides us a nice example for writing our own procedure to open a window of the correct size for an image and then to display the image in it.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
pro showimage,image>window=window
; A short procedure to display an image in a window the same size as
; the image. The program refuses to display an image which is
; uselessly small. This is important as opening a window 0 pixels wide
; causes IDL to hang up.

if n_elements(window) ne 1 then window=0

siz=size(image)
if(siz(1) gt 4 and siz(2) gt 4 ) then begin
```

¹The exception is the gif format which is encumbered by some tiresome patent issues. If you need to read in a gif, convert it to png or ppm with the programs gif2png and giftoppm

```

        window,window,xsize=siz(1),ysize=siz(2)
        tv,image

endif else print,'Image too small'

end
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

```

Now we can display our image with:

```
IDL> showimage,meteosat,win=2
```

The image still looks a bit wishy-washy, though. If it was on the television, you would turn up the contrast. We can see why if we plot a histogram of the image:

```
IDL> plot,histogram(meteosat)
```

Note how most of the data are in the 120-250 range. We can use the `hist_equal` function to spread the values more evenly.

```
IDL> showimage,(hist_equal(meteosat))
```

and this shows *its* histogram:

```
plot,histogram(hist_equal(meteosat))
```

As you can see, it is more evenly spaced.

IDL has a few built-in functions for image processing. There is a boxcar smoother:

```
IDL> showimage,bytsc1(smooth(meteosat,9))
```

This can be used to reduce noise in an image. The second argument is the size of the box used. If you set this to 9, then each pixel in the new image is the average of the pixels in a 9 X 9 square in the original image. Be aware that there are better ways to smooth an image. There are two edge-detecting functions, called Roberts and Sobel, presumably after the inventors of the algorithms used:

```
IDL> showimage,roberts(meteosat),win=0
IDL> showimage,sobel(meteosat),win=2
```

In our example the main edges present are the continental outlines put on the image by Eumetsat so that we can tell where Scotland is when it is cloudy.

That just about covers what you need to know to get started with greyscale images. The next part looks at displaying colour images.

4.2 Colour images

A colour image is really three images, a red one, a green one and a blue one. If you want to do image processing things to a colour image, smoothing for example, then you have to apply your algorithm to the three colours separately. As we have seen, images are stored on computers in various file formats. Just like computer screens, some image formats (tiff, jpeg, ppm, some .png) are 24 bit true colour and store a red, a green and a blue image. Others (.gif, some .png) store an 8-bit image and a colour table to say which value in the image should be what colour.

Here is how to read in and display a pseudocolour png image

```
IDL> pg=read_png('/home/hcp/wrk/idlcourse/colourpeng.png',r,g,b)
%%IDL> pg=rotate(pg,7) ; wont be needed after upgrade
IDL> tvlct,r,g,b
IDL> showimage,pg,win=2
```

Remember that on 24-bit displays, you will need to do:

```
IDL> device,true_color=24,decomposed=0,retain=2
```

for this to work.

The ppm file format stores 24 bit images. Here is how to read in a ppm file, we use the `help` command to see what the resulting array is like:

```
IDL> read_ppm, '/home/hcp/wrk/idlcourse/ferry.ppm', ferry
IDL> help, ferry
FERRY          BYTE          = Array[3, 593, 413]
```

This is an 593 X 413 pixel image containing three 'layers', red, green and blue.

We can do this to see the individual layers:

```
IDL> loadct, 0
IDL> showimage, reform(ferry[2,*,*]), win=2
IDL> showimage, reform(ferry[1,*,*]), win=1
IDL> showimage, reform(ferry[0,*,*]), win=0
```

Note that the boat in the centre foreground looks dark in the red and green images but light in the blue image. To display this as a colour image on a 24-bit terminal, you just need to use the /true keyword with tv:

```
IDL> tv, ferry, /true
```

... and that's it. I hope you enjoy using IDL and that this course has made learning it a little easier.