The Underground Guide to IDL

A collection of IDL ideas and hints that will
save you time,
and
make your colleagues jealous.

Version 1.0
7 August 1996

Liam E. Gumley
liam.gumley@ssec.wisc.edu

Before you get started, you will need:

(1) IDL 4.0.1. Get it from

```
ftp://ftp.rsinc.com/pub/idl
```

Most of the examples shown in this document will work in IDL demonstration mode.

(2) The ESRG IDL user library. It may be already installed - ask your system guru. If not, then get it from

```
ftp://crseo.ucsb.edu/pub/idl/esrg_idl_3.3.tar
```

Make sure that the ESRG library directory is in your IDL path. Let's say you installed the library in `$HOME/esrg`. The path syntax would be as follows:
For Unix ksh users, the path syntax would be something like

```
$ IDL_PATH=$IDL_PATH:$HOME/esrg; export IDL_PATH
```

For Unix csh users, the path syntax would be something like

```
% setenv IDL_PATH=$IDL_PATH:$HOME/esrg
```

(3) Liam's IDL user library. The components you will need are available at

```
http://cimss.ssec.wisc.edu/~gumley/index.html
```

under "IDL Local Resources". Get the Frame Tools, Image Mapping Tool, and HDF SDS Tool. Stick the files in a handy directory, and set up your IDL path as shown in (2) to include the directory.

(4) Some knowledge of IDL. I'm not going to hold your hand - well maybe I will, but I'll be dragging you pretty fast. In particular, make sure you can use the on-line help. For built in routines, just type

```
IDL> ?
```

and the interactive help viewer will appear. If you need help on a user library procedure (`tvim.pro` for example), type

```
IDL> doc_library, 'tvim'
```

and the help text embedded within `tvim.pro` will be displayed. Incidentally, if you get an error when you type the last command, then the ESRG library was not set up correctly in your IDL path. If this is the case, go back to step (2).

Writing IDL programs (skip this bit if you already know how)

IDL has a very nice interactive command line interface. It's so nice, that you might be tempted to think that's all there is to IDL. The truth is that IDL is a very handy and powerful programming environment (remember, IDL stands for Interactive Data Language). So if you haven't done so already, it's time you started writing some IDL programs.

The basic IDL program unit is the procedure. For example, create the following as an ASCII text file named `test.pro` using your favorite editor:

```
pro test
print, 'Hello world'
end
```

Then start IDL, and type the command

```
IDL> test
```

You should see the output

```
Hello world
```

Congratulations! You just wrote an IDL procedure. When you typed `test` at the command line, IDL searched the current directory for a file named `test.pro`. When found, it compiled and ran the procedure. If it had not been found, IDL would have started searching directories in the IDL path. Now let's say you want to modify and re-compile the procedure. You can invoke your editor from within IDL by using the `$` prefix on the command line, which tells IDL to execute the command outside of IDL (i.e. in a Unix shell). So to start vi you could type

```
IDL> $vi
```

So edit `test.pro` to print the string `'Just a test'`. To re-compile the procedure, type

```
IDL> .rnew test.pro
```

and then to run the procedure, type

```
IDL> test
```

That's the basics of writing IDL procedures. If you want more information, then take a look at the training manual "Learning IDL" (I have an old copy).

Displaying 2D images (the wonders of TVIM)

If you've used IDL for a while, you're probably familiar with the TV and TVSCL commands used to display 2D images. The ESRG user library contains a very nice enhanced image display procedure named TVIM. Personally I hardly ever use TV or TVSCL any more because TVIM is so useful. Let's take a quick look at what it can do:

```
IDL> tvim, dist(32), /scale, title='An Image', range=[10,20]
```

Isn't it cool? And what's more, since the image is created in a plot window, you can overlay points or lines very easily, as shown here

```
IDL> oplot, indgen(10), psym=1
```

You can't do that in TV or TVSCL (at least, not easily). One of the best features of TVIM is that it scales the image to fit in the current display window, so you don't have to worry about whether the image is too big or small to fit nicely. Take a look at the help for TVIM by typing:

```
IDL> doc_library, 'tvim'
```

You can get help for any IDL library procedure this way (but not for the built in routines). For more extensive help, type ? at the command line.

TVIM also interfaces very nicely with other ESRG library utilities, most notably TOGGLE (switch Postscript output on or off), for example:

```
IDL> toggle, file='test.ps'
IDL> tvim, dist(256)
IDL> toggle
```

What you did was create a single-page Postscript file name test.ps containing the TVIM image. How much simpler could it be? Go ahead and print the image if you like. We'll delve more into Postscript printing later in ***The Joy of Postscript***.

One note: TVIM only likes to display 2D arrays. How do find out if your array is 2D? Use the help command like this:

```
IDL> help, dist(256)
```

If your array is not 2D, then use the REFORM command to make it 2D.

TVIM is IMHO the single most useful IDL user-library procedure. Read the help for TVIM thoroughly, use it, and reap the accolades.

The Joy of Postscript (or, why you'll never print a GIF again)

Yet another outstanding feature of the ESRG library is the command `TOGGLE`, which provides a simple means of creating Postscript output. You've already seen how simple it is to create a Postscript image in ***Displaying 2D images***. In this section, I'll show you a few more things that will spice up your plots and images.

First, a word about IDL graphics displays. For Postscript output, you generally want 256 colors (I won't talk about 24 bit displays here). However, when you start IDL, there is no guarantee that you will get 256 colors. Unless you specifically request it, IDL will grab an arbitrary number of colors the first time a graphics window is created after IDL startup. The number of colors stays fixed for the rest of the IDL session. You can however specify how many colors you want on the command line. For example, as soon as you start IDL, you can type

```
IDL> window, /free, colors=256
```

which will grab 256 colors, but may have weird effects on your display (on my SGI Unix box it works fine - your mileage may vary). Try it on your box and see. If your display turns weird (if it has, then you know what I mean), then no problem. You can still create your Postscript output, but you should do it in an IDL procedure, as shown in ***Writing IDL Programs***. Then you can test your display in the graphics window, tinker with it until you are happy, and then exit IDL, restart IDL, and type

```
IDL> window, /free, /pixmap, colors=256
```

which creates a graphics window in memory that is not displayed. Then just run your procedure to create the Postscript output.

Now, back to the `TOGGLE` command. The way you use it is first you `TOGGLE` on, then display your image or plot, and then `TOGGLE` off. The following example shows some of the most useful keywords:

```
IDL> toggle, /landscape, /color, file='color.ps'
IDL> loadct, 39 & !p.font=0 & device, /helvetica
IDL> tvim, dist(256), /scale, title='Color Image'
IDL> toggle & !p.font=-1
```

First, Postscript was turned on in landscape mode, with color enabled (black and white is the default. Then a rainbow color table was loaded, and the font changed to Postscript Helvetica (which looks very similar to Arial in MSWord). Then an image was displayed. Finally, the Postscript file was closed and the default font restored. You can use ghostview or gs to verify that the image was created properly. To create multi-page Postscript output, use the `ERASE` command at the end of each page. Note that `TVIM` automatically scales the image to Postscript resolution (cool!). No more printing GIFs!.

Using Graphics Frames (yes, just like McIDAS)

One day, someone said to me “Gee I wish IDL had graphics frames like McIDAS”. I thought that was a hell of an idea, so I wrote set of graphics frame utilities for IDL (finally, I get to talk about something I wrote). These utilities make it easy to flip back and forth between plots or images, and you can have as many frames as memory will allow. The color table for each frame is saved, as are the window parameters. You can create a plot in one frame, and then come back later and overplot in data coordinates. There is also a simple way to create movie loops from a sequence of images or plots. Have I got your attention? Here come the details.

First, you need to create a set of graphics frames. Just type

```
IDL> FSET
```

and a set of four graphics frames, sized at 640x480 pixels, will be created. The display window will appear, and you are ready to start creating images or plots as you would with a graphics window created by the `WINDOW` command. For example,

```
IDL> tvim, dist(8)
IDL> af
IDL> tvim, dist(64)
IDL> bf
```

As you can see, the `AF` command advances one frame, and the `BF` command backs up one frame. Check the on-line help for more `FSET` options, such as specifying the size and number of graphics frames. Only one set of `FSET` frames can be active at any time. For example, if you type

```
IDL> fset, /landscape, frames=2
```

the existing set of frames will be destroyed, and a new set created.

Frame looping is done with the `LF` (loop frames) command, as shown here:

```
IDL> fset
IDL> for i=2,5 do begin & tvim, dist(2^i) & af & endfor
IDL> lf
```

Options for `LF` include setting the delay interval between frames, the direction of looping, and the first and last frame numbers. The last frame command is `SF` (show frame) command, which shows a specified frame, or with no arguments, prints the current frame number. Use the `/HELP` keyword, or the `DOC_LIBRARY` command to get help for `FSET, AF, BF, LF`, or `SF`.

Reading HDF SDS data (mainly for MODIS and MAS fans)

IDL has HDF3.3r4 support built in, so reading HDF SDS datasets is pretty easy. Don't ask me about HDF VSETs, VDATAs, RIGs or any other kind of HDF thingy, because I only know about SDSs.

Let's say you have a HDF file called '`ancillary.hdf`' which contains an SDS named '`total_ozone`'. To read the whole SDS, the commands are as follows:

```
IDL> sd_id = hdf_sd_start( 'ancillary.hdf' )
IDL> index = hdf_sd_nametoindex( sd_id, 'total_ozone' )
IDL> sds_id = hdf_sd_select( sd_id, index )
IDL> hdf_sd_getdata, sds_id, data
IDL> hdf_sd_endaccess, sds_id
IDL> hdf_sd_end, sd_id
```

The commands look complicated, but all that is happening is
(1) Open the file in SDS mode,
(2) Get the index number for the desired SDS name,
(3) Select the desired SDS,
(4) Read the entire SDS into the variable `data`,
(5) End access to the SDS,
(6) Close the file.

The most likely option you will want to use is to tell `HDF_SD_GETDATA` to only read part of the SDS. You can do this by using the `START` and `COUNT` keywords. See the on-line help for `HDF_SD_GETDATA` for more information.

For a simpler approach, I wrote a procedure named `SDS_READ`, which provides a point-and-click interface for reading a SDS. The command can be used as shown:

```
IDL> sds_read, pickfile(), data
```

The optional keywords `NAME, UNITS, SCALE`, and `OFFSET` can be used with `SDS_READ` to return the SDS long name, units string, scale value, and offset value (if they exist - default values are returned otherwise). Thus, to create and image of the ozone data set described above (assuming it's a 2D array), you could do

```
IDL> sds_read, 'ancillary.hdf', ozone, $
IDL> name = name, units = units, scale = scale, offset = offset
IDL> tvim, ozone*scale+offset, /scale, title=name, stitle=units
```

Note that IDL 4.0.1 has HDF3.3r4 support only - odd things may happen if you try to read an SDS dataset created with HDF4.0.

Displaying data on map projections (the black art)

So you've got a 2D array, and you know the latitude and longitude for each element in the array. It could be on a regular grid (e.g. model output), or an irregular grid (an AVHRR image). You want to display the image on a map projection with coastline overlaid. For datasets defined on a regular grid, it's very easy. Let's assume you have a 2D array named `data`, which is defined on a global grid spanning -90S to +90N degrees latitude, and -180W to +180E degrees longitude (Greenwich = 0). Try this:

```
IDL> data = dist(256)
IDL> map_set
IDL> image = map_image( data, xx, yy, /bilinear, compress = 1 )
IDL> tv, image, xx, yy
IDL> map_continents
IDL> map_grid
```

If your data is on a regular grid that does not cover the whole globe, then limit the extent of the map projection using the `LIMIT` keyword to `MAP_SET`, which takes the form `[LATMIN,LONMIN,LATMAX,LONMAX]`. See the on-line help for `MAP_SET` and `MAP_IMAGE` to learn about topics such as other map projections in `MAP_SET`, and creating Postscript output with `MAP_IMAGE`.

It gets tricky when your data is on an irregular grid. There are two ways to go about it. The first method works well on small datasets (a few thousand points or less). You first need to resample the data to a regular grid. Here's an example:

```
IDL> seed = 1L; x = randomu(seed,1000); y = randomu(seed,1000)
IDL> z = exp(-3*((x-0.5)^2)+(y-0.5)^2))
IDL> triangulate, x, y, tri
IDL> data = trigrid( x, y, z, tri )
```

At this point, `DATA` is now defined on a regular grid, and you can display it using `MAP_SET` and `MAP_IMAGE` as shown above.

The second method is more useful when you have a large (say 500x500 points or larger) dataset like a satellite (AVHRR for example) image. The `TRIANGULATE` and `TRIGRID` routines just don't work on datasets this large. Let's say you have a 2D image array, and corresponding latitude and longitude arrays. Try this (example taken from the `IMAGEMAP` embedded help text):

```
IDL> ; Create latitude, longitude, and image arrays
IDL> c = complex(2,2) + cmplxgen(250,200,/center)/100
IDL> c = c + c^2
IDL> lon = float(c) - 100.0
IDL> lat = 20 + imaginary(c)
IDL> image = sqrt(abs(sin(lon*!pi)*sin(lat*!pi)))^0.3
IDL> ; Resize arrays to simulate 1 km resolution imagery.
```

```
IDL> lat = congrid(lat,1000,800,interp=1)
IDL> lon = congrid(lon,1000,800,interp=1)
IDL> image = congrid(image,1000,800,interp=1)
IDL> ; Display data on Mercator projection
IDL> w11x8
IDL> imagemap,image,lat,lon
IDL> map_continents
IDL> map_grid
```

Note that when using IMAGEMAP, you need a largish window defined first where the image can be created. I like to use the ESRG library commands W8X11 or W11X8 to simulate portrait or landscape 8.5”x11” paper. The image may look better in one compared to the other. Also note that if you set the ISOTROPIC keyword to IMAGEMAP the resulting image may look nicer, since it is displayed on an isotropic projection. This procedure really just makes an image that is good for looking at - I don’t claim that it’s a robust resampling algorithm. In fact, if you come up with a fast and robust algorithm for resampling images 1000x1000 in size or larger to a map projection, please let me know!

Also note that IMAGEMAP won’t work properly if you have Postscript output turned on, since it resamples the data to the current graphics device resolution. You are better off to create image first (use the IMAGE keyword to IMAGEMAP), and then display the image in Postscript mode.

Sections to be added

Contouring 2D data (making pictures your boss will love)

Writing an IDL GUI procedure (widgets are your friends)