

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

IDL is a scientific programming/graphics package. "Scientific graphics package" means that its purpose in life is to turn your dull programs that generate reams of data about some system into programs that not only generate reams of data, but also generate exciting-looking, informative and illuminating visualizations.

Getting Started**Install IDL**

If you will be installing IDL on your dorm computer or a laptop, obtain an appropriate CD and instructions from Professor Boccio. IDL will run under Mac OSX 10.3 or greater, Windows XP or Redhat Linux.

Initial Setup

Decide on an IDL working folder. For example, suppose we choose the folder `/Users/boccio/Documents/idl` as our IDL working folder.

- (1) Create a text file containing these lines

```
cd, '/Users/boccio/Documents/idl'  
DEVICE, decomposed=0, retain=2
```

and save it as the file `setup.pro` in your working folder.

- (2) START IDL
In IDL select Preferences under the File menu.
- (3) Click STARTUP.
Click SELECT WORKING DIRECTORY and browse and select your working folder
Click SELECT STARTUP FILE and browse and select the file `setup.pro`
- (4) Click PATHS
Click INSERT and browse and select your working folder
Click the box in the window next to the listing of your working folder (check mark should appear)
- (5) Click APPLY
Click SAVE
Click OK
Restart IDL

Using the IDL Command Line

These notes are meant to be a **hands-on**. You should read them sitting in front of a computer and you should be typing commands and seeing what happens as you read. So here is what you need to know to get started.

In addition to being a complete programming language, IDL is an interactive compiler, enabling users to quickly perform complex tasks by entering a few commands on a command-line. A lot can be learned from typing IDL commands at the command line. In particular, you learn to figure things out, to try things, to experiment with your data. This is called "**learning by fooling around**". I think it is one of the best ways to learn and use IDL.

So let us make a quick pass through IDL using the **COMMAND LINE**.

Anatomy of an IDL Command

This command prints the integer 15, the result of 3 multiplied by 5.

```
IDL> print,3*5
      15
```

The word **print** in the command above is the name of the IDL **command** or **program**. It must be spelled out in its entirety - no shortcuts! It is **not case-sensitive**.

A command is always followed by a **comma**. The arguments or **variables** to be used by the command come next.

In the example above 3*5 (3 multiplied by 5) is the variable. Besides variables, we can add **keywords** which modify the operation of the command (more about keywords later).

Variables can be **dynamically created** in IDL. This command assigns an **integer** with the value of 3*5 to the variable `a`.

```
IDL> a=3*5
```

The **HELP** routine confirms that `a` is a single(scalar) integer variable with the value 15.

```
IDL> help,a
      A          INT          =          15
```

Help with IDL Commands

IDL comes with an extensive on-line help system, which is accessed by simply typing a question mark at the IDL command prompt or selecting from the Help menu. Type:

```
IDL> ?
```

You will be creating many variables in these classes. It will help if you know a little about them before you get started. Variable names must **start** with a letter. They can include other letters, digits, underscore characters, and dollar signs. A variable name may have up to 255 characters. They are not case-sensitive. Here are some valid variable names:

```
image2  or  this_image
```

As with most languages, IDL has several types of variables. They are listed in the table below:

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=long(x)
			j=147483647	
floating point	$\pm 10^{38}$, 7 significant figures	4	y=1.7	y=float(x)
double precision	$\pm 10^{308}$, 14 significant figures	8	y=1.7d0	d=double(x)
complex	two floating point numbers	8	z=complex(1.2,0.3)	z=complex(x)
string	used for text	0-32767	s='blah'	s=string(x)

We note the following definitions:

bit = 0 or 1

byte = 8 bits

```

00000001 = 1
00000011 = 3
00000111 = 7
00001111 = 15
00011111 = 31
00111111 = 63
01111111 = 127
11111111 = 255 = 28 - 1

```

Therefore a 1-byte integer can take on values from 0 to 255 (byte variable above).

A 2-byte integer can take on values from 0(00000000 00000000) to 255*255 (11111111 11111111) = 65535 = $2^{16} - 1$ (integer variable above). 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 (we will see how later).

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.

Now redefine the variable `a` to be the square root of its previous value and display information about `a` . The ampersand (&) character is used to separate multiple statements on the same line. Notice that the variable `a` has become a floating-point variable after execution of the `sqrt` command.

```
IDL> a=sqrt(a) & help,a
      A                FLOAT      =      3.87298
```

This means that variable type is dynamic in IDL. So be careful!

Line Continuation Characters

The **line continuation** character in IDL is the dollar sign, `$`. It indicates that the IDL command is continued on the next command line. This will be useful in IDL programming later, but is not very active when using the command line.

Creating Vectors

You can create a vector or an array (a vector is just a 1-dimensional array) variable at the IDL command line by enclosing the vector values in square brackets, like this:

```
IDL> vector = [1, 2, 3]
```

This is an integer vector because the data values are integer values. Use the Help command to learn about this new variable.

```
IDL> Help, vector
      VECTOR      INT      =      Array[3]
```

The variable vector is a three-element one-dimensional array.

If you want to add a fourth element to this vector, this is easily done in IDL. Just type this, for example:

```
IDL> vector = [vector, 4]
IDL> Print, vector
      1      2      3      4
```

Now make a 6-element vector containing the integers 1 through 6.

```
IDL> a=[1,2,3,4,5,6]
```

All IDL operators and functions work on both scalar and array data types with no change in notation.

```
IDL> print,a,2*a
      1      2      3      4      5      6
      2      4      6      8     10     12
```

Take the square root of each element of array a and put those values into variable b.

```
IDL> b=sqrt(a)
```

Entering the **HELP** command shows that both a and b are 1-dimensional arrays with dimension of 6. The elements of a are integers while the elements of b are floating-point values.

```
IDL> help,a,b
A                INT      = Array(6)
B                FLOAT    = Array(6)
```

Display the 6 floating-point elements in the array `b` .

```
IDL> print,b
1.00000      1.41421      1.73205      2.00000      2.23607
2.44949
```

These examples make it clear that by using arrays of numbers in a program we will be able to speed up operations by acting on all array elements at once.

Using Array Subscripts

If you want to add another element between the second and third elements in the array, then you use **array subscripting**. Array subscripts have their lower and upper bound separated by a colon. For example, you specify the first three elements of the vector above by:

```
IDL> print, vector(0:2)
      1      2      3
```

Notice that vector (or array) subscripts **start** at 0 and not at 1. Notice also that vector subscripts use parentheses to distinguish themselves. This will make it difficult sometimes to distinguish a call to a **function command** (discussed shortly) from a subscripted array. To help with this problem, IDL also allows you to write array subscripts with square brackets. In other words, if you are running IDL you can type this:

```
IDL> print, vector[0:2]
      1      2      3
```

To use array subscripting to put another element between the second and third elements of the vector, you can do this:

```
IDL> vector = [vector(0:1), 5, vector(2:3)]
IDL> Print, vector
      1      2      5      3      4
```

As earlier stated, IDL is a programming language that excels at working with vector or array data, so there are a number of built-in IDL commands for creating vectors and arrays of the different data types. In particular, there are functions for creating arrays of the proper data type in which each element is initialized to zero, and there are functions for creating arrays of the proper data type in which each element is initialized to its own index in the array.

For example, to create a 100x100 float array (2-dimensional array) of zeros, you can type this:

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

To create a vector of 100 floating point values ranging in value from 0 to 99, you can type this:

```
IDL> vector = findgen(100)
```

We will also use `indgen(num)`, which generates a vector of integers.

A wide variety of vectors can be created using these array creation routines. For example, to create a 6-element floating-point vector with values ranging from 0 to 50, you can type this:

```
IDL> vector = findgen(6) * 10
IDL> Print, vector
```

```
0.000000      10.0000      20.0000      30.0000      40.0000      50.0000
```

Now define a as an array(vector) of 100 floating-point elements.

```
IDL> a=fltarr(100)
```

Initially, the array elements are all set to zero.

The **FOR loop** below stores in each element of a the value of its subscript. For example, the value of a(0)=0 and the value of a(40)=40.

```
IDL> for i=0,99 do a(i)=i
```

Note that the command line structure of for loops has a special format which is different than the format used later in IDL programs.

Remember IDL subscripts begin at 0 and go to one less than the number of elements. Command line loops **cannot** be continued to a second line as will be possible in programs.

The FOR loop above works as follows:

```
enter loop
set i = 0 then a(0) = 0
set i = 1 then a(1) = 1
set i = 2 then a(2) = 2
.....
.....
.....
.....
set i = 99 then a(99) = 99
exit loop
```

This command prints the first and last of our 100-element array.

```
IDL> print,a(0),a(99)
      0.00000      99.0000
```

Subarrays can be specified by using subscript ranges. This command prints the values of a(10) **through** a(19), which, r is really the 11th through 20th elements.

```
IDL> print,a(10:19)
10.0000      11.0000      12.0000      13.0000      14.0000
15.0000      16.0000      17.0000      18.0000      19.0000
```

Creating Arrays

Small arrays can also be created **directly** from the IDL command line. For example, we can create a **3 column and 2 row array** like this:

```
IDL> array = [ [1, 2, 3], [4, 5, 6] ]
IDL> Print, array
      1      2      3
      4      5      6
```

Notice that this is an identical operation to first creating a vector and then **reformatting** that vector into a 3 column by 2 row array with the **Reform** command, like this:

```
IDL> vector = IndGen(6) + 1
IDL> array = Reform(vector, 3, 2)
IDL> print,vector
      1      2      3      4      5      6
IDL> print,array
      1      2      3
      4      5      6
```

Reform is a built-in IDL function which reforms (reshapes) IDL variables into new shapes (3,2) = (columns,rows).

What this also tells you is that vectors and arrays are stored in IDL in **row order**, i.e., fill first row, then second row and so on. This becomes important when you are writing IDL programs because you will often want to take advantage of the way data is stored in IDL.

Extracting Vectors and Subarrays

IDL also makes it easy to extract vectors and subarrays from within arrays. For example, consider this 10x20 array filled with randomly generated data in the interval (0,1):

```
IDL> data = Randomu(seed, 10, 20)
```

To pull out the columns 6-10 and rows 12-15, you can type this:

```
IDL> subarray = data(5:9, 11:14)
IDL> print,subarray
0.168011      0.848252      0.539318      0.236219      0.254671
0.869349      0.599377      0.185227      0.761207      0.580923
0.364475      0.0525108     0.538724      0.731956      0.847045
0.0842568     0.842200      0.821506      0.898021      0.488177
```

To create a vector of the 14th row, you can type:

```
IDL> vector = data(*,13)
```

The * symbol means ALL when used in subscripting.

To create an array of the last 5 columns of the data, type:

```
IDL> subarray = data(5:9, *)
IDL> Help, subarray
SUBARRAY      FLOAT      = Array[5, 20]
```

You see that the subarray is now a 5 column by 20 row array. You can,

of course, also use the symbol * to mean "all the rest" of the data. For example, to create the subarray with the last 5 columns of the data, you can also type this:

```
IDL> subarray = data(5:*, *)
IDL> Help, subarray
      SUBARRAY          FLOAT          = Array[5, 20]
```

Creating Graphics Windows

A graphics window is created with the Window command. For example, you can create and open a window by typing this:

```
IDL> Window,10
```

Notice that the title bar of this window has a 10 in it. This is this window's graphics **window index number**. Each graphics window has a unique graphics window index number associated with it when the window is created.

Positioning and Sizing Graphics Windows

Windows are positioned and sized according to an internal algorithm when they are created. You can position and size windows when you create them with **keywords** to the Window command. For example, to create a window that is 200 pixels wide and 300 pixels high, use the XSize and YSize keywords, like this:

```
IDL> Window, 1, XSize=200, YSize=300
```

Windows are positioned on the display with respect to the upper left-hand corner of the display in pixel or device coordinates. To position a window with its upper lefthand corner at location (75,150) on the display, use the XPos and YPos keywords, like this:

```
IDL> Window,2,XSize=200,YSize=300,XPos=75,YPos=150 ;Where is (0,0)?
```

Putting a Title on a Graphics Window

Sometimes you would like your graphics window to have a more descriptive title than just its graphics window index number. You can use the Title keyword to put a title on the window, like this:

```
IDL> Window, 3, Title='Example IDL Graphics Commands'
```

Erasing a Graphics Window

To erase the contents of a graphics window, you can use the Erase command, like this:

```
IDL> Erase
```

Flow Control

IDL has most of the constructs that you would expect for arranging loops, conditions, etc. We will use these structures shortly in programming.

The IF statement

This is used when you want to do one thing if a condition is true and something else otherwise.

The first case is to do something if condition is TRUE or continue on if condition is FALSE.

```
if condition then begin
    statement
    .....
endif
```

The second case is to do something #1 (if condition is TRUE) or something #2 if condition is FALSE.

```
if condition then begin
    statement
    .....
endif else begin
    statement
    .....
endelse
```

The "**condition**" is a statement that uses **relational and/or boolean operators**. Here is a table of **relational and boolean operators** which you can use in the "condition" of an if statement.

Purpose	IDL	C	FORTTRAN
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		

Some examples of "**conditions**" are:

(A LT 0.0) and ((A EQ B) AND (B LT 0.0))

Here is an example of an IF statement:

```
if (A GT 2.0) then begin
  A=A+1.0
endif else begin
  A=A-1.0
endelse
```

What happens if A = 3.0? If A = 2.0? If A = 1.0?

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 j=start_val,stop_val,increment do begin
  statement
  .....
endfor
```

The variable j will begin at start_val and the statements will be executed over and over again, with increment being added to variable j each time, until variable j reaches stop_val. If you leave out increment, then it has a default value of 1.

Here is an example of a FOR loop:

```
for j=1,100 do begin
  print,j,j^2
endfor
```

In this case, j begins with 1 and increments by 1 until j reaches 100. The result would be:

```
1 1
2 4
3 6
4 16
...
...
100 10000
```

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 begin
  statement
  .....
endwhile
```

We will see examples of the statements in programs later in the notes.

Programming in IDL

Any text editor can be used to prepare programs or functions of more than a few lines.

The GUI front-end for IDL includes a built-in text editor for your convenience. We recommend BEdit on MacOSX.

Files containing IDL programs, procedures, and functions are assumed to have the extension name **.pro**.

Once the program has been entered into a file from some text editor, we run IDL and compile one or more program files using the **.RNEW** command (or using the **COMPILE/RUN** entries in the menus).

We will work our way through some functions and programs in these notes discussing new commands as they appear.

You should enter the code into text files and save the files and run them in IDL as we proceed through the notes.

Function and Program Files

Here is a sample function:

```
function stepvect,min,max,step
;this function generates a vector of values
;from min to max in increments of step.
n=floor(((max-min)/step)+1.0)
;n = number of values between min and max
;(including min and max) to return
return,min+indgen(n)*step
end
```

The function structure is:

- (1) definition line: function function_name, arguments
- (2) body of function: IDL commands
- (3) return statement: returned value of function
- (4) end statement
- (5) saved in a file named **function_name.pro**

The function **stepvect** returns a vector of values from **min** to **max** with stepsize **step** .

Create a text file named stepvect.pro and use it as below.

It is used via the IDL command:

```
IDL> xx=stepvect(1.0,5.0,0.2)
IDL> print,xx
```

A function is a self-contained code unit that **returns a value** and can thus be **used** inside other expressions, i.e.,

```
IDL> yy=sqrt(stepvect(1.0,5.0,0.2))
IDL> print,yy
```

Programs

An IDL program is a self-contained code unit with a unique name that is called by other code units to **perform a set of commands**. The calling code unit and the procedure **communicate** via arguments passed between them.

The program structure is:

- (1) definition line: `pro program_name, arguments`
- (2) body of function: IDL commands
- (3) end statement
- (4) saved in a file named `program_name.pro`

A First Program and a First Plot

Now that we know a little bit about IDL, let's make a plot with it.

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 will 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 can just type this:

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

and there it is.

Note that a window with default size and location is used.

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

```
IDL> plot,x,y,title='Should He Resign?',xtitle='Weeks', $
      ytitle='Popularity'
```

(do not use the `$` symbol on the command line)

We suppose that the data is the popularity ratings of a politician as determined by a polling organization. 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**, that is, the order they come in matters. Here, the **first** parameter is the **horizontal** coordinates of our data, the **second** is the **vertical** coordinates. **Optional** things like the labels are passed as **keyword parameters**. You can supply them in any order you like. The possible positional parameters and possible keywords are defined in the HELP system (accessed with `?`)

Now let's suppose that our politician's popularity is being measured by two polling organizations. We put both sets of measurements on the same plot by using the **plot** command for the first set and the **oplot**

(**overplot**) command 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 favorite text editor to create a file called **clinton.pro**, put the following lines (just the lines we entered above) in it and save it:

```
pro clinton
;Here are the data
; horizontal coordinates
x = [1, 2, 4, 5, 6.7, 7, 8, 10]
; vertical coordinates (1st set)
y = [40, 30, 10, 20, 53, 20, 10, 5]
; vertical coordinates (2nd set)
y2 = [30, 28, 8, 19, 50, 22, 12, 6]
;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
;psym chooses a symbol type
;negative sign tell IDL to connect symbols with lines
;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. To compile and run the program, type

```
IDL> .rnew clinton
IDL> clinton
```

It is important that the file name ends in ".pro" or IDL may not find it. The file also has to be in the IDL **working** directory as specified in the setup.

That's it we have created our first program!

Saving 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. On the Macintosh under OSX there exists a "screen capture" program. With the graphics window at the front, enter the following command

This brings a cursor onto the screen (big + sign). Move the cursor to the upper left corner of the plot and click and drag to lower right corner and unclick. The graphics window is now saved in a file on the DESKTOP.

Program Examples

Enter and run all of these programs. Try various parameters.

- (1) Program to print "Hello World" n times. The number of times n is an argument of the program.

```
pro prog_01,n
; note use of argument of the function
for j = 0, n-1 do begin
    print, 'Hello World'
endfor
end
```

- (2) Program to generate a vector of length n whose elements are randomly distributed about 5.0 with a spread of ± 0.5 and then compute the average and standard deviation of the set of numbers. Note how the print statements contain strings to make the output easier to understand.

```
pro prog_02,n
x=5.0+0.5*randomu(seed,n)
sum1=0.0
sum2=0.0
for j = 0, n-1 do begin
    sum1=sum1+x(j)
    sum2=sum2+x(j)^2
endfor
xav=sum1/n
x2av=sum2/n
sd=sqrt((x2av-xav^2)/(n-1))
print, 'average = : ', xav
print, 'standard deviation = : ', sd
end
```

- (3) Version of program (2) that uses the powerful IDL function TOTAL to eliminate the FOR loop.

```
pro prog_03,n
n=long(n) ; must define n as a LONG integer or n < 32567 only
x=5.0+0.5*randomu(seed,n)
xav=total(x)/n
x2av=total(x*x)/n
sd=sqrt((x2av-xav^2)/(n-1))
print, 'average = : ', xav
print, 'standard deviation = : ', sd
end
```

Compare programs (2) and (3) for $n=10000$, 100000 , 1000000 and see the dramatic effect of eliminating the FOR loop.

- (4) Program to calculate an approximation for the factorial function $f(n) = n! = 1*2*3*....*n$ for large n .

```
pro prog_04, n
;Computes n!
;(only works up to n = 142)
n = double(n)
nfact = sqrt(2.0D*n*!pi)*n^n*exp(-n)*(1+1.0/(12.0*n)+1.0/(288.0*n^2))
print, 'The factorial of your number is:', nfact
end
```

- (5) Program to take a number entered at the keyboard (radius of a sphere), calculate the volume and print it out.

```
pro prog_05
rad=fltarr(1)
print, 'Enter number: '
READ, rad
vol=(4.0*!pi/3.0)*rad^3
print, 'The sphere volume is ', vol
end
```

- (6) Program to take three numbers entered as a, b, c which are the coefficients in a quadratic equation and compute the roots. Note use of complex numbers.

```
pro prog_06, a, b, c
det=b^2.0-4*a*c
if (det ge 0) then begin
    sol1=(sqrt(det)-b)/(2.0*a)
    sol2=(-b-sqrt(det))/(2.0*a)
endif else begin
    sol1=complex(-b/(2.0*a),sqrt(-det)/(2.0*a))
    sol2=complex(-b/(2.0*a),-(sqrt(-det)/(2.0*a)))
endelse
print, 'Solutions are : ', sol1,sol2
end
```

- (7) Program to do numerical integration of the function $f(x)=x^3$ by adding rectangles.

```
function intfun,x
z=x^3
return,z
end

pro prog_07, steps
a=0.0
b=2.0
n=long(steps)
h=(b-a)/n
x=a+h*findgen(n)
f=intfun(x)
print, 'Calculated integral = ', h*total(f)
print, "Theory = 4.00"
end
```

Exercise #1

Change prog_07 to do the integrals

- (1) x^4 (0,1)
- (2) $2*x+4*x^3$ (0,4)

Check your answers.

Exercise #2

Write a program that does this calculation:

- (1) $f(N) = \sum_{n=1}^N \frac{1}{n}$ for $N=40$ and $N=400$.
 - (a) Use a FOR loop.
 - (b) Do not use any loop.

- (2) $f(N) = \sum_{n=1}^N ne^{-n/10} \sin(n)$ for $N=40$ and $N=400$.

Use a function subprogram within your program.

Exercise #3

Make a plot of the function $f(x) = e^{-x} \sin(x)$ in the range $x=(0,3)$.

Your plot should have:

- (a) Plot a continuous solid line using 101 values in the range (include the endpoints).
- (b) Overplot 11 points including the endpoints using the "triangle" symbol. You will need to look up in HELP how to use symbols; also see clinton.pro above.
- (c) Your plot should specify X RANGE, Y RANGE, TITLE, XTITLE, YTITLE and anything else you find of interest. You will need to look up in HELP how to specify these KEYWORDS.

Introducing Colors

Let us now introduce colors into our program, that is, we switchy the background from black to white, switch the axis drawing from mwhite to black and use red and green for the data.

```
pro clinton_color
;Here are the data
; horizontal coordinates
x = [1, 2, 4, 5, 6.7, 7, 8, 10]
; vertical coordinates (1st set)
y = [40, 30, 10, 20, 53, 20, 10, 5]
; vertical coordinates (2nd set)
y2 = [30, 28, 8, 19, 50, 22, 12, 6]
; the next few lines allow us to control colors
; this is the RGB color model
; 255,255,255 = white
; 0,0,0 = black
; 255,0,0 = red
; 0,255,0 = green
```

```

; 0,0,255 = blue
; and so on
; set allowed colors
; black,white,red,green,blue,magenta,yellow,cyan
red= [0,1,1,0,0,1,1,0]
green=[0,1,0,1,0,0,1,1]
blue= [0,1,0,0,1,1,0,1]
; load color table - change first 8 entries
tv!ct,255*red,255*green,255*blue
; open a window
window,0,xpos=50,ypos=50,xsize=600,ysize=400,title='Using Colors'
; set axes, scales, colors with out plotting
; set background to white (instead of black) and draw axes, etc
; in black instead of white - better for printing
plot,x,y,xtitle='Weeks',ytitle='Popularity', $
      title='Should He Resign?',xrange=[0.0,10.0], $
      yrange=[0.0,60.0], $
      color=0, background = 1, /nodata
; plot calculated values - use symbols to mark discrete set
;make a plot of the first set of data - use color red
oplot,x,y,color=2,psym=-2
;Note how we use $ in the above command to
;continue onto the next line
;psym chooses a symbol type
;negative sign tell IDL to connect symbols with lines
;Add the second set to the plot
oplot,x,y2,color=3, psym=-6 - use color green
;end of the program - don't forget this!
end

```

In lab/class we will discuss this program line-by-line so that everyone understands the purpose of each line and exactly what each line accomplishes.

Exercise #4

We can generate a set of data with a random error about some mean value with the commands

```

num=100
spr=0.1
data=5.0+spr*(1.0-2.0*randomu(seed,num))*randomu(seed,num)

```

What is this line doing?

Definitions:

$$x_{mean} = x_{average} = \langle x \rangle = \frac{1}{N} \sum_{j=1}^N x_j$$

$$\sigma = x_{standard_deviation} = \sqrt{\frac{1}{N} \sum_{j=1}^N (x_j - \langle x \rangle)^2} = \sqrt{\frac{1}{N} \left(\sum_{j=1}^N x_j^2 - 2\langle x \rangle \sum_{j=1}^N x_j - \langle x \rangle^2 \right)} = \sqrt{\langle x^2 \rangle - \langle x \rangle^2}$$

Write a program which computes the mean and standard deviation of the data. Use FOR LOOPS.

Exercise #5

Eliminate the FOR LOOPS in program #4 using the IDL TOTAL function.