Getting started with GDL

::sven geier::2006::Oct

This document will: Give you the quick-n-dirty introduction to GDL so you can start being productive.

This document will not: teach you programming. There's lots of good books about that out there.

(1) Basics:

- Get GDL through gnudatalanguage.sourceforge.net
- install it
- (compile/build gotchas here?)
- make sure the `./src/pro` directory is copied somewhere accessible
- type "gdl" to enter GDL

(2) Variables:

```
GDL - GNU Data Language, Version 0.9
For basic information type HELP,/INFO
GDL>
```

2.1) Scalars:

Variables do not have to be declared. They are "typed as appropriate". The "print" command in GDL is "print", but all tokens are separated by a comma - even from the print command itself.

```
GDL> a = 5
GDL> print,a
       5
```

Note the "print comma a". More info on variables comes from the "help" command:

```
GDL> help,a
A                INT       =        5
```

Again: "help comma a". Here "a" is an INT because that's all that's needed to hold the number 5
INTegers are 16-bit! by default - they range out at 32768:

```
GDL> f = 5000
GDL> help,f
F                INT       =     5000
GDL> help,f*10
<Expression>     INT       =   -15536
```

Here I'm using "f*10" instead of a variable. An expression can go almost anywhere there can be a variable, as long as it is in parentheses:

```
GDL> a = (b = (c = 0))
GDL> print,(b = b+10)
      10
GDL> print,(b = b+10)
      20
GDL> print,(b = b+10)
      30
```

If you need 32-bit integers, you'll have to mark your numbers as "long" (or "ulong" for unsigned numbers):

```
GDL> f = 4000
GDL> help,f
F               INT       =     4000
GDL> help,(f=f*1000uL)
F               ULONG     =      4000000
```

By the way: you don't need any of the spaces around the "=" sign and the "u" or the "L" can be upper or lower case. On 64-bit machines there may also be "L64" or "UL64" data types.

By itself GDL will not promote INTs to FLOATs but perform integer arithmetic. In the 2nd example below, note the "2." which indicates that the "2" here is a float:

```
GDL> f = 5/2
GDL> help,f
F               INT       =        2
GDL> f = 5/2.
GDL> help,f
F               FLOAT     =       2.50000
```

2.2) Arrays

GDLs power lies very much in its array handling:

```
GDL> a = [1,4,2,3]
GDL> help,a
A               INT       = Array[4]
GDL> print,a
       1       4       2       3
GDL> b = a^2
GDL> print,b
       1      16       4       9
```

Arrays behave like expected from math.

```
GDL> c = a+b
GDL> print,c
       2      20       6      12
```

Matrix product is written as "#":

```
GDL> print, a#c
           2           8           4           6
          20          80          40          60
           6          24          12          18
          12          48          24          36
GDL> print, (a#c)#b
         454        1816         908        1362
```

Arrays can also be created like this:

```
GDL> s = intarr(100)
GDL> t = fltarr(50)
GDL> g = lonarr(200,/nozero)
```

Where the "/nozero" means the array is not initialized and its contents are undefined. They may be zeros.

Another powerful way to initialize an array is with the "index generator", which returns an array of the given type, filled with the (zero-based) indices of the elements:

```
GDL> a = indgen(10)
GDL> print,a
       0       1       2       3       4       5       6       7       8
9
```

Individual elements or ranges in an array can be subscripted using round or square parentheses, the latter are strongly preferred, though (because round ones are for function calls):

```
GDL> print,a[4]
       4
GDL> print,a[4:6]
       4       5       6
GDL> a[0:3] = a[5:*]
GDL> print,a
       5       6       7       8       4       5       6       7       8
9
```

Here the "*" means "however many more elements are needed". Multi-dimansional arrays work basically the same. Note that slicing rows or columns produces row- or column-verctors:

```
GDL> b = findgen(4,4)
GDL> print,b
      0.00000      1.00000      2.00000      3.00000
      4.00000      5.00000      6.00000      7.00000
      8.00000      9.00000      10.0000      11.0000
      12.0000      13.0000      14.0000      15.0000
GDL> print,b[2:*,3]
      14.0000      15.0000
GDL> print,b[3,2:*]
      11.0000
      15.0000
```

("f"indgen for floats, "d"indgen for double precision, "l"indgen for long ints, "b"indgen for bytes, "s"indgen for strings...)

(3) Plotting

```
GDL> plot,sin(findgen(200)/100*!pi)
```

Here findgen(200)/100 gives a vector ranging from {0..2}. The !pi is a system variable which unsurprisingly equals 3.141... (There is also a !dpi which has a double-precision version of that). All system variables start with a "!".

If plot has only one array as an argument, that array represents the y-values. If there are two arrays, then they are x and y respectively:

```
GDL> x=findgen(150)/50
GDL> plot,x,sqrt(x)
```

The 'plot' command takes a metric gazillion of keywords - help,/lib will list some of them. You can also plot into an already-established plot-frame by using "oplot" (as in "overplot"):

```
GDL> plot,x,cos(x),ytitle='cos(x)',xrange=[.5,.9],yrange=[.5,1]
GDL> oplot,x,sin(x),color='1ff'x
```

The expression 'ddd'x is simply an easy way to express hex numbers. Can be used wherever decimals can be used. 'dddd'o makes it octal. Note that that can sometimes lead to confusion when a string is specified somewhere that starts with a zero.

You plot into a file like this:

```
GDL> set_plot,'ps'
GDL> plot,x,cos(3*x)*exp(-x),xtitle='x',title='dampened cosine'
GDL> device,/close
GDL> set_plot,'x'
```

As of this writing, the generated postscript file is always prefixed by an empty page - a minor bug, I suppose. If you have gv on your system, you could now preview your plot like this:

```
GDL> $gv gdl.ps
```

where the "$" executes your linux shell command.

Slightly fancier:

```
GDL> s = sin(findgen(100,100)/1000)
GDL> for j=0,99 do s[j,*] = s[j,*]*cos(j/16.)
GDL> surface,s
```

Note the for-loop in there: for var=index1,index2[,stepsize] do {command}
(this will become more powerful a few sections down). Labeling can be done like this:

```
GDL> plot,[0,1]
GDL> xyouts, .3,.6,"data!"
```

By default, data points are drawn as dimensionless points, connected by lines. Other things are possible:

```
GDL> s=randomn(seed,10)
GDL> plot,s,psym=4
GDL> oplot,s,color=50000
```

Randomn makes normally distributed random numbers, mean=0, sigma=1. There is also randomu which makes uniformly distributed ones (in the interval {0..1}).

A special number to use for "psym" is 10, which draws the points as horizontal lines and connects them for a histogram-like plot:

```
GDL> s=randomn(seed,10000)
GDL> t = histogram(s,binsize=.25,omin=omin)
GDL> plot,findgen(100)*.25+omin,t,psym=10,xrange=[omin,max(s)]
```

There's a lot of new functionality in these three lines, have a play at it.

(4) File access

Not yet mentioned is the "format" parameter to the 'print' statement. It works pretty much like FORTRAN:

```
GDL> f = 99.3
GDL> print,f,format='(i)'
          99
GDL> print,f,format='(i5)'
   99
GDL> print,f,format='(i5.5)'
00099
GDL> print,f,format='(f8.2)'
   99.30
GDL> print,f,format='(e8.2)'
9.93e+01
```

Etc. "Z" is hex-output. Files can be read or written to using this kind of formatting, default formatting or unformatted (binary) I/O. Files are opened with "openw" (for writing) or "openr" (for reading only).

```
GDL> openw,unit,'testfile.dat',/get_lun
GDL> print,unit
          64
GDL> printf,unit,indgen(6)*3+9
GDL> free_lun,unit
GDL> $cat testfile.dat
       9      12      15      18      21      24
```

Could use "format" there to format these in specified columns or such.

```
GDL> openr,unit,'testfile.dat',/get_lun
GDL> readf,unit,d
GDL> free_lun,unit
GDL> print,d
      9.00000      12.0000
      15.0000      18.0000
      21.0000      24.0000
```

Writing/reading unformatted (binary) data uses "readu" and "writeu". I.e. if "unit" is open for writing one could do something like this:

```
GDL> writeu,unit,'12345678'x
GDL> free_lun,unit
GDL> $od -Ax -x testfile.dat
000000 5678 1234
000004
```

Several statements can be appended to each other with the "&" sign so they can be re-run together with the "arrow up" function:

```
GDL> print,f & print,(f = f*5)
      99
     495
```

The "@" will read lines of input from a file and execute it:

```
GDL> $echo "print,primes(10)" > testfile.dat
GDL> @testfile.dat
% Compiled module: PRIMES.
           2           3           5           7          11          13
          17          19          23          29
```

This can be used for some simple scripting, but for anything longer than a few lines the section on programs should be studied.

(5) Strings

Should be mentioned somewhere, so I'll do it here: most of the time " and ' are fairly interchangable.

```
GDL> k = 'test' + " something"
GDL> print,k
test something
```

Length need not be specified when making string arrays:

```
GDL> s = strarr(5)
GDL> s[3]="hello"
```

etc.

string(5) is "   5" with default formatting, but string() accepts formats:

```
GDL> s = string(92,format='(z4)')
GDL> print,s
  5c
```

Converting a byte-array into a string means the ascii characters of the string will be the bytes in the array:

```
GDL> b = bindgen(5)+49b
GDL> print,string(b)
12345
```

Note the use of "49b": if this were just "49" it would be an int and the whole array would be ints:

```
GDL> b = bindgen(5)+49
GDL> print,string(b)
      49        50        51        52        53
```

The format code for a string is (A). The format code "$" means "no newline". E.g.:

```
GDL> for i=0L,1e6 do print,i,string(byte(13)),format='($,i,(A))'
```

Note the "0L" as otherwise this would only be able to count to 32768

(6) Programs

The "@" facility is nice, but as each line is read and executed individually, one cannot use it to produce code that spans lines. For that, there are programs:

In the simplest case, an IDL/GDL program is a text file ending in ".pro". It contains lines somewhat like above, the last line must be "end", comments start at a semicolon. In a ".pro" file, loops and conditionals can be expanded into blocks, by replacing any one statement with a begin...end pair.
In this simplest case, this file is executed by typing ".run filename[.pro]".

For example put this into a file and call it "test.pro":

```
; true-color image:
img=fltarr(3,300,300)

for i=0,299 do begin ; "begin" replaces the command that would go here
    img[2,i,*]=300*abs(sin(i/150.*!pi))
    img[0,*,i]=300*abs(sin(i/100.*!pi))
    for j=0,299 do begin ; again
        img[1,i,j]=300*abs(sin(i/300.*!pi*cos(j/75.*!pi)))
    endfor
endfor

window,xsize=300,ysize=300
tv,img,/true
end
```

You can then run this routine from the command-line with ".run test". It will read the file, compile it and run it. The way this is done here, it will retain all the variables after it finishes, so you could work with them from here on in:

```
GDL> print,i
     300
GDL> img[0,*,*]=transpose(img[0,*,*])
GDL> tvscl,img,/true
```

Etc. This also shows the way you turn a single-line statement into a block statement by replacing the command in the for... statement with a "begin {any amount of code} end" pair. GDL allows but does not require to name the "end" appropriate for the given loop -- in this case "endfor". It would be allowed to just say "end" here. It is recommended to use "endfor", "endif", "endwhile" etc, though, since that'll generate an error if you use the wrong one, which is quite helpful in debugging.

Thus an if-statement reads

```
 if condition then command [else other-command]
```

but in a program it could expand to

```
if condition then begin
  command(s)
endif [else begin
  other-command(s)
endelse]
```

etc.

In the next step, our routine can be encapsulated by prefixing the ".pro" file with the line

```
pro routinename, var1, var2
```

which now makes it into a routine that can be called by name and have parameters passed to it.

Here's a trivial one that subtracts one from a variable (I'm sure you can find a better example):

```
pro dec, x
  x = x - 1
  return
end
```

called like this:

```
GDL> i=5
GDL> dec, i
% Compiled module: DEC.
GDL> print,i
       4
```

This means parameters are handed down and back in IDL.

Finally there's a type of program that returns a value. I.e. a function. Works like in all other languages:

```
function factorial, x
  if n_elements(x) ne 1 then stop, "Need a scalar input here!"
  return,product(1+indgen(x))
end
```

produces this:

```
GDL> print,factorial(7)
% Compiled module: FACTORIAL.
       5040.0000
```

Note that these are only compiled the first time you call them. If you find your function lacking and you change it and want to use the new version, you'll have to issue a ".compile fname" by hand to force GDL to recompile.

(7) Control structures

GDL understands the following:

```
if (condition) then (command) [else (command)]
```

Numerical comparators are "lt", "gt", "eq". Expressions are true if their LSB is set (i.e. if they are odd) if integer. Floats are true if nonzero, strings if non-empty. AND, OR, NOT work as expected.

```
for (var)=(start),(end),(step) do (command)
```

If var doesn't have enough range to go all the way to (end), you're hosed.

```
while (condition) do (command)
repeat (command) until (condition)
```

Should be self-explanatory. "command" can always be expanded into a begin...end pair.

```
case (var) of
 (choice-1): (command-1)
 [(choice-2): (command-2) [...]]
[else: (command-else)]
endcase
```

The case..endcase pair can also be called switch..endswitch - in that case all the commands beginning with the matching one will be executed (as opposed to *just* the matching one).

```
break
```

will exit the current (innermost) loop, case etc.

```
continue
```

will immediately start the next iteration of the curent loop.

```
label:
goto label
catch
on_error
on_ioerror
```

do what you expect.

(8) Array power

Most operations that require loops can be avoided in GDL by operating directly with the arrays over which one would loop in other languages. This speeds up operation dramatically: whenever you think IDL/GDL is slow, you should see where you can compress any for-loop .

For example if one wanted to set all numbers less than -.5 in an array of random numbers to zero, on could do something like

```
for i=0l,n_elements(arr)-1 do if arr[i] lt -0.5 then a[i]=0
```

but a much faster option is

```
a[where(a lt -0.5)] = 0
```

`"where"` alone can probably eliminate 75% of all cases where other languages use loops. Note the loop-less implementation of a factorial up in section (5). Other powerful functions are

`v=sort(arr)` returns a vector v of indices such that arr[v] is sorted.
`v=total(arr)` sum of all elements
`v=uniq(arr)` remove duplicate elements

An array containing the numbers from 0 to 5000 in random order would be obtained by

```
arr=sort(randomn(seed,5000))
```

If 'names' and 'dates' are two arrays to be sorted by date, one would

```
s = sort(dates)
dates=dates[s]
names=names[s]
```

There is a histogram function, that counts the number of instances of the individual elements in an array. It can be used for many things other than scientific graphs. For example a file could be read into a bytarr, histogram(b_arr) computed and the value in the 10th element of the result would be the number of linefeeds (i.e. the number of lines) in the file.

The most powerful aspect of histogram() is probably the reverse_indices parameter. Check it out.

(9) ... (more to come) ...