file: idl-simple-manual.txt = introduction to IDL basics
last: Jun 16 2021 Rob Rutten Deil

SIMPLE IDL INSTRUCTION FOR ASTRONOMY STUDENTS

Robert J. Rutten

Lingezicht Astrophysics Deil
Institutt for Teoretisk Astrofysikk Oslo

This compact IDL tutorial is a beginner's introduction to IDL, showing how to do simple calculations, make plots, write IDL programs.

It consists of a didactic sequence of IDL commands that you should try out on the IDL command line. It starts after an extensive introduction with general information and weblinks.

There are parallel txt, pdf, and html versions of this manual at

   https://robrutten.nl/Manuals.html

The html and pdf versions have active weblinks.

This manual was written in the early 1990s for second-year astronomy students at Utrecht University doing the "Stellar Spectra" exercises at

   https://robrutten.nl/Exercises.html

I irregularly add more IDL fads and fallacies that I stumble upon.

INTRODUCTION TO THIS INSTRUCION

Why use IDL?

IDL is an interactive programming language with the following advantages:
 - programming language, not a package: make up your own stuff, experiment
 - interactive "interpreter": test statements and tricks on the command line
 - array notation: c = a + b handles multi-dimensional arrays (images, movies)
 - journaling: keep a log of all trials, then pick out what worked best
 - save/restore: store a complete session to share with others

Although IDL is commercial and licenses are excessively expensive, it was the mainstay in astronomical image processing - but public Python is taking over. IDL was indispensable in solar physics through the extensive SolarSoft library at

   http://www.lmsal.com/solarsoft

```
 but
SunPy is on its way to replace this.  (I haven't tried Fawlty Language
= a public IDL replacement.)

My habits
---------
I run ancient IDL 6.4 (2007) under Ubuntu linux in the emacs IDLWAVE shell.
I often use SolarSoft routines from
   https://www.lmsal.com/solarsoft

I sometimes use Coyote Graphics "cg" routines from
   http://www.idlcoyote.com/documents/programs.php

I habitually swear at IDL because:
 - it has far to many counter-intuitive idiosyncracies
 - its figure layout differs hardware-dependently between screen and ps
 - its figure annotation remains a hassle even with textoidl
 - it has confusing plot parameter choices between graph area and plot area
 - it counts my fingers 0 to 9
 - its array notation [column,row] describes images, not matrices
 - its CNTRL d is not next-character-delete as in Emacs but kills the session
 - its CNTRL c does not stop program execution but may kill the session
 - it does not have command-line tab completion (except in IDLWAVE)
 - it does not have a comprehensive !! system parameter reset
 - it started prefering square brackets for array indices far too late
 - its error messages are primitive and often bewildering
 - my "life-long license" is nearly impossible to re-activate

Other IDL manuals
-----------------
The online help (type ? in an IDL session) is reasonably complete but
most examples are too simplistic.  The IDL 6.4 help GUI is primitively
browser-like.

IDLWAVE accesses the IDL help files by keystrokes on procedure names.
Extensive manual (but assuming nontrivial knowledge of emacs) at
   http://www.gnu.org/software/emacs/manual/html_mono/idlwave.html

A searcher that can also search IDL Google groups resides at
   http://www.physics.emory.edu/~weeks/lab/searchidl.html


Explanatory comment blocks:
  Many user-supplied routines (functions, procedures, full programs), as
  those in the SolarSoft, Astronomy, and Coyote IDL libraries, start
  with explanatory comment blocks between ;+ and ;- lines.
  You can read these by typing
    doc_library, 'routinename'
  at the IDL prompt, but it may be more convenient to produce a html help
  tree that you can inspect with your prefered html browser with, for example:
```

```
  mk_html_help,'~/idl/coyote','~/idl/help/coyote.html'
  (the Coyote library contains this as file: program.documentation.html).
  IDLWAVE opens such ;+...;- comment blocks with keystrokes.
  I prefer to use my misclib sp.pro ("show program") to open them in a
  separate editor window.

Weblinks:
  Numerous url's for astronomical IDL are collected at
   http://idlastro.gsfc.nasa.gov/other_url.html


Books:
  David Fanning: "Traditional IDL graphics" (2011)
  David Fanning: "IDL Programming Techniques, 2nd Edition" (2000)
  Lilian Gumley: "Practical Idl Programming"
  Ken Bowman:    "An Introduction to Programming with IDL"

IDL routine libraries
---------------------
David Fanning's coyote library, including 2011 cg routines used below:
   http://www.idlcoyote.com/documents/programs.php


textoidl.pro: get the version under pro/plotting in the Sloan library at
   http://code.google.com/p/sdssidl/downloads/list


Astronomy IDL library (not used here; it has been converted to cg):
   http://idlastro.gsfc.nasa.gov/homepage.html


SolarSoft = "ssw" = solar physics IDL library:
   https://www.lmsal.com/solarsoft


IDL startup
-----------
IDLWAVE for Emacs
  Recommended modus of IDL operation, offering many keystroke shortcuts
  and debugging options:
   http://www.gnu.org/software/emacs/manual/html_mono/idlwave.html


  The IDLWAVE settings in my own .emacs file are shown at
   https://robrutten.nl/Recipes_linux_unix.html

  My setup defines hyperkey+mouse-middle-click to call my misclib sv.pro
  ("show variable') to diagnose the variable content as print or plot or
  movie.

Solarsoft startup
  In my Ubuntu linux I use a shell script "idl" to always run ssw:
```

```
    #!/bin/csh
    setenv SSW /usr/local/ssw                       # if ssw stuff sits here
    setenv SSW_INSTR "sot aia hmi trace ontology"  # select instruments
    source $SSW/gen/setup/setup.ssw
    sswidl

IDL startup code to resolve library clashes
  SolarSoft took Coyote routines long ago and changed them without
  name change.  The worst clasher is "linkedlist__define.pro".  The remedy
  is to make IDL search the coyote library before the ssw libraries.
  SolarSoft puts its ssw libraries before any others, so this cannot be done
  in a .login file or a shell resource (.bashrc, .cshrc) file,
  but needs the following use of Coyote's "addtopath.pro" in your
  "idlstartup.pro":
    cd, '/home/usr/idl/coyote',current=thisdir    ; adapt to your coyote path
    addtopath
    cd,thisdir
    cd,current=workdir   ; repeat for your actual working dir
    addtopath,workdir    ; routines in your workdir now override any others
  NB: in "idlstartup.pro" I also have, following page 47 in Fanning 2011:
    device,retain=2,decomposed=0              ; indexed colors (255 only)
    window,xsize=10,ysize=10,/pixmap,/free    ; initializing window
    wdelete,!d.window                         ; to avoid empty white window

Format of this instruction
--------------------------
  IDL executes on the command line when you hit return ("interpreter").
  This makes it easy to try new statements and statement sequences.
  The up cursor arrow brings back earlier commands.

  The main body of this instruction consists of a didactic sequence of
  command-line entries.  Simply enter the IDL statements consecutively
  on the IDL> command line (type or copy-paste).  Predict their action
  before you enter them!  Many are goodies but some will surprise you
  negatively.

  The end of the instruction describes program structure, parameter
  passing, session saving, etc.

  Enjoy!

                ================================
                START OF THE ACTUAL INSTRUCTION
                ================================

IDL MATH BASICS
===============
```

```
help
----
  ? [search term]       ; IDL's help: inspect some IDL routines and concepts

number games
------------
  print,3*5             ; semicolon = comment, IDL skips the rest of the line
  a=3*5                 ; no variable declaration needed
  a = 3 * 5             ; add spaces as you like
  help,a                ; show nature and value of this variable
  help,A                ; IDL is case-insensitive, shows variables in caps
  whatever_name_you_like$like_this_perhaps = a       ; _ and $ are permitted
  print,whatever_name_you_like$like_this_perhaps     ; no spaces, +, -, *
  spectrum_AR10910=1    ; variable names must start with alphabetic character
  d=32767               ; "short" integers run from -32768 to + 32767
  print,d+1             ; did you predict this value?
  print,d+1.            ; IDLWAVE: SHIFT mouse2 = print variable under cursor
  print,2^15            ;
  print,2.^15           ; why is the integer word length not 16 bits?
  ? integer             ; check the other number formats
  print,32767001        ; long integer, sign+31 bits
  print,3276700ul       ; unsigned long integer, 32 bits
  print,3276700ull      ; unsigned long long integer, 64 bits
  print,3/5
  print,3/5.            ; operation with one float makes the result a float
  print,2^15.
  a=[1,2,3,4,5,6]       ; IDL variables can be 1-8 dimension arrays
  a=[0,a,7]             ; lengthen this 1D "vector" by adding value(s)
  print,a,1E6*a         ; single precision: 6 significant digits, < 10^38
  print,a,1D6*a         ; double precision: 16 significant digits
  print,a,1/a           ; divide by 0 gives error message without stop
  print,a,1./a
  print,a,a^2
  print,a,alog10(10^a)                  ; NaN = Not a Number
  print,a,alog10(10^float(a))
  a=1.*a                                ; convert into float
  print,a,alog10(10^a)
  print,a,alog(exp(a))
  print,a,acos(cos(a))                  ; a in radians
  print,a,acos(cos(!pi/a))*180./!pi     ; !something is a system variable
  print,!dpi                            ; double precision value of pi
  print,!dtor                           ; so what is this?
  print,a,acos(cos(!pi/a))*!radeg       ; another one
  print,a,a mod 2
  print,fix(!pi)        ; fix = entier to short integer
  print,long(!pi*1E8)   ; long = entier to long integer
  b=sqrt(a)             ; type of b is defined through its assignment
  a=3
```

```
  if (a=1) then print, 'yes, a=',a else print,'no, a=',a      ; IDL quirk
  a=3                                                         ; try again
  if a eq 1 then print, 'yes, a=',a else print,'no, a=',a     ; better
  if (a eq 1) then print, 'yes, a=',a else print,'no, a=',a   ; nicer
  if ~(a eq 1) then print, 'yes, a=',a else print,'no, a=',a  ; ? ~ operator
  help                  ; help without variable shows all variables

string manipulation
-------------------
  print,'b=',b          ; 'something' is a string
  pathfile='rootdir/homedir/ownerdir/workdir/todaydir/thisfile.txt'
  print,strmid(pathfile,strpos(pathfile,'/',/reverse_search)+1)  ; IDL...
  print,file_test('path/file')          ; check file exists
  fileonly=file_basename(file)
  print,str_match(file,'substring')     ; does filename contain substring?
  newstring=str_replace(string,'-','.')  ; replace all - by .
  print,'b = ',string(b,format='(f5.2)')                ; ancient Fortran
  print,'b = ',strmid(string(b,format='(f5.2)'),1)      ; IDL...
  print,'b = ',strmid(string(b+1e3,format='(f7.2)'),1,6) ; with zero padding
  print,'b = ',ntostr(b)                ; that's easy!  Google ntostr.pro
  print,'b = ',ntostr(b,format='(f5.2)') ; better spaces removal
  print,'b = ',trim(b)                  ; SSW alternative
  print,'b =',trimd(b,3)                ; my own number printer, 3 decimals
  c=!pi^50                              ; make a large number
  print,c,c,c,c,c,c,c,c,c               ; wide printout
  print,ntostr([c,c,c,c,c,c,c,c,c,c],format='(20E10.3)')  ; compact printout
  print,ntostr([c,c,c,c,c,c,c,c,c,c],format='(G15.5)') ; chooses float or exp

one-dimensional arrays
----------------------
  a=bytarr(100)         ; define a as byte array a[0],..,a[99]=0
  a=intarr(100)         ; define a as integer array a[0],..,a[99]=0
  a=fltarr(100)         ; define a as floating number array a[0],..,a[99]=0.0
  a=dblarr(100)         ; double-precision float array = 0.0000000
  a=a+1                 ; now they are all 1.0000000
  for i=0,19 do a[i]=i  ; remember that IDL starts counting at 0
  a=indgen(20)          ; same thing: a=[0,1,....,19] without a[] declaration
  print,a[0],a[19]      ; always mind the virtual startoff finger
  print,a[10:19]
  print,a[*]            ; same as print,a and as print,a[0:19]
  print,moment(a)       ; mean, variance, skewness, kurtosis (set /double?)
  b=sqrt(a)             ; check that b is a float array - why?
  print,a+b
  c=b                   ; define float array the same size as a and b
  for i=0,19 do if (b[i] gt 3) then c[i] = a[i] + b[i] else c[i] = a[i]
  print,c
  print,a+b*(b gt 3)       ; the same, processes faster, needs no declaration
  print,a+b>3              ; beware: gives 3 or a+b where (a+b)>3
```

```
  print,a+(b>3)            ;         gives a+3 where b<=3, a+b where b>3
  print,a+(b gt 3)         ;         gives a, adding 1 where b>3
  print,a+b gt 3           ;         gives 0 for (a+b)<3, 1 for (a+b)>3
  print,a+b[where(b gt 3)] ;         gives b[10:19] added to a[0:9]
  print,max(1,2,3)         ; did you predict the answer?
  print,max([1,2,3])

two-dimensional arrays
----------------------
  ar = [[1,2,3],[4,5,6]]   ; integer [3,2] array
  print,ar                 ; 1st index = column number, "runs fastest"
                           ; 2nd index = row number
  print,ar[0],ar[0,0]      ; mind the virtual finger
  print,ar[0,*]            ; * = all values of this index
  print,n_elements(ar)     ; predict all these
  print,total(ar)          ; for large arrays set /double
  print,shift(ar,-1)
  print,transpose(ar)
  print,reverse(ar)
  print,invert(ar)         ; needs square array
  ar=ar+1                  ; add 1 to each array element
  ar=temporary(ar)+1       ; idem but in place requiring less memory
  vec1=[1,2]
  vec2=[3,4]
  ar=[[vec1],[vec2]]             ; simple 2x2
  print,ar
  print,ar*vec1                  ; f*g  = f[i,j]*g[i,j]
  print,ar#vec1                  ; f#g  = columns x rows (IDL habit)
  print,ar##vec1                 ; f##g = rows x columns = transpose(f#g)
  print,ar#reverse(ar)           ; predict or check manually
  print,ar##reverse(ar)          ; predict or check manualy
  print,invert(ar)#ar            ; unit diagonal, OK
  ar=[[1,2,3],[4,5,6],[7,8,9]]       ; now 3x3 without virtual finger
  ar=indgen(3,3)+1                   ; the same
  print,invert(ar)#ar                ; should be unit diagonal but isn't
  arinv=invert(ar,status,/double)    ; try again
  print,arinv#ar                     ; as bad in double precision
  print,status                       ; status=1: singular, so invalid

three-dimensional arrays
------------------------
  ar=indgen(3,4,5)+1          ; let's say 3x4 px frames in a 5-frame movie
  print,ar                    ; successive indices run slower
  ar3=ar(*,*,2)               ; third movie frame
  print,total(ar)             ; sum all elements
  print,total(ar,1)           ; (4,5) row sums = sum over other dimensions
  print,total(ar,2)           ; (3,5) column sums
  print,total(ar,3)           ; (3,4) frame sums
```

```
  sizear=size(ar)
  print,sizear       ; nr dims, dim1, dim2, dim3, type (integer), nr elements
  mean=total(ar,3)/sizear(3)  ; temporal mean of this movie
  xslice=ar[*,0,*]            ; distill (x,t) timeslice at y=0
  help,xslice                 ; oops, still 3D array
  xslice=reform(xslice)       ; reform removes degenerate dimensions
  help,xslice                 ; 2D array now
  br=[[[ar]],[[ar]],[[ar]]]   ; what is this?
  help,br

     ; more of the same / soortgelijks / und so weiter / ibid
  ar=indgen(6,5,4,3,2)+1
  print,ar
  print,size(ar)

free array to regain memory space
---------------------------------
  undefine,arra,arrb,arrc,..  ; regain memory anywhere (cg program)
  delvar,arra                 ; regain memory but only in main part
  ar=0                        ; doesn't regain memory but leaves a hole


GRAPH PLOTTING
==============

basic plot
----------
  x=findgen(100)          ; float array x=0., 1., ...., 99.
  plot,sin(x/10)          ; 10 doesn't have to be 10. since x is float
  y=sin(x/5.)/exp(x/50.)  ; but I like float specification for safety
  plot,y                  ; plot,x,y uses array index for x if not given
  plot,alog10(x),y        ; x and y may differ in array size
  oplot,alog10(x),y^2     ; over-plots in existing graph
  plot,alog10(x),y^2+10   ; too much emptiness in this graph
  plot,alog10(x),y^2+10,/ynozero   ; /ynozero is alternative for ynozero=1
  plot,abs(fft(y,1)^2),/ylog       ; power spectrum on logarithmic scale
  plot_io,x,abs(y)+0.1)   ; log-linear plotter, not in the IDL help?
  erase                   ; wipe current plot window
  wdelete                 ; kill current plot window
  while !d.window ne -1 do wdelete,!d.window ; kill all IDL windows ("easy")

Coyote cg window alternative
----------------------------
  cgplot,x,y,/window ; resizable window, "save-as-postscript" clicker

plot beautification
-------------------
  plot,x,y,psym=1          ; defined for psym=1-7,10; try them out
```

```
                          ; something=something: optional "keyword" parameter
                          ; check PLOT (? plot); check GRAPHICS KEYWORDS
  plot,x,y,psym=-4        ; plot both curve and diamonds at sample values
  plot,x,y,linestyle=1    ; defined for linestyle=0,...,5, try them all
  oplot,x,y*2,linestyle=2 ; overplot another graph in the same frame
  plots,20,70,psym=2,symsize=1.5      ; mark location with asterisk
  plots,[20,70],[-0.5,+0.5]           ; overplot line segment [x1,x2],[y1,y2]
  plots,[50,50],[-1E10,1E10],noclip=0 ; overplot line cut at edges (NOT /clip)
  plot,x,y,xtitle='x axis',ytitle='y axis',thick=2,xthick=2,ythick=2,$
      charthick=2,charsize=2                    ; $ extends to next line
  ytitle=textoidl("sin(x/\alpha) e^{-x/\beta}")
  print,ytitle          ; !U=up, !D=down, !N=normal, !7=Greek, !X=entry font
  angstrom=textoidl("\AA")
  angstrom=string(197B)  ; alternative = "byte constant" character code
    ; find symbol codes such as this by Googling <table isolatin1>
    ; but they may not be valid in the PostScript font you choose
    ; why the &$#@$% doesn't IDL accept full latex strings for all fonts?
  set_plot,'ps'
  angstrom=cgsymbol("angstrom") ; Coyote cg, probably most robust
  set_plot,'x'
  xtitle='x  ['+angstrom+']'
  print,xtitle
  plot,x,y,xtitle=xtitle,ytitle=ytitle,charsize=2
  alpha=5                                       ; let's add annotation
  alphaspec=textoidl("\alpha = ")+strtrim(string(alpha),2) ; not so simple...
  alphaspec=cgsymbol("alpha",/ps)+' = '+ strtrim(string(alpha),2) ; cg for ps
  alphaname=strlowcase(scope_varname(alpha))  ; get variable name as string
  alphaspec=greek(alphaname)+' = '+ strtrim(string(alpha),2)
  xyouts,80,0.7,alphaspec,charsize=2             ; x,y in data units
  xyouts,0.7,0.8,/norm,alphaspec,charsize=2      ; x,y in window size units
  plot,x,y,xrange=[-10,+110],yrange=[-1.5,1.5]    ; your choice axis lengths
  plot,x,y,xrange=[-10,+110],yrange=[-1.5,1.5],xstyle=1,ystyle=1
                          ; now the axes obey your ranges exactly

plot beautification in a Coyote cg window
-----------------------------------------
  cgplot,x,y,xtitle=xtitle,ytitle=ytitle,charsize=2,$    ; similar to above
    /window,$       ; resizable window
    aspect=2./3,$   ; fixed aspect ratio
    psym=-15,$      ; many more choices; see doc_library,'symcat'
    /_extra,thick=2,xthick=2,ythick=2,charthick=2   ; any plot keywords
  cgplot,x,y*2,/overplot,/add,color='darksalmon',thick=5,linestyle=2
    ; overplot is now an option of cgplot (add /add), not a separate routine
    ;   cgplot can also easily color curves, symbols, etc
    ;   select cgcolor name from palette: color = cgcolor(/selectcolor)
    ;   but oops: sticky, makes colors add up spoiling your next display
    ;   see also doc_library,'cgcolor' or browse program.documentation.html
  cgtext,80,0.7,alphaspec,charsize=2,/addcmd      ; replaces xyouts
```

```
PostScript figure with traditional IDL
--------------------------------------
  set_plot,'ps'                 ; change plot output to postscript format
  device,filename='demo1.ps'    ; the plot commands now write to this file
  plot,x,y,xtitle=xtitle,ytitle=ytitle,thick=2, xthick=2,ythick=2,$
      charthick=2,charsize=2              ; redo all the above
  xyouts,80,0.7,alphaspec,charsize=2      ; idem
  device,/close                 ; done, write postscript file
  set_plot,'x'          ; back to output on Unix/linux/MacOS Xwindows screen
  ; set_plot,'win'      ; back to output on a Micro$oft Windows screen
  ; help,/device        ; /device is the same as device=1 (enable)
  $gv demo1.ps                  ; starting $ on command line escapes to shell
  filename='demo1.ps'           ; make it a variable for
  spawn,'gv '+filename          ; generic shell escape, also in a program

  OOPS! ..|.. IDL!  The ps plot differs much from what you had on
  your screen.  The thickness parameters in plot do NOT apply to ps
  output.  Also the charsize multiplier in plot does NOT work.  The
  vertical annotation spacing differs (even hardware-dependently,
  depending on the character pixel matrix).  So, this demo exhibits
  severe IDL shortcomings.  First, there is no clicker or single
  command to obtain ps output that reproduces exactly what you have on
  your screen - you cannot develop a nice on-screen display and then
  hit or command "save as ps".  Instead, you have to repeat the whole
  sequence of plot commands that made your nice on-screen display once
  again for the ps "device", as shown above.  Second, there are
  inconsistencies between such plotting on the screen and in ps, and
  some of these are hardware-dependent.  The awkward upshot is that
  there is not much point in beautifying the on-screen product.
  Instead, you should beautify the ps output, independent of what you
  get on the screen.  Because the plot thickness keywords do not work
  for ps, one then has to muck around with the various !p.thick system
  parameter settings.  These are sticky, so changes must subsequently
  be undone not to get problems later (for example in the next
  on-screen plot).  Similary, the IDL font codes for Greek characters
  differ between the screen and some ps fonts.  Argh...

  However, David Fanning's cg routines with their /window option can
  serve to develop IDL figures on your screen and obtain ps output
  like these (and raster pixmaps) without explicit sequence repeat but
  indeed per clicker or a single command.  The sequence repeat still
  occurs but is hidden within cg routines that call the coyote-library
  "ps_start" and "ps_end" routines internally.  Fanning added
  "evalkeyword" and "evalparams" options to provide run-time
  evaluation for things that differ between devices, such as thickness
  keywords and Greek characters.  See below.  They work nicely for
  simple figures, but for elaborate ones you may prefer to go back to
```

```
  the traditional repeat-sequence approach (I usually do).

  Below I first demonstrate the traditional way of making postscript
  graphs through repeating the entire plot sequence, then coyote cg
  usage to avoid such repetition.

Postscript figure following Alfred de Wijn
------------------------------------------
   http://www.iluvatar.org/~dwijn/idlfigures
  set_plot,'ps'                                  ; postscript format
  !p.font=1                                      ; true-type fonts
  !p.thick=2 & !x.thick=2 & !y.thick=2           ; & = multiple commands/line
  !p.charthick=2                                 ; reset system default
  xsize=8.8                                      ; cm; this is A&A column width
  ysize=xsize*2/(1+sqrt(5))                      ; aspect golden ratio 1.61803
  filename='demo2.eps'
  device,filename=filename,xsize=xsize,ysize=ysize,/encapsulated,/portrait,$
    /tt_font,set_font='Times',font_size=11       ; fit size to publication font
  ytitle=textoidl("sin(x/\alpha) e^{-x/\beta}")      ; repeat for ps font
  alphaspec=textoidl("\alpha = ")+strtrim(string(alpha),2)   ; idem
  plot,x,y,$
    position=[0.2,0.2,0.95,0.95],/normal,$       ; set margins around plot
    xticklen=0.03,yticklen=0.03*ysize/xsize,$    ; same-length ticks
    xtitle=xtitle,ytitle=ytitle
  xyouts,80,0.7,alphaspec                        ; x,y in data units
  device,/close
  set_plot,'x'          ; back to output on Unix/linux/MacOS Xwindows screen
  !p.font=-1            ; back to default (Hershey) fonts for screen display
  !p.thick=0 & !x.thick=0 & !y.thick=0 & !p.charthick=0  ; reset defaults
  spawn,'cat '+filename+$                              ; replace irritating
    '| sed "s|Graphics produced by IDL|'+filename+$    ; IDL plot banner
    '|" >  idltemp.ps; mv idltemp.ps '+filename        ; with the file name
  spawn,'gv '+filename              ; set gv to "watch file" for rewrites
   ; NB: textoidl doesn't give true-type font but at least it works in ps;
         for Greek it has to be run again, now in the ps device environment
   ; NB: I minimize the bounding box later with epstopdf, pdfcrop, pdf2ps

PostScript figure with Coyote ps_start and ps_end
-------------------------------------------------
  xsize=8.8 & ysize=xsize*2/(1+sqrt(5))
  ps_start,filename='ctdemo2.eps',font=1,tt_font='Times',$
    /nomatch,xsize=xsize,ysize=ysize,/metric,/encapsulated,charsize=0.9
             ; default ps thicknesses are temporarily reset to 2
  !p.thick=3 & !x.thick=3 & !y.thick=3 & !p.charthick=3  ; if you prefer 3
  ytitle=textoidl("sin(x/\alpha) e^{-x/\beta}")      ; textoidl repeat for ps
  alphaspec=textoidl("\alpha = ")+strtrim(string(alpha),2)   ; idem
  plot,x,y,$
    position=[0.2,0.2,0.95,0.95],/normal,$
```

```
    xticklen=0.03,yticklen=0.03*ysize/xsize,$
    xtitle=xtitle,ytitle=ytitle
  xyouts,80,0.7,alphaspec
  ps_end     ; back to screen windows, Hershey fonts, original ! values
  spawn,'gv ctdemo2.eps'

PostScript figure from a Coyote cg screen window
------------------------------------------------
  cgplot,x,y,/window,$
    charsize=2,xtitle=xtitle,position=[0.25, 0.25, 0.9, 0.9],$
    evalkeywords=['thick','xthick','ythick','charthick','ytitle'],$
      thick='(!d.name eq "PS")?5:1',$      ; 5 for ps, 1 for screen
      xthick='(!d.name eq "PS")?5:2',$     ; ps thick because size is large
      ythick='(!d.name eq "PS")?5:2',$     ; PS must be in capitals
      charthick='(!d.name eq "PS")?5:1',$
      ytitle='textoidl("sin(x/\alpha) e^{-x/\beta}")'  ; Greek, redo for ps
  cgtext,0.7,0.8,/norm,$
    'greek(alpha)+" = "',evalparams=[0,0,1],$          ; Greek, redo for ps
    charsize=2,/addcmd
  cgtext,0.77,0.8,/norm,$          ; 0.77 results from manual fitting on ps
    strtrim(string(alpha),2),$                         ; normal parameter
    charsize=2,/addcmd
       ; click on file > save as postscript > ps output; or instead enter:
  cgcontrol,create_ps='cgdemo2.eps',/ps_encapsulated,/ps_metric
  spawn,'gv cgdemo2.eps'

add second axis
---------------
    example of adding a top x-axis with nonlinear scaling with respect
    to the bottom x-axis (in this case mu = cos(theta) over the solar disk
    versus r/R_sun = sin(theta) with theta the viewing angle)
  plot,rvalues,averzones,psym=-4,$
    position=[0.2,0.2,0.8,0.8],$  ; wide margins to accommodate extra axes
    xrange=[0,1],yrange=yrange,$
    xstyle=9,ystyle=1,$      ; no axis along top
    xtitle=textoidl("r/R_{sun} = sin \theta"),$
    ytitle='whatever averzones was about'
  mutickpos=[1.0,0.9,0.8,0.7,0.6,0.5,0.4,0.0]
  muticknames=['1.0','0.9','0.8','0.7','0.6','0.5','0.4','0.0']
  nmuticks=n_elements(mutickpos)-1
  rmuticks=sqrt(1.-mutickpos^2)
  axis,xaxis=1,xticks=nmuticks,xtickv=rmuticks,xtickname=muticknames,$
    xminor=1,xtitle=textoidl("\mu = cos \theta")

add zero to a second axis
-------------------------
    IDL's AXIS routine to generate extra axes has the annoying failure
    that it may not plot the label zero when an axis starts at zero.
```

```
    Below an example how to correct this, plotting functions "tau(height)"
    and "temp(height)", the tau axis at left, the temp axis at right:
  heightrange=[0,2300]
  taurange=[-3,7]
  plotaspect=1.62                                ; golden ratio
  plot,height,alog10(tau),$
    position=[0.2,0.2,0.8,0.95],/normal,$        ; set margins around plot
    xticklen=0.03,yticklen=0.03/plotaspect,$     ; same-length ticks
    xtitle='height  [km]',ytitle='log  (optical depth)',$
    xrange=heightrange,yrange=taurange,xstyle=1,ystyle=9,linestyle=1
  temprange=[0,30000]
  tempscaled=taurange[0]+(temp-temprange[0])/(temprange[1]-temprange[0])*$
    (taurange[1]-taurange[0])                    ; rescale temp to log(tau)
  oplot,height,tempscaled,thick=3                ; overplot temp(height)
  axis,yaxis=1,yrange=temprange,ystyle=1,$       ; dummy axis to get ticks
    ytickinterval=1000,ytitle='',ytickname=replicate(' ',60),$
    ytick_get=tempaxticks
  tempticknames=string(tempaxticks,format='(i5)')
  axis,yaxis=1,yrange=temprange,ystyle=1,$       ; plot temp axis at right
    ytickinterval=1000,ytitle='temperature  [K]',ytickname=tempticknames
```

multi-panel figures

IDL offers !p.multi for stacking multiple plots into one display. Quite cumbersome and non-versatile. Alfred de Wijn has a better recipe at:

http://www.iluvatar.org/~dwijn/idlfigures

I myself never make multi-panel displays with IDL. Instead, I make fully-annotated separate graphs and stack them up in LaTeX, using LaTeX macros to remove superfluous annotation between panels. This way I choose the figure layout only when writing the paper, which makes collaboration in the analysis phase much easier. See:

https://robrutten.nl/rrweb/rjr-edu/manuals/student-report/cutmultipanel.tex

ARRAY/IMAGE PLOTTING
===

two-dimensional array plotting

```
  k=indgen(100)              ; let's make a nice 100x100 array
  f=sin(k/5.)/exp(k/50.)     ; the same f(x) as y(x) above
  g=cos(k/5.)*exp(k/50.)     ; similar function g(y) for the other coordinate
  s=f#g                      ; make an array
  help,s                     ; a 2-dim (100,100) float array
  print,s[0:4,0:9]           ; better use square brackets for array elements
```

```
  plot,s[7,*]               ; plot 8th column (mind the virtual zero finger)
  oplot,s[*,95],linestyle=5 ; overplot 96th row, dashed
  tvscl,s                   ; view as byte-scaled image
    ; Compare the image (in the bottom-left plot corner), graph, and printout.
    ; The first index is the column number, the second index the row number.
    ; IDL's [column,row] is opposite to matrix algebra.  See ? array majority.
    ; IDL's [column,row] fits the notion of an image f(x,y), that's why.
    ; The printout has s[0,0] at the top-left corner, but
    ; the image display has s[0,0] at its lower-left corner ("origin").
  print,minmax(s)           ; show extrema
  print,array_indices(s,where(s eq max(s)))  ; the two plots sample max(s)
  print,s[5:9,94:96]        ; check
  surface,s                 ; I dislike such plots, hard to read off values
  shade_surf,s              ; idem
  show3,s                   ; yet worse
  xsurface,s                ; primitive tool to change viewing point etc
  isurface,s                ; not for me
  cgsurface,s               ; Coyote alternative, much better
                            ;   grab and change viewpoint with left mouse
                            ;   zoom in and out with right/middle mouse
                            ;   various clicker options
  cgsurface,s,/shaded       ; idem
  contour,s
  contour,s,nlevels=50
  contour,s,nlevels=20,/downhill
  cgcontour,s,nlevels=20,/window  ; Coyote alternative in resizable window
  cghistoplot,s,nbins=50,/window  ; histogram = occurrence distribution
  hist=histogram(s,nbins=50,omin=omin,omax=omax) ; the same clumsily a la IDL
  binsize=(omax-omin)/49.
  normhist=hist/float(max(hist))
  xhist=omin+indgen(50)*binsize
  plot,xhist,normhist,psym=10

image display
-------------
  ssize=SIZE(s)              ; get array type and size
  nx=5*ssize[1]              ; ssize[0] = number dimensions
  ny=5*ssize[2]              ; etcetera for more dimensions
  s5=rebin(s,nx,ny)          ; resample s for larger display
  tvscl,congrid(s,188,188,/interp) ; arbitrary resizing (slow)
  wdelete
  window,xsize=nx,ysize=ny   ; window equal to image size
  tv,s5                      ; oops, tv expects value range 0-255
  print,min(s5),max(s5)      ; show extrema
  tv,s5<0                    ; same selection, tv wraps negative values
  tv,(s5-min(s5))/(max(s5)-min(s5))*255     ; rescale to range (0-255)
  tvscl,s5                   ; same
  s5b=bytscl(s5)             ; make bytscale image (8 bits = shades 0 - 255)
```

```
  tv,s5b                      ; same as tvscl,s5
  s5pos=fltarr(nx,ny)         ; declare same-size array set to zero
  s5pos=0.*s5                 ; the same if you don't have nx, ny
  indpos=where(s5 gt 0)       ; 1D index vector counting along rows
  s5pos[indpos]=s5[indpos]    ; equate to s5 for only these indices
  tvscl,s5pos                 ; shows s5 where s5>0, 0 elsewhere
  tvscl,s5>0                  ; the same but quicker
  tvscl,s5 gt 0               ; I hope you expected that. Honestly?
  tvscl,s5<(-1)               ; parentheses needed
  tvscl,s5>(-1)<1             ; clip cutoffs at -1 and +1
  tv,bytscl(s5,min=-1,max=1) ; idem
  indcut=where(s5 gt -1 and s5 lt 1)  ; try the same this way
  s5cut=fltarr(nx,ny)                 ; where gives 1D vector, need array
  s5cut[indcut]=s5[indcut]            ; s5cut equals s5 where > -1 and < 1
  tvscl,s5cut                         ; why different from tvscl,s5>(-1)<1?
  profiles,s5cut              ; slice image, left mouse toggles rows, columns
                              ; stop with right mouse (with cursor on image)
  loadct                      ; set colour table; choose e.g. 4
  tv,s5b                      ; hideous; real scientists prefer monochrome
  xpalette                    ; tool to adjust color table
  xloadct                     ; idem (I like this one better)
  tvscl,s5b>127               ; display brighter half (not the same as s5>0)
  erase
  tvscl,s5[0:nx/2-1,0:ny/2-1]  ; bottom-left quarter bytescaled on its own
  wdelete                     ; kill window (I use my wdelall.pro)
  tvbox,size,x,y,color        ; SSW box overplot (color=0 black, 255 white)

PostScript image following Alfred de Wijn
-----------------------------------------
  nx=5       ; define s again but let's now have large pixels
  ny=5       ; square image
  xaxisarr=indgen(nx)*float(nx)/(nx-1) ; add 1 for pixelated image
  yaxisarr=indgen(ny)*float(ny)/(ny-1) ; add 1 for pixelated image
  xaxisarr=(indgen(nx)*float(nx)/(nx-1)-CRPIX1)*CDELT1+XCEN    ; solar X axis
  xaxisarr=(indgen(nx)*float(nx)/(nx-1)-(nx+1)/2.)*CDELT1+XCEN ; solar X axis
  axrat=yaxisarr[ny-1]/xaxisarr[nx-1]
  k=indgen(nx) & f=sin(k/5.)/exp(k/50.) & g=cos(k/5.)*exp(k/50.) & s=f#g
  set_plot,'ps'                                    ; postscript output
  !p.font=1                        tv                ; true type fonts
  !p.thick=2 & !x.thick=2 & !y.thick=2 & !p.charthick=2  ; I like thick
  filename='demo3.eps'
  device,filename=filename,xsize=10,ysize=10*axrat,bits_per_pixel=8,$
    /encapsulated,/tt_font,set_font='Times',font_size=12,/portrait
  tv,bytscl(s),0.15,0.15,xsize=0.8,ysize=0.8,/normal       ; bytescaled data
  contour,s,xaxisarr,yaxisarr,/nodata,/noerase,/xstyle,/ystyle,$  ; add axes
    position=[0.15,0.15,0.95,0.95],xticklen=-0.02,yticklen=-0.02*axrat,$
    xtitle='x  [px]',ytitle='y  [px]'
    ; The tv and contour position and size values must correspond
```

```
    ; (here square image as 8 cm square with borders 1.5 and 0.5 cm);
    ; the wider bottom and left margins (1.5 cm) serve for axis labels.
    ; Bware: position x and y ranges must be equal for square pixels
    ; The negative tick lengths produce outward ticks.
    ; Redefine the indgen arrays for axis scaling
  device,/close          ; write ps file
  set_plot,'x'           ; back to output on Unix/linux/MacOS Xwindow screen
  ; set_plot,'win'       ; back to output on Micro$oft Windows screen
  !p.font=-1             ; back to default IDL (Hershey) fonts
  !p.thick=0 & !x.thick=0 & !y.thick=0 & !p.charthick=0   ; reset
  spawn,'cat '+filename+$                          ; replace irritating
    '| sed "s|Graphics produced by IDL|'+filename+$  ; IDL plot banner
    '|" >  idltemp.ps; mv idltemp.ps '+filename      ; with the file name
  spawn,'gv '+filename             ; set gv to "watch file" for rewrites
    ; NB: Mac users see smoothed pixels in Preview; first use epstopdf

PostScript image with Coyote ps_start and ps_end
------------------------------------------------
  xsize=8.8 & ysize=xsize*2/(1+sqrt(5))
  ps_start,filename='ctdemo3.eps',font=1,tt_font='Times',$
    /nomatch,xsize=xsize,ysize=ysize,/metric,/encapsulated,charsize=0.5
  !p.thick=3 & !x.thick=3 & !y.thick=3 & !p.charthick=3  ; cg default=2
  cgimage,bytscl(s),/keep_aspect,position=[0.15,0.15,0.95,0.95],$
    /axes,axkeywords={font:1,ticklen:-0.02,xtitle:'x  [px]',ytitle:'y  [px]'}
  ps_end    ; this also resets the ! thicknesses back to what they were
  spawn,'gv ctdemo3.eps'
    ; Other axis scales: define axkeywords xrange and yrange

PostScript image from a Coyote cg screen window
-----------------------------------------------
  cgimage,bytscl(s),/interpolate,/keep_aspect,charsize=2,$
    /window,position=[0.15,0.15,0.95,0.95],$
    /axes,axkeywords={font:1,ticklen:-0.02,xtitle:'x  [px]',ytitle:'y  [px]'}
  ; get ps by clicking on 'save window > as ps file' under 'file', or use
  cgcontrol,create_ps='cgdemo3.eps',/ps_encapsulated,/ps_metric
  spawn,'gv cgdemo3.eps'
   ; NB: the cgimage screen image is smoothed by /interpolate,
   ;     whereas the ps output remains pixelated.  Use rebin (as above for
   ;     s5) to smooth the latter too.  I might do that for a math
   ;     function but I wouldn't for actual data.
   ; NB: similarly, the addition of an endpoint to the axes befits
   ;     a pixelated image but not a math function.


INPUT/OUPUT
===========

read/write formatted files
```

```
--------------------------
  openw,1,'myfile.ext'  ; open file myfile.ext on "logical unit" 1 for writing
  printf,1,s            ; write free-format file
  close,1               ; free "lun" 1
  openr,1,'myfile.ext'  ; now open that file for reading as unit 1
  ss=fltarr(100,100)    ; define variable type and size
  readf,1,ss            ; read free-format file from unit 1 into array ss
  help,/files           ; show which files are open as "unit"
  close,/all            ; free all units, closing the files

read/write binary files
-----------------------
  writeu,readu               ; unformatted binary read/write, faster
  openr,1,/xdr,'myfile.ext'  ; portable binary format, hardware independent

random access into a file through assoc
---------------------------------------
    ; to sample files that exceed the available memory
    ; very useful for terabyte-challenged laptop owners!
  get_lun, unit                     ; the official way to open a file
  openr,unit,'big-3D-data_cube'     ; file is intarr(nx,ny,nt)
  p = assoc(unit, intarr(nx,ny))    ; define image addressing
  image=p[1000]                     ; this gets image[*,*,1000]
  free_lun,unit                     ; closes the file too

FITS files (much used in astronomy; run ssw)
--------------------------------------------
  writefits,'filename.fits',array [,header]      ; adds header if you don't
  array=readfits('filename.fits' [,header])      ; no lun specification needed
  mreadfits,file(list),index,data,[....]         ; ssw, fits with extensions
  mwritefits,index,data,[outfile=outfile,..]     ; ssw, fits with extensions
  mwrfits,something,filename,/create             ; multi-purpose fits write
  something=mrdfits(filename)                    ; multi-purpose fits read
  header=headfits('filename.fits')               ; read header only
  nx=fxpar(header,'naxis1')                      ; when header = string array
  sxaddpar,outheader,'naxis1',nx_new,'new NX'    ; (re)set string parameter
  nx=header.'naxis1'                             ; when header = structure
  openr,1,'filename.fits',/swap_if_little_endian ; fits files are big_endian
  p = assoc(1,intarr(nx,ny),2880)                ; N x 2880 = skip fits header
  data_swap=swap_endian(data)  ; swap endian of variable, array, structure
  mkhdr,header_out,outtype,[nx,ny,nt]            ; make simple file header
  modfits,file,data,header                       ; replace data or header
  filelist=file_search(path+filenamepart)        ; string with * wild
  fileonly=file_basename(file)                   ; remove path in file string
  filename=repstr(fileonly,'.fits','')           ; filename without extension

saving IDL command sequences
----------------------------
```

```
  journal,'filename'       ; copies all typed commands to a journal file
  save,filename='name.sav' ; saves a full session (not in Student Edition)
  save,filename='name.sav',var1,var2,...     ; save only selected variables
  restore,'name.sav'       ; restart that session (you or your colleague)

read ASCII tables
-----------------
   using as example file falc.dat (solar atmosphere model) at:
   https://robrutten.nl/rrweb/rjr-edu/exercises/ssb/falc.dat


  with readcol.pro (Google for it; in SSW/idlastro astrolib library)
    readcol,'falc.dat',h,tau5,colm,temp,vturb,nhyd,nprot,nel,ptot,$
      pgas_ptot,dens,skipline=4
    NB: add eg: ,format='I,I,A,F' for initial integer + string columns

  primitive, as above:
    openr,1,'falc.dat'
    falc=fltarr(11,80)   ; 11 columns, 80 lines, no string entries
    dummy=''
    for iskip=1,4 do readf,1,dummy  ; skip 4-line header
    readf,1,falc
    h=reform(falc[0,*])
    tau5=reform(falc[1,*])
      etcetera

  as a structure, with read_struct.pro (Google for it; in sdssidl library):
    falcfile='falc.dat'
    falcstruct={height:0.0,tau5:0.0,mass:0.0,temp:0.0,v_mic:0.0,$
      n_h:0.0,n_p:0.0,n_e:0.0,p_tot:0.0,p_ratio:0.0,dens:0.0}
    read_struct,falcfile,falcstruct,falc,nlines=84,skiplines=4
    help,/structure,falc
    plot,falc.height,falc.temp<10000,/ynozero
    print,falc[0].height  ; print the first value (top of FALC)
    h=falc.height         ; select variable
    NB: read_struct.pro does not work for columns with irregular strings

  as a structure with IDL's own read_ascii.pro and ascii_template.pro:
    falctemplate=ascii_template('falc.dat')        ; opens GUI, work through
    save,falctemplate,filename='falctemplate.sav'  ; save for next time
    restore,'falctemplate.sav'                     ; use next time
    table=read_ascii('falc.dat',data_start=1,num_records=80,$
      missing_value=0,template=falctemplate)       ; read into structure
    help,table,/struct
    h=table.field01[*]                             ; get first column

write ASCII tables
------------------
    writecol,'filename.dat',vect1,vect2,vect3,fmt='(3f15.3)'
```

```
   ; in my misc.lib or google for the pro; up to 14 (19) vectors
        ; alternative: SSW forprint.pro
```

PROGRAM STRUCTURE

```
  Start a new file filename.pro; edit it (Windows: IDL desktop; Unix:
  external editor or idlde.  Emacs with IDLWAVE gives great pro layout
  and offers many shortcuts (Google idlwave).

  In linux the file name must be lowercase.  Its structure:

  pro procedurename,param1,param2,...,keyword1=keyword1,....
    ;+
    ; standard header with information
    ;-
    IDL statements
    IDL statements     ; all local parameters are only known within this pro
  end

  function functionname,param1,param2,...,keyword1=keyword1,....
    ;+
    ; standard header with information
    ;-
    IDL statements
    IDL statements
    something=...                ; value to the function
    return,something             ; output of the function
  end

  ; -------- start of main-level program (if any) ----------

  ;; pro routinename,param1,param2,..,keyword1=keyword1,.. ; in when perfect
    IDL statement
    IDL statement

    procedurename,a,b,keyword=c
    x=functionname(a,b,keyword=c)

    stop     ; for intermediate command-line inspections, continue with .con

    IDL statement
    IDL statement
  end

  The last "main-level part" is a sequence of IDL statements after the
  last procedure or function that does not start with PRO or FUNCTION.
  It must end with END.  You compile this program with ".com filename"
```

```
  and run it with ".r filename" or ".rnew filename" which cleans out
  earlier variables and recompiles too.  The latter recompiles the
  subroutines within the file also.

  After the program completion all main-level variables remain
  available for inspection and tests on the command line.  Use this
  main level for trying out and adding new things.  Insert temporary
  stops to check on local variables or diagnose an error.  When your
  development is done, then convert the program into a procedure or
  function by inserting its name as "pro routinename" or "function
  routinename" above the start of the statements, as illustrated above.
  This new routine may go to a separate routinename.pro file or may
  remain in the present filename.pro file.  You can add a main part
  calling it underneath for modification testing.  If you do this
  rightaway then on-the-fly testing while developing a subroutine
  is very easy when using emacs IDLWAVE.

  It is confusing that IDL procedures/functions have extension .pro
  but that IDL main programs have these also.  And perhaps your IDL
  batchfiles too.  I use .idl for the latter and instruct
  emacs to give these IDLWAVE appearance with .emacs entry: (setq
  auto-mode-alist (cons '("\\.idl\\'" . idlwave-mode) auto-mode-alist))

  It is confusing that somename() is not always interpreted by IDL as
  a function but sometimes as a variable, because in older days
  (before edition 5.0) IDL used parentheses instead of square brackets
  for array indices.  You can ascertain function interpretation and
  recompile with: forward_function somename (proname without quotes).


Using procedures and functions
------------------------------
  IDL> .run programname            ; compilation (only main program is run)
  IDL> .r programname              ; idem; IDL accepts unique abbreviations
  IDL> .rnew programname           ; first discard all existing variables
  IDL> .r -t programnname          ; show content in manpage format
  IDL> .com procedurename.pro      ; compile a procedure or function
  IDL> procedurename,param1,...    ; run a compiled procedure
  IDL> a=functionname(param1,...)  ; evaluate a compiled function
  IDL> reset_session               ; wipe everything, also commons, & restart

  IDLWAVE: remain in the emacs window with your program and use its tons
  of fast keybindings including (with C = CONTROL):
    C-c C-d C-c   ; compile and run program (set auto separate shell opening)
    C-c C-d C-p   ; print value of variable under cursor in 2nd window
    SHIFT-mouse2  ; idem
    C-c ?         ; show help for procedure or keyword under cursor
    C ALT q       ; re-indent the routine the cursor is in
```

```
  C-c C-d C-x   ; jummp to next syntax error

function example (in a separate file addup.pro):
  function addup,arr
    ;+
    ; sums 1D array ARR (but IDL's total is faster and more general)
    ;-
    arraysize=SIZE(arr)
    if (arraysize[0] ne 1) then print,'addup input is not a 1D array'
    sumarr=0
    for i=0,arraysize[1]-1 do sumarr=sumarr+arr[i]
    return,sumarr
  end
  IDL> .com addup               ; recompile after every program change
  IDL> try=findgen(100)         ; try = floats 0.,.....,99.
  IDL> print,addup(try)
  IDL> print,total(try)         ; check with IDL array summation

"Disappearing variables": after an error in a procedure or function
your session stops within that procedure/function.  HELP displays
the local variables valid there.  That serves to check out these,
e.g. by printing or plotting or manipulating them.  RETURN gets you
back one level higher.  RETALL gets you back to the top level where
the variables of your main program or session reside.  Recompiling a
routine (.com procedurename) also returns to the top. IDLWAVE offers
slick checkpoint jumping.

If you restart after a stop in a subordinate routine you are likely
to get error messages as:
  "Attempt to subscript XXX with <YYY (ZZZ)> is out of range"
  "Variable is undefined: XXXX
which means that you forgot to type return or retall and are still stuck
within the subroutine.

STOP in a procedure/function/main stops it right there to let you
inspect the local variables at that place in the statement sequence.
Continue with .continue (or .con).

.skip N on he command line: skip N lines and continue.  Default N=1.
.out on the command line: completes the subroutine but stops after
exiting back to the higher level.

Keyword inheritance: if your program uses e.g. plot, you don't have
to supply all the plot keywords as parameters.  Add a keyword
_extra=plotkeywords to your routine definition and use the same in
its call of plot.  Now you can add any plot keyword to the call of
your program.  See ? inheritance.  Unfortunately, you can specify
only one such inheritance per routine call, but you may have layered
```

```
  inheritances (one routine calling another, each with its own
  _extra=whatever).

conditional statements
----------------------
   if (i gt 16) then begin   ; such sequences can also be run interactively
     IDL statement           ; on the command line by first typing
     IDL statement           ; IDL> .run
   endif else begin          ; then enter the sequence, and conclude with
     IDL statement           ; IDL> end
     IDL statement
   endelse

  if (y eq 3) then x=2 else x=1  ; relational operators: EQ NE LE LT GE GT

  for j=0,9 do number[j]=sin(region[j]*!pi)    ; ! gets system variable

  for j=0,20,2 do begin                    ; third number = step 2
    number[j]=sin(region[j]*!pi)
    region[j]=0
  endfor

  while (a and (cnt ne 0)) do begin        ; logical operators: AND OR XOR
    print,'Still going at count: ',cnt
    cnt=cnt-1
  endwhile

  if (n eq 0) goto, JUMP
  IDL statement
  IDL statement
  JUMP:
  IDL statement
     ; but since good programmers never use goto, a better solution is:
  if (n neq 0) then begin
    IDL statement
    IDL statement
  endif
     ; or the use of break
  for itrans=0,ntrans-1 do begin
    IDL statements
    if (transition[itrans].i eq i and transition[itrans].j eq j) then break
  endfor

  if (keyword_set(fontsize) eq 0) then fontsize=9   ; set keyword default
    ; but keyword_set=0 when supplied keyword=0, giving non-zero default
    ; therefore better use: if (n_elements(fontsize) eq 0) then fontsize=9

loop speedup
```

```
------------
  - use implicit loops instead of explicit loops wherever possible,
    so not:
        for i=0,100 do intensity[i]=planck(wavelength,temp[i])
    but:
        intensity=planck(wavelength,temp)
    by making sure that your function (planck.pro here) can handle
    arrays (temperature here, idem for wavelength, but you cannot call both
    as unsubscripted arrays).  With my laptop the second version is
    typically 20x faster.

  - replace an asterisk as first array index on the left-hand side of an
    assignment statement by zero, so not:
      for i=0,n-1 do array[*,i]=shift(array[*,i],delta[i])
    but instead:
      for i=0,n-1 do array[0,i]=shift(array[*,i],delta[i])
    which looks like an IDL mistake but actually speeds it up, in my
    case typically 3x.  See
   http://www.idlcoyote.com/code_tips/asterisk.html

passing parameters
------------------
  - main programs
      when running a sequence of programs, each with
        .r programname
      on the command line, the subsequently called programs know the
      variables of the earlier called programs. The most primitive way
      to pass parameters.

  - @batchfile.  A file with a sequence of single-line IDL
      commands can be run as @batchfilename on the command line or
      from a program (only spaces are then allowed before the @
      symbol, on a new line).  The file may not contain begin-end
      blocks unless concatenated by $ signs.  If an @file is run on
      the command line it may contain ".r programname" lines.  This
      way you can make an @script concatenating multiple main progams.
      (I give these files extension .idl instead of .pro, and instruct
      IDLWAVE via .emacs to treat these as IDL pro files.)

  - procedure/function parameters
      The parameter names in the call may of course differ from the
      corresponding parameter names in the procedure/function body.
      However, if the procedure/function changes the parameters, the
      changed versions are passed back to the calling program at the
      procedure/function completion.  If values are entered in the
      call they do not change.  See IDL help ? passing parameters.

  - commons
```

```
      The traditional FORTRAN manner of passing blocks of parameters.
      Example: common fourier,nx,ny,nt,cad
      Put it in all pro's that need the parameters, and in the main
      part if need be.  Initiate the parameter values in the main
      part, or in the first pro that is called.  The traditional
      problem is that the same parameter name may already be used in
      another program (by another programmer).  Also, common blocks
      cannot be shared between multiple IDL instances.

  - structures
      The newer way.  Much used in SolarSoft data reduction software.
      They collect big parameters sets under a single name or anonymously
      to be passed as parameter.  Google "IDL structures".
      Example:
a=1.5
b='Never a dull moment with Kees D'
c=1
d=[4.,5.,7.]
s={a:a,b:b,c:c,d:d}   ; definition without name: anonymous structure
print, s.a
print, s.b+'. from whom I took this example'

- pointers
      serve for variables that persist outside a routine, for
      example pointing at a given location (address) within a
      structure.  See:
   http://www.idlcoyote.com/misc_tips/pointers.html

   http://www.idlcoyote.com/misc_tips/precedence.html

      c32=(*hatom.Cij_ptr)[*,2,1]   ; select a vector using a pointer

widgets
-------
  Interactive gui's to use mouse actions.  Not treated here but nice examples
  (from Oslo) are shown in my movex.pro.

programming hints
-----------------
  - never ever forget that IDL array indices start at 0 ("fingers 0-9"
  - never forget that you may need to type "retall" at some error
  - try, experiment, check on the command line, than insert into program
  - split programs in separate procedures and functions, test separately
  - use parameters instead of numbers to get dynamical adaptivity
  - use size(array) to get unknown array dimensions in procedures
  - choose clear variable names (in English please)
  - add lots of explanatory comments (in English please)
  - add detailed explanation at procedure/subroutine start between
      ;+ and ;- lines for doc_library (as astronlib and SolarSoft do;
```

```
    Emacs IDLWAVE inserts a template at C-c C-h)
- answer a procedure call without parameters or a function() call with:
    if (n_params() lt N) then begin    ; N = nr required parameters
      print,'procedurename, yyy, zzz'
      print,'   yyy = ...'
      return       ; return,-1 for a function called as x=function()
    endif
- indent begin ... end structures (two spaces is my habit)
- journal,'filename' records all your command-line entries, useful for
    subsequent conversion of the successful trials into programs
- use "save" to copy your work to a colleague
```