6th COSPAR Capacity Building Workshop
Sinaia, 4-16 June 2007
Joachim Vogt

# Basic Analysis Techniques & Multi-Spacecraft Data — Computer Session —

**Sheet 1** 5 June 2007

## 1 Getting started with IDL

Follow the instructions to learn how to use IDL and to invoke basic features. Note that this guide is not meant to be used without further reference. If you have questions, please consult the IDL help system or ask the instructor.

### 1.1 Preparations

Log into the Linux PC. If you do not have an account of your own, go to the workshop directory and create a subdirectory where you can store your files without interfering with other workshop participants.

```
Linux> mkdir your_name
Linux> cd your_name
```

All the sample program files can be found in the subdirectory `ComputerSessions/BasicAnalysisTechniques_Vogt/ex1/` of the workshop web page directory[1].

Start IDL, first option: line mode.

```
Linux> idl
IDL> (waiting for input)
```

Note that IDL programs, functions and procedures can be conveniently edited in `xemacs IDLWAVE mode`.

```
Linux> xemacs test.pro &
```

Start IDL, second option: IDL Development Environment.

`Linux> idlde &` (a new window is opened)

Invoke the help system

```
IDL> ?
```

and look for documentation.

[1] http://www.faculty.iu-bremen.de/jvogt/cospar/cbw6/

Keeping track of your input is not absolutely necessary but can be quite useful. The following command writes it to the file `idlsave.pro`.

```
IDL> journal
```

## 1.2 1D arrays and line (2D) plots

One-dimensional arrays can be created with the family of `indgen` routines.

```
IDL> a = findgen(21)
IDL> help, a
IDL> print, a
```

Functions of arrays can be defined in a straightforward way.

```
IDL> b = sin(a)
IDL> help, b
IDL> print, b
```

You may access individual elements of the array or a subarray. `a[0]` is the first element of the array `a`.

```
IDL> print, a[0]
IDL> print, a[3:5]
IDL> c = a[where(b gt 0)]
```

Create a simple line plot.

```
IDL> plot, b
IDL> plot, a, b
```

You may want to redefine the array variables to get a better resolution. Note the differences between the various `plot` commands, and that `linestyle` is an optional *keyword*.

```
IDL> a = findgen(201)/10.
IDL> b = sin(a)
IDL> plot, b
IDL> plot, a, b
IDL> plot, a, b, linestyle=2
```

A 1D floating-point array `x` with `nx` elements which ranges from `xmin` to `xmax` can be created in the following way.

```
IDL> nx = 21
IDL> xmax = 5.
IDL> xmin = -5.
IDL> x = xmin + (xmax-xmin)*findgen(nx)/float(nx-1)
```

## 1.3 2D arrays

The easiest way to define multidimensional arrays is the `#` operator.

```
IDL> a = findgen(5)
IDL> b = (findgen(4)+1.)/2.
IDL> c = a#b
```

Here the variable `c` is a two-dimensional array of first dimension 5 and second dimension 4 whose elements are formed by multiplication of the elements of `a` and `b`.

```
IDL> help, c
IDL> print, c
```

This array can also be displayed by simple contour plots. Note the different results of the three commands.

```
IDL> contour, c
IDL> contour, c, a, b
IDL> contour, c, a, b, nlevels=20, /follow
```

If arrays are interpreted as vectors and matrices in the usual sense, matrix multiplication is achieved by means of the `##` operator rather than the `#` operator.

### 1.4 2D coordinate arrays for defining scalar and vector fields

Very often two-dimensional arrays for $x$ and $y$ coordinates are needed. The coordinate arrays should have a structure such that functions $f$ of two variables $(x, y)$ can be evaluated on a two-dimensional grid directly as $f(x, y)$ (without addressing individual elements of the array). Here is a procedure to achieve this.

1. Define two 1D arrays which give the range and the resolution of the independent variables $x$ and $y$ (coordinate axes).

   ```
   IDL> x = findgen(101)/10.-5.
   IDL> y = findgen(51)/5.
   ```

   The array `x` has 101 elements and ranges from $-5$ to 5. The array `y` has 51 elements and ranges from 0 to 10.

   More generally, you can specify number of elements and range in the following way.

   ```
   IDL> nx = 101 & xmin = -5. & xmax = 5.
   IDL> x = xmin + (xmax-xmin)*findgen(nx)/float(nx-1)
   IDL> ny = 51 & ymin = 0. & ymax = 5.
   IDL> y = ymin + (ymax-ymin)*findgen(ny)/float(ny-1)
   ```

2. Define two 1D arrays `x1` and `y1` with the same number of elements as the arrays `x` and `y`, respectively.

   ```
   IDL> x1 = fltarr(101)
   IDL> y1 = fltarr(51)
   ```

   Initialize all elements with the value 1.

   ```
   IDL> x1[*] = 1.
   IDL> y1[*] = 1.
   ```

   These two steps can be combined as follows.

   ```
   IDL> x1 = replicate(1.,101)
   IDL> y1 = replicate(1.,51)
   ```

3. Finally, construct two 2D arrays which represent $x$ and $y$ but have identical (2D) structure.

   ```
   IDL> xx = x#y1
   IDL> yy = x1#y
   ```

For convenience, all these steps are combined into the IDL function `coord2d.pro`. Copy this function into your working directory and call
`IDL> COORD2D, 101, 51, xx, yy, xmin=-5., xmax=5., ymin=0., ymax=5.`

An array `zz` which contains the values of, e.g., $f(x, y) = \sin(\sqrt{x} + y)$ on the grid given by `xx` and `yy` is now easily created through
`IDL> zz = sin( sqrt(xx) + yy)`
Note that `zz` automatically inherits the properties of `xx` and `yy`.

### 1.5 3D plots: displaying functions of two variables

Plotting an arbitrary function of two variables can be conveniently achieved in the following way.

1. Define two 2D coordinate arrays `xx` and `yy`.
   `IDL> COORD2D, 101, 51, xx, yy, xmin=-5., xmax=5., ymin=0., ymax=5.`
2. Now evaluate a function $f(x, y)$ at grid points $(x_i, y_j)$ and store the result in a third array `zz`, e.g.,
   `IDL> zz = exp(-xx^2)*sin(yy)`
3. Plot the array `zz` by means of a 3D plotting routine.
   `IDL> contour, zz, xx, yy, nlevels=20, /follow`
   `IDL> surface, zz, xx, yy`
   `IDL> show3, zz, x, y, e_contour={nlevels:20}`

### 1.6 Programs, procedures and functions

A sequence of commands can be stored into a file and then forms a program. A small sample program is `contour-example.pro` which comprises the commands given in the previous section to produce a contour plot. Execute the program by typing
`IDL> .r contour-example`.

Procedures and functions are also sequences of commands which perform a specific task. However, such routines are more flexible because they can have parameters with values specified only at execution time. See the IDL documentation for more information.

### 1.7 Creating postscript output

So far we have only plotted on the screen. To get a hardcopy we first create a postscript (ps) file which can then be sent to a printer. This implies changing the display variable. Without going into details, have a look at the commands in the files of the procedures `openps.pro` and `closeps.pro` and type
`IDL> openps, 'contour.ps', dnameold`

```
IDL> .r contour-example
IDL> closeps, dnameold
```
Check the result with a ps viewer, e.g.,
```
Linux> gv contour.ps
```

### 1.8 Converting postscript to other graphics formats

If you do not have a postscript viewer/converter on your computer, you should convert the postscript files to other graphics formats on the Linux PC. Here are two options.

- `convert` can produce a large number of different graphics formats (jpg, png, tif, gif ...). For example, you may simply type
  `Linux> convert filename.ps filename.png`
  at the Linux prompt to convert a postscript file to png. See the documentation on `convert` for details:
  `Linux> info convert`
  `Linux> man convert`

- `epstopdf` generates pdf output. Type
  `Linux> epstopdf filename.ps`
  to yield a file named `filename.pdf` .

### 1.9 Reading and writing files

There are several ways in IDL to write/read data to/from a file. The most convenient is the `save/restore` combination (see documentation).

Unfortunately, very often we cannot use this option, e.g., when we have to read data from files which have been formatted in a specific way by other programmes. In general, we must specify the format through a keyword in the `printf/readf` commands. However, in many cases the 'IDL free format' works fine, and additional keywords are not necessary.

The following sequence of commands illustrates how to write variables to a file using the free format. The keyword `get_lun` guarantees that the variable `lun` is assigned a free file unit.
```
IDL> a = findgen(5)
IDL> b = sqrt(a)
IDL> c = sqrt(b)
IDL> openw, lun, 'file1.dat', /get_lun
IDL> printf, lun, a, b, c
IDL> close, lun
```
Now you may use a text editor (e.g., `xemacs`) to check `file1.dat`, or you may type
```
IDL> $more file1.dat
```
Reading this data file is straightforward. You first have to define suitable arrays, open the file, and then read the data into the variables.
```
IDL> x = fltarr(5)
IDL> y = x
```

```
IDL> z = x
IDL> openr, lun, 'file1.dat', /get_lun
IDL> readf, lun, x, y, z
IDL> close, lun
IDL> print, x
IDL> print, y
IDL> print, z
```
Note that in this example the arrays are stored as *rows*.

Now suppose you want to read a data file where it is appropriate to assign different arrays to the *columns* (the typical situation if time series are considered). To illustrate such a case, have a look at the program files `write-file2.pro` and `read-file2.pro` and run the programs.
```
IDL> .r write-file2
```
Check the result:
```
IDL> $more file2.dat
```
Now read the file.
```
IDL> .r read-file2
IDL> print, x
IDL> print, y
IDL> print, z
```

### 1.10 Random number generator

So-called random processes are often used to model the influence of noise on measurements. They are characterized in probabilistic terms. We come back to this point later. For the moment it suffices to know that the IDL functions `RANDOMU` (uniorm distribution) and `RANDOMN` (normal=gaussian distribution) can be used to generate series of random numbers. Try
```
IDL> plot, randomu(seed,200)
```
and
```
IDL> plot, randomn(seed,200)
```
to get a feeling how a random process looks like.

A harmonic signal contaminated by gaussian noise at a signal-to-noise ratio of 10:1 can be visualized by
```
IDL> time = findgen(200)/10.
IDL> signal = sin(time) + 0.1*randomn(seed,200)
IDL> plot, time, signal
```