Introduction to IDL

4 – Files

Paulo Penteado
pp.penteado@gmail.com
http://www.ppenteado.net

Files

Nearly every complex program needs to store data in files.

May be small text files with a few parameters, or huge data files.

Despite common practice, **text files are not always the best choice to store data.**

There is no “the best format”. Each has advantages and disadvantages.

Often you have no choice – the files are given to you.

Files - text X binary

In any format, files are stored as a sequence of binary digits.

But there are **two main ways to code them**:

- **Text files**
  - Data are transformed into characters, using some formatting (even if implicit), and written with some text encoding (often, but not necessarily, ASCII).
  - **A text file is nothing more than strings.** The same string conversions must be made as when printing to the screen.
- **“Binary” files**
  - Even though every file is binary, this name is used for those where the data are not written as strings.
  - Often (not always) the data are stored identically as they would be stored in memory.

Many formats have a mix of text and binary data:

- A text ***header***, which provides information (***metadata***) about the data stored in the binary part, which follows.
- The header has, most importantly, information used to know how to read the binary data.

Files - text X binary

Text file advantages:

- *Human readable*.
- **May be** self-sufficient: all that is needed to understand the file is contained in it.
- Are the least language / software dependent: usually written in **ASCII** or **Unicode**; Every language always has some support for reading and writing text.
- There are some standard text formats (**CSV, XML**), which are supported in many different languages.

Files - text X binary

Advantages of binary files:

- **Usually** there are no conversions between memory and file data encoding. Thus storage is more efficient and there are no changes in data values between memory and files.

- Most standard format are binary. Exs: **FITS, CDF, NetCDF, HDF, HDF5, JPEG, JPEG2000, TIFF, GeoTIFF, MPEG, GIF, PNG, ISIS Cube**, etc.

- Easier to find ready made libraries (even high level ones) for querying, reading and writing.

- The best formats are **selfdocumented** (with text or binary data): The user does not need to provide any prior knowledge about how the data are stored.

- Every language / platform has itw own native binary format, which may be the easiest to use.
  - ➔ High level languages have sophisticated native format where everything works automagically (**IDL**, R, Python, Java).

- Many formats have the option of compressing the data.

File formats – proprietary x standard

Proprietary format: you make it up to suit your need.

Main advantage: The most convenient encoding might be chose.

Main disadvantage: **need to write all the code to read / write the file:**

- Usually more work than using a standard format.
- When sharing the file with others, you will have to document how to read it, and they will have to write code to read it.
- Each new environment where you want to read /write such a file will need new code to read / write in that format.

If others have already done all the work to develop good standard formats, well supported on many environments, why reinvent (program, document, test) the wheel?

Some common standard formats used in science

Text:

- **CSV: *Comma-separated values*** – well supported in a lot of environments:
  - Many programming languages.
  - Spreadsheets (Excel, Calc, Google Docs).
  - Plotting software.
  - Web applications to download / upload data.
  - Databases.
  - Even Gmail understands it, even in cell phones.

- **“Fixed column width”**: Not a well-defined standard, but commonly used for tables (2D arrays).

- **XML: *Extensible Markup Language* –** The most flexible text standard, widely used in general programming, to store anything (even Open Office files are XML).
  - Well supported (at a low level) by a lot of software.
  - XML files are often complex – a lot of work to process their data into something useful.

Some common standard formats used in science

Binary:

- **FITS: *Flexible Image Transport System*** – The most common format for data and tables in astronomy. Contains one or more arrays of simple types or structures, plus metadata.

- **NetCDF: *Network Common Data Form*** – evolved from **CDF**, selfdocumented, commonly supported. Each files stores one or more **arrays.**

- **HDF5: *Hierarchical Data Format*** – selfdocumented, common and well supported, allowing simple arrays and structures (it is a hierarchical format).

- **Image formats: JPEG, JPEG2000, TIFF, GeoTIFF, GIF, PNG**, etc. - store 2D/3D arrays of integers (in some cases, reals), with out without compression (lossy or lossless compression). **JPEG2000has** has advanced features, like multiresolution images.

- **Vector formats: PS, EPS, DXF, SHP, SVG, PDF** – varied levels of complexity to store drawings, shapes, and other types of data.

- **ISIS Cube**: similar to FITS files, but only used in remote sensing, Earth sciences and planetary sciences.

- **Native formats:** In the high level formats, all the work of organizing and rettrieving the data (even if there are many, complicated variables) is done automagically. Common in **IDL**, R, Python, Java. Each language has its own.

Native IDL files - savefiles

IDL's native format is the savefile.

The most convenient to use from IDL: it takes care of saving / reading several variables, however complicated they may be.

Disadvantage: outside of IDL, there is almost no support for savefiles (there is a library in Python).

Native IDL files - savefiles

```
IDL> x=dindgen(10)
IDL> y=randomu(seed,10)
IDL> a={filename:'some_file.cdf',date:'2014-08-23',$
time:'09:35:23',flux:dblarr(200),wavelengths:dblarr(200)}
IDL> b=replicate(a,27)
IDL> help
A                STRUCT    = -> <Anonymous> Array[1]
B                STRUCT    = -> <Anonymous> Array[27]
SEED             ULONG     = Array[628]
X                DOUBLE    = Array[10]
Y                FLOAT     = Array[10]
IDL> save,file='example_savefile.sav',a,b,x,y
IDL> .full
IDL> help
IDL> restore,'example_savefile.sav'
IDL> help
% At $MAIN$
A                STRUCT    = -> <Anonymous> Array[1]
B                STRUCT    = -> <Anonymous> Array[27]
X                DOUBLE    = Array[10]
Y                FLOAT     = Array[10]
```

Native IDL files - savefiles

IDL's native format is the savefile.

The most convenient to use from IDL: it takes care of saving / reading several variables, however complicated they may be.

Disadvantage: outside of IDL, there is almost no support for savefiles (there is a library in Python).

There are more complicated ways to use savefiles, to read only a few variables (not the whole file), or change the name of the variables.

- In standard library: IDL_Savefile class
- In Craig Markwadrt's library: cmsave/cmrestore (http://www.physics.wisc.edu/~craigm/idl/)

Text files - newlines

Despite the appearance, **text files do not have, intrinsically, multiple lines.**

- Text files are just a (1D) sequence of characters, written with some encoding.
- Lines separation (which allows interpreting the file as 2D, with lines and columns) is specified by conventions.
- **There are multiple conventions to specify line ends.**

The most primitive convention: **No separation:** all lines are written iwth a fixed number of characters, and it is up to the reader to understand that a new line happens every N characters.

- The least portable e most inconvenient convention. (It takes previous knowledge, or some guessing, to figure out the line width.
- Example (from an actual **FITS** header):

```
SIMPLE  =                    T / Fits standard                                  BITPIX  =                   16 / Bits per pixel                                 NAXIS   =                    0 / Number of axes                                 EXTEND  =                    T / File may contain extensions
```

With the lines wrapped at the right place, it would look like:

```
SIMPLE  =                    T / Fits standard
BITPIX  =                   16 / Bits per pixel
NAXIS   =                    0 / Number of axes
EXTEND  =                    T / File may contain extensions
```

Text files - newlines

The most common way to specify line termination is with a specific marker – the **newline code**.

It is a special character(s), not normally present in the text, which means a line end.

There are multiple newline standards. The most common:

- **LF (*Line Feed*, ASCII 10)** – Unix, Linux, Mac OS X.
- **CR - (*Carriage Return*, ASCII 13)** - Mac OS <10.
- **CR+LF (*CR* followed by *LF*)** - Windows, DOS.

This is why Windows' Notepad* does not understand newlines from Linux-style files. The file only has **LF**, while the editor expects **CR+LF**.

Some systems use the literal **`\n`** in code to specify a newline, which gets coded according to the standard in use.

Usually, file reading/writing libraries use the newline native to the system in use. (This is the case with IDL).

Software to convert among these 2 formats is common (ex: ***dos2unix***).

*Windows' Wordpad does not have this limitation.

Text files - CSV

Comma-separated values

The most best **text** standard for tables (2D arrays). Often, though not necessarily, written in ASCII.

Well-supported by languages' standard libraries and other software (Excel, Calc, Origin, Gmail, Google Docs, web applications, etc.).

Contains

- **Header lines** (optional): Lines with any text, describing the file,, plus one line for the table header (the names of the columns).
- **One (only one) table**, where all lines have the same number of rows:
  - **Columns separated by commas** (usually; sometimes, it may be another character).
  - **Lines terminated by *newline***.
  - **Strings may be delimited by “ “** (so that strings may contain commas).

Header

Ex:

```
NAME,CALMPOS,FILNAME,ECHLPOS,DISPPOS,TARGNAME,POSDIR,CLASS,MJD-OBS,ITIME,COADDS
"dec18s0001",0,"NIRSPEC-5-AO",62.6300,36.4500,"HD85258","NIRSPEC-5-AO/p1","STAR",54087.6,100.000,1
"dec18s0002",0,"NIRSPEC-5-AO",62.6300,36.4500,"HD85258","NIRSPEC-5-AO/p1","STAR",54087.6,100.000,1
"dec18s0012",0,"NIRSPEC-5-AO",63.5800,36.4500,"itan140","NIRSPEC-5-AO/p2          T","ITAN",54087.6,300.000,1
"dec18s0014",0,"NIRSPEC-5-AO",62.6300,36.4500,"itan140","NIRSPEC-5-AO/p1          T","ITAN",54087.6,300.000,1
"dec18s0015",1,"NIRSPEC-5-AO",62.6300,36.4500,"itan140","NIRSPEC-5-AO/p1","ARC",54087.6,4.00000,1
"dec18s0016",0,"NIRSPEC-5-AO",62.6300,36.4500,"HD85258","NIRSPEC-5-AO/p1","STAR",54087.6,100.000,1
```

The columns do not need to have the same type, or even be written with the same width.

Text files - CSV

There is ample support to read and write them, **with no need to do it yourself at a low level.**

May be directly read into Excel, Calc, Origin, Google Docs, Databases, web applications, read easily read with standard (or common) libraries. Ex:

```
IDL> c=read_csv('filesearch_scam.csv',header=h)
IDL> print,h
NAME CALMPOS FILNAME ECHLPOS DISPPOS TARGNAME POSDIR CLASS MJD-OBS ITIME COADDS

IDL> help,c
** Structure <c8bca508>, 11 tags, length=5256, data length=5248, refs=1:
   FIELD01          STRING    Array[41]
   FIELD02          LONG      Array[41]
   FIELD03          STRING    Array[41]
   FIELD04          DOUBLE    Array[41]
   FIELD05          DOUBLE    Array[41]
   FIELD06          STRING    Array[41]
   FIELD07          STRING    Array[41]
   FIELD08          STRING    Array[41]
   FIELD09          STRING    Array[41]
   FIELD10          DOUBLE    Array[41]
   FIELD11          LONG      Array[41]

IDL> print,c.field01[0:3]
dec18s0001 dec18s0002 dec18s0003 dec18s0004

IDL> print,c.field03[0:3]
NIRSPEC-5-AO NIRSPEC-5-AO NIRSPEC-5-AO NIRSPEC-5-AO
```

Text files - CSV

There is ample support to read and write them, **with no need to do it yourself at a low level.**

May be directly read into Excel, Calc, Origin, Google Docs, Databases, web applications, read easily read with standard (or common) libraries. Ex:

But I want my columns to have the name specified in the file! Not things like field01, field02,

One solution (from pp_lib, http://ppenteado.net/idl):

```
IDL> c=pp_structtransp(read_csv_pp('filesearch_scam.csv'))
IDL> help,c
C               STRUCT    = -> <Anonymous> Array[41]
IDL> help,c[0]
** Structure <78256968>, 11 tags, length=136, data length=128,
refs=2:
   NAME            STRING    'dec18s0001'
   CALMPOS         LONG                 0
   FILNAME         STRING    'NIRSPEC-5-AO'
   ECHLPOS         DOUBLE           62.630000
   DISPPOS         DOUBLE           36.450000
   TARGNAME        STRING    'HD85258'
   POSDIR          STRING    'NIRSPEC-5-AO/p1'
   CLASS           STRING    'STAR'
   MJD_OBS         STRING    '54087.6'
   ITIME           DOUBLE           100.00000
   COADDS          LONG                 1
```

Text files - CSV

Creating a CSV from arrays:

3 1D arrays (12 elements each), one for each column.:

```
IDL> x=dindgen(3)
IDL> y=dindgen(4)
IDL> xx=reform(rebin(x,3,4),12)
IDL> yy=reform(rebin(reform(y,1,4),3,4),12)
IDL> f=xx+yy*10
IDL> help,xx,yy,f
XX               DOUBLE    = Array[12]
YY               DOUBLE    = Array[12]
F                DOUBLE    = Array[12]
```

File writing:

```
IDL> write_csv,'example.csv',xx,yy,f,header=['X','Y','X+10*Y']
```

Result:

```
X,Y,X+10*Y
0.0000000,0.0000000,0.0000000
1.0000000,0.0000000,1.0000000
2.0000000,0.0000000,2.0000000
(...)
0.0000000,1.0000000,10.000000
1.0000000,1.0000000,11.000000
```

Text files - CSV

Creating a CSV from arrays:

Array of structures: each element (each structure) is one row. Each field is a column:

```
IDL> write_csv_pp,'example1.csv',c,/titles
```

Result:

```
NAME,CALMPOS,FILNAME,ECHLPOS,DISPPOS,TARGNAME,POSDIR,CLASS,MJD_OBS,ITIME,COADDS
"dec18s0001",0,"NIRSPEC-5-AO",62.6300000000000,36.4500000000000,"HD85258","NIRSPEC-
5-AO/p1","STAR","54087.6",100.000000000000,1
"dec18s0002",0,"NIRSPEC-5-AO",62.6300000000000,36.4500000000000,"HD85258","NIRSPEC-
5-AO/p1","STAR","54087.6",100.000000000000,1
"dec18s0003",1,"NIRSPEC-5-AO",62.6300000000000,36.4500000000000,"HD85258","NIRSPEC-
5-AO/p1","FLAT","54087.6",4.60000000000000,5
"dec18s0004",1,"NIRSPEC-5-AO",62.6300000000000,36.4500000000000,"HD85258","NIRSPEC-
5-AO/p1","DARK","54087.6",4.60000000000000,5
"dec18s0005",1,"NIRSPEC-5-AO",62.6300000000000,36.4500000000000,"HD85258","NIRSPEC-
5-AO/p1","ARC","54087.6",5.00000000000000,0
"dec18s0006",1,"NIRSPEC-5-AO",63.5800000000000,36.4500000000000,"HD85258","NIRSPEC-
5-AO/p2","FLAT","54087.6",4.60000000000000,5
...
```

Text files - “regular columns” / “fixed width”

Not exactly a standard, it is a common practice.

Store a table (2D array) as lines. Each column has a constant width.

May have some header lines, describing the file contents, and/or with the column names.

The rest of the file is the table.

Relatively simple to read, though not as much as csv.

Ex: A simple table with only real numbers. 14 columns x 4 lines.

```
   wavl       CH4=3.3      wavl       CH4=2.5      wavl       CH4=1.0      wavl       CH4=0.8      wavl       CH4=0.5      wavl      CH4=0.26      wavl       CH4=0.2
 477.330000   0.090130 477.330000   0.091110 477.330000   0.089250 477.330000   0.087000 477.330000   0.087140 477.330000   0.090080 477.330000   0.088110
 480.040000   0.090930 480.040000   0.091930 480.040000   0.090160 480.040000   0.087930 480.040000   0.088110 480.040000   0.090950 480.040000   0.089090
 482.750000   0.091710 482.750000   0.092730 482.750000   0.091060 482.750000   0.088850 482.750000   0.089080 482.750000   0.091810 482.750000   0.090060
 485.450000   0.092530 485.450000   0.093570 485.450000   0.092000 485.450000   0.089810 485.450000   0.090100 485.450000   0.092730 485.450000   0.091090
```

```
IDL> a=read_ascii('specs_27s_n.txt',data_start=1,header=header)
IDL> print,header
   wavl       CH4=3.3      wavl       CH4=2.5      wavl       CH4=1.0      wavl
CH4=0.8      wavl       CH4=0.5      wavl      CH4=0.26      wavl       CH4=0.2
IDL> help,a
** Structure <d423c9a8>, 1 tags, length=224, data length=224, refs=1:
   FIELD01         FLOAT     Array[14, 4]
```

Text files - “regular columns” / “fixed width”

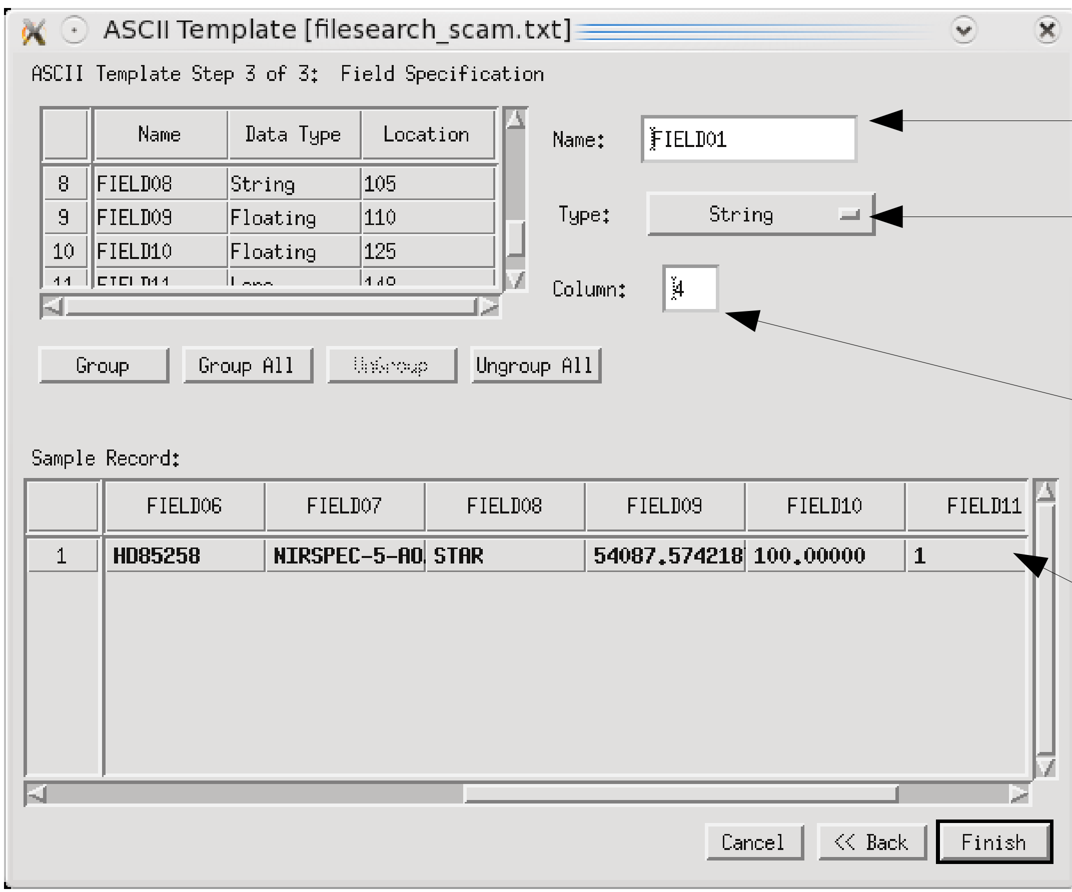
A more difficult case: Not all columns are the same type.

```
      NAME CALMPOS       FILNAME    ECHLPOS    DISPPOS         TARGNAME           POSDIR         CLASS         MJD-OBS     ITIME     COADDS
dec18s0001       0   NIRSPEC-5-AO   62.6300    36.4500          HD85258     NIRSPEC-5-AO/p1         STAR 54087.57421875 100.00000            1
dec18s0002       0   NIRSPEC-5-AO   62.6300    36.4500          HD85258     NIRSPEC-5-AO/p1         STAR 54087.57421875 100.00000            1
dec18s0003       1   NIRSPEC-5-AO   62.6300    36.4500          HD85258     NIRSPEC-5-AO/p1         FLAT 54087.57812500   4.60000            5
dec18s0004       1   NIRSPEC-5-AO   62.6300    36.4500          HD85258     NIRSPEC-5-AO/p1         DARK 54087.57812500   4.60000            5
```

This should be read into structures.

IDL offers an interactive way to specify how to read it:

```
IDL> templ=ascii_template('filesearch_scam.txt')
```



Field name (default being shown)

Field type (with an automatic guess)

Position (character) of the column start (with an automatic guess)

Example line, to check the column separation.

Text files - “regular columns” / “fixed width”

After the **template** is made, the file can be read with just:

```
IDL> a=read_ascii('filesearch_scam.txt',template=templ)
IDL> help,a
** Structure <d4ec4608>, 11 tags, length=416, data length=416,
refs=1:
   FIELD01          STRING    Array[4]
   FIELD02          LONG      Array[4]
   FIELD03          STRING    Array[4]
(...)
   FIELD09          FLOAT     Array[4]
   FIELD10          FLOAT     Array[4]
   FIELD11          LONG      Array[4]
IDL> print,a.field01
dec18s0001 dec18s0002 dec18s0003 dec18s0004
IDL> print,a.field04
      62.6300      62.6300      62.6300      62.6300
```

The **template** (a structure with data on how to read the file) can be used to read other files in the same format.

- Instead of being made interactively, could have been read from some file, or created in the source code.
- Could have specified names for the fields, more useful than things like `FIELD11`.

Text files - “regular columns” / “fixed width”

An alternative: readcol (from IDL Astro library, http://idlastro.gsfc.nasa.gov/contents.html):

File contents:

```
      NAME CALMPOS         FILNAME    ECHLPOS    DISPPOS          TARGNAME               POSDIR          CLASS         MJD-OBS      ITIME     COADDS
dec18s0001       0    NIRSPEC-5-AO    62.6300    36.4500           HD85258    NIRSPEC-5-AO/p1           STAR 54087.57421875 100.00000          1
dec18s0002       0    NIRSPEC-5-AO    62.6300    36.4500           HD85258    NIRSPEC-5-AO/p1           STAR 54087.57421875 100.00000          1
dec18s0003       1    NIRSPEC-5-AO    62.6300    36.4500           HD85258    NIRSPEC-5-AO/p1           FLAT 54087.57812500   4.60000          5
dec18s0004       1    NIRSPEC-5-AO    62.6300    36.4500           HD85258    NIRSPEC-5-AO/p1           DARK 54087.57812500   4.60000          5
```

```
IDL>
readcol,'filesearch_scam.txt',name,calmpos,filname,echlpos,disppos
,targname,posdir,class,mjd_obs,itime,coadds,format='A,I,A,D,D,A,A,
A,D,D,I'
% READCOL: Skipping Line 1
% READCOL: 4 valid lines read
IDL> help
CALMPOS         INT       = Array[4]
CLASS           STRING    = Array[4]
COADDS          INT       = Array[4]
DISPPOS         DOUBLE    = Array[4]
ECHLPOS         DOUBLE    = Array[4]
FILNAME         STRING    = Array[4]
ITIME           DOUBLE    = Array[4]
MJD_OBS         DOUBLE    = Array[4]
NAME            STRING    = Array[4]
POSDIR          STRING    = Array[4]
TARGNAME        STRING    = Array[4]
```

NetCDF, HDF5

Network Common Data Form, Hierarchical Data Format

The only binary standards widely used in science, outside of astronomy (where FITS is king).

Store data as several named variables. Each variable can have metadata:

- Name
- Array dimensions
- Units
- Comments
- Any other attributes (key/value pairs)

Both formats allow for multidimensional arrays.

Only HDF5 allows structures (even complicated structures, not just simple tables).

Selfdocumented: any software that knows these formats can read any file, with no prior knowledge given by the user.

Well supported, in standard and non-standard libraries, interactive software and data visualization software (even web browser plugins):

- http://www.unidata.ucar.edu/software/netcdf/
- http://en.wikipedia.org/wiki/Hdf5

NetCDF is simpler to use, because it has no hierarchy. **If hierarchy is not needed, NetCDF is more convenient than HDF5.**

NetCDF

There is also the **CDL**, an ASCII version of **NetCDF**. NetCDF's standard toolkit has a tool to convert NetCDF to CDL.

Part of a NetCDF file's contents, shown in CDL (made by **ncdump**):

```
$ ncdump -h refspec_g01_0.nc
netcdf refspec_g01_0 {
dimensions:
        nlay = 51 ;
        nwn = 400 ;
        nleg = 33 ;
        numu = 2 ;
        nlev = 52 ;
        nphi = 3 ;
        ngas = 2 ;
        nwnc = 1 ;
        scal = 1 ;
        v3 = 3 ;
        dnl = 1 ;
        nk = 1 ;
        tdisr = UNLIMITED ; // (0 currently)
        na = 16 ;
```

```
variables:

    float alb(nwn) ;
    float z(nlay) ;
    float t(nlay) ;
    float p(nlay) ;
    float wl(nwn) ;
    float iof(nwn) ;
    float mtau(nwn) ;
    float htau(nwn) ;
    float hctaus(nwn) ;
    float gtau(nwn) ;
    float outcos(scal) ;
    float phi(nleg) ;
    float phic(scal) ;
    float umu(numu) ;
    float flux(v3, nwn, nlev) ;
    float mix(nlay, ngas) ;
    float c(nlay) ;
    float psat(nlay) ;
    float ga(nwnc) ;
    float wlc(nwnc) ;
    float inc(scal) ;
    int dm(scal) ;
    int ord(scal) ;
    float fbeam(scal) ;
    float wn(nwn) ;
    float tautot(nwn, nlay) ;
    float htaus(nwn, nlay) ;
    float htaux(nwn, nlay) ;
    float taug(nwn, nlay) ;
    float phase(nwn, nlay, nleg) ;
    float ssa(nwn, nlay) ;
    float uu(nwn, nphi, nlev, numu) ;
    float u0u(nwn, nlev, numu) ;
    float tray(nwn, nlay) ;
    float gtauo(tdisr, nwn, nlay) ;
}
```

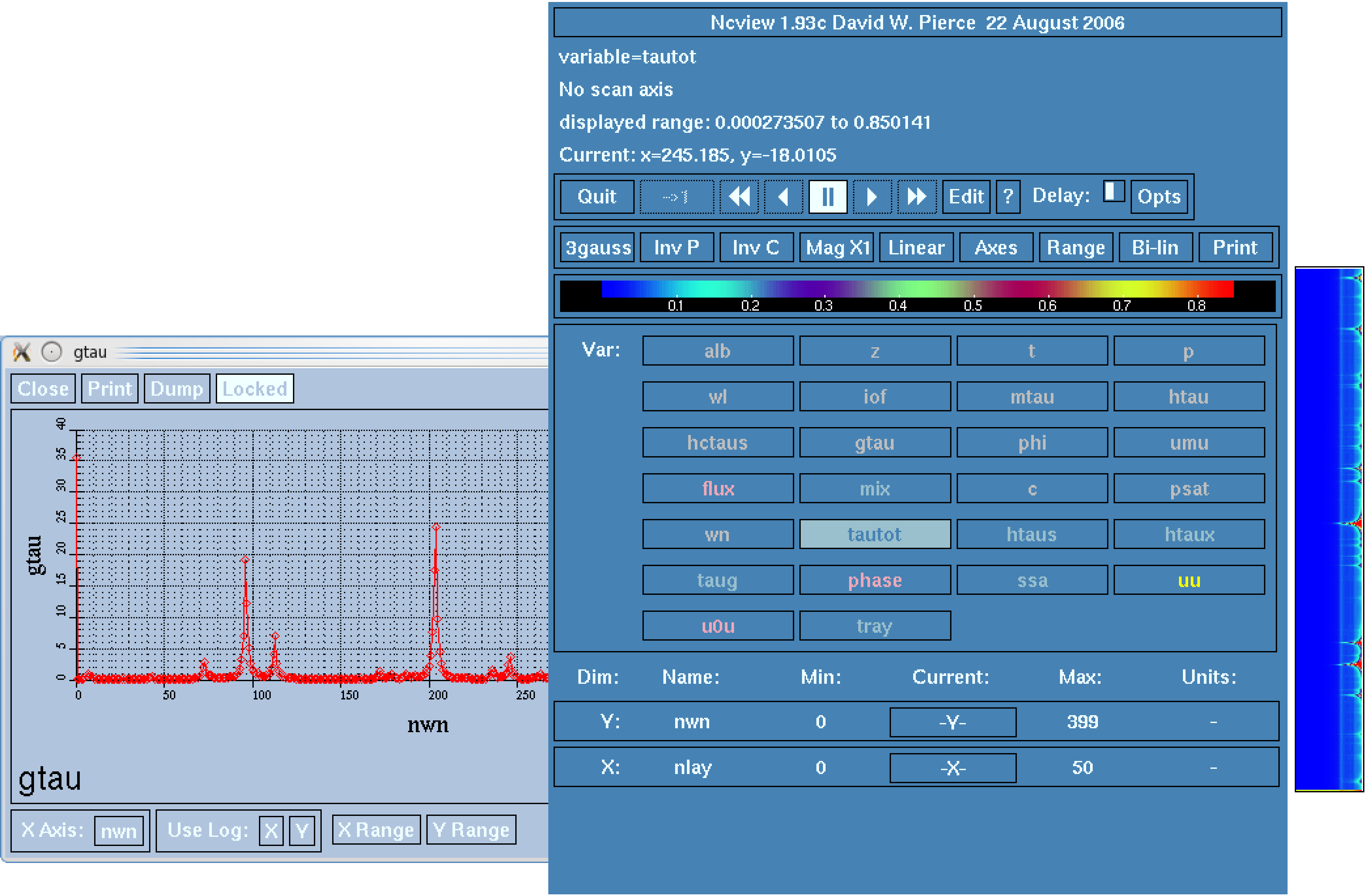
14 variables used as dimensions, for the 35 variables that are arrays

Ex: **`alb(nwn)`** means that **`alb`** is a 1D array of **`nwn`** elements.

`nwn` is stored as one of the dimensions, and is **400.** Therefore, **`alb`** has **400** elements.

NetCDF

Contents of that file shown with **ncview** (part of NetCDF's tookit):



NetCDF

The whole file can be easily read with the available libraries, and stored in convenient containers. Using a reader from pp_lib (http://ppenteado.net/idl):

```
IDL> h=pp_readncdfs('refspec_g01_0.nc',/hash)
IDL> print,h
var_dims: <ObjHeapVar229(HASH)>
vars: <ObjHeapVar152(HASH)>
dims: <ObjHeapVar118(HASH)>

IDL> print,h['vars']
T:      173.203      175.459      175.848      176.001      175.975 ...
Z:      431.869      418.858      406.067      393.403      380.464 ...
FLUX:   3.00660      1.45536     0.705180     0.341669     0.165532 ...
PHASE: 1.00000     0.779034     0.660858     0.542052     0.449983 ...
TAUTOT:0.694377     0.693420     0.693474     0.693540     0.693636...
(...)

IDL> help,(h['vars'])['TAUTOT']
<Expression>    FLOAT     = Array[51, 400]

IDL> print,(h['var_dims'])['TAUTOT']
NLAY NWN

IDL> print,(h['dims'])[(h['var_dims'])['TAUTOT']]
NLAY:       51
NWN:      400
```

NetCDF - writing

Using pp_lib (http://ppenteado.net/idl/pp_lib/doc/index.html):

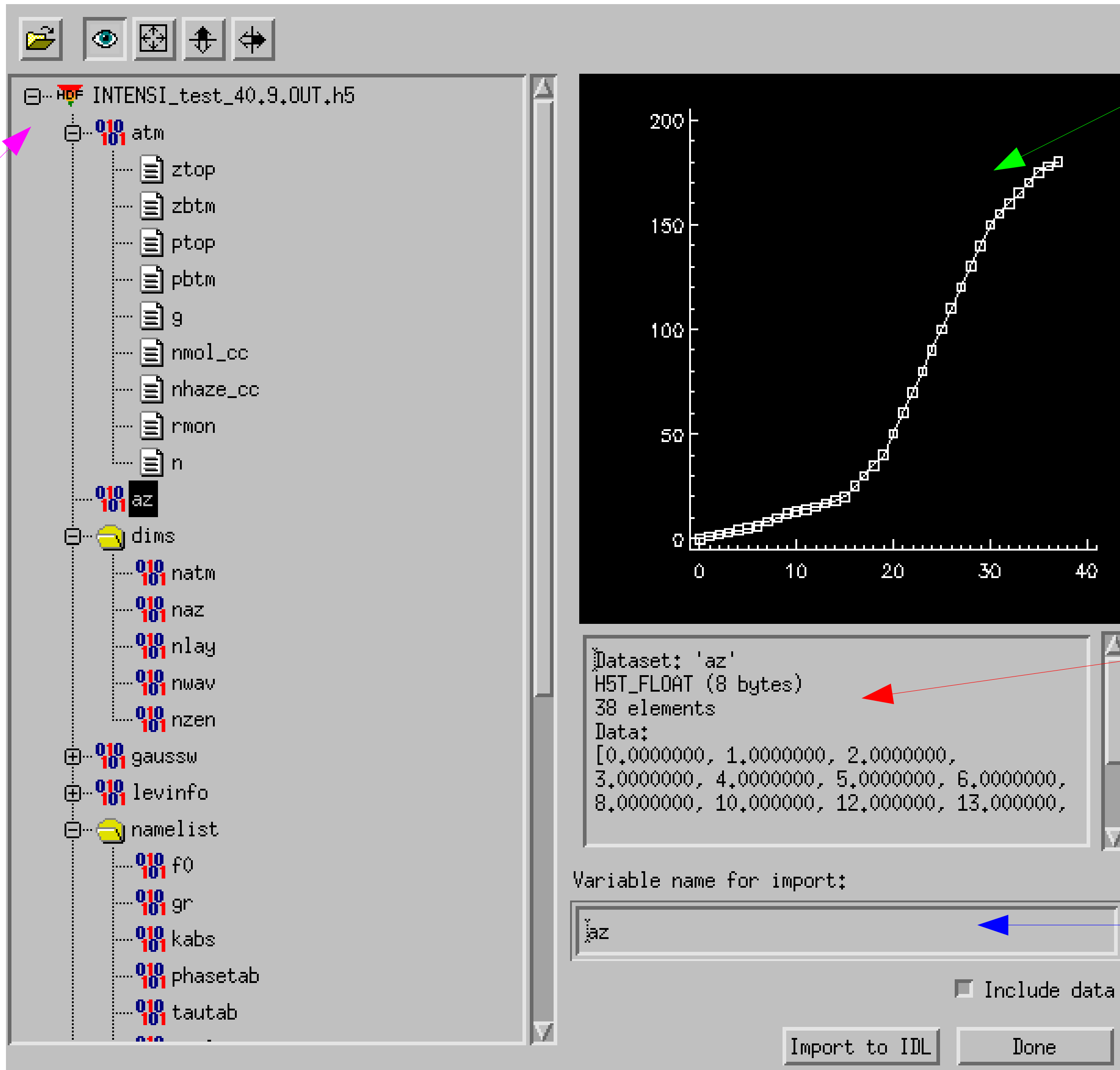
```
IDL> nx=100
IDL> nx=200
IDL> ny=100
IDL> x=dindgen(nx)
IDL> y=dindgen(ny)
IDL> z=dist(nx,ny)
IDL> ncdf={ncdfname:'example.nc',d:{nx:nx,ny:ny},n:
{x:['nx'],y:['ny'],z:['nx','ny']},v:{x:x,y:y,z:z}}
IDL> pp_writencdf,ncdf
netcdf file example.nc done
```

See also http://www.exelisvis.com/docs/NCDF_Overview.html

HDF5

IDL has a graphical tool to browse the contents of an HDF5 file and read the data.

```
IDL> d=h5_browser('INTENSI_test_40.9.OUT.h5')
```



File variables hierarchy.

Visualization of the selected variable.

Metadata and some values of the selected variable.

Variable to be created when importing from the file.

HDF5

IDL has a graphical tool to browse the contents of an HDF5 file and read the data.

```
IDL> d=h5_browser('INTENSI_test_40.9.OUT.h5')
% Imported variable: az

IDL> help,az
** Structure <d5723f38>, 13 tags, length=448, data length=444, refs=1:
   _NAME              STRING    'az'
   _ICONTYPE          STRING    'binary'
   _TYPE              STRING    'DATASET'
   _FILE              STRING    'INTENSI_test_40.9.OUT.h5'
   _PATH              STRING    '/'
   _DATA              DOUBLE    Array[38]
   _NDIMENSIONS       LONG                 1
   _DIMENSIONS        ULONG64   Array[1]
   _NELEMENTS         ULONG64                       38
   _DATATYPE          STRING    'H5T_FLOAT'
   _STORAGESIZE       ULONG                8
   _PRECISION         LONG                64
   _SIGN              STRING    ''

IDL> print,az._data
       0.0000000       1.0000000       2.0000000       3.0000000
4.0000000       5.0000000       6.0000000       8.0000000
(...)
```

HDF5 – non-interactive reading

```
PRO ex_read_hdf5
file = FILEPATH('hdf5_test.h5', $
SUBDIRECTORY=['examples', 'data'])
file_id = H5F_OPEN(file)
; Open the image dataset within the file.
; This is located within thhtml/images group.
; We could also have used H5G_OPEN to open up the group first.
dataset_id1 = H5D_OPEN(file_id, '/images/Eskimo')
; Read in the actual image data.
image = H5D_READ(dataset_id1)
; Open up the dataspace associated with the Eskimo image.
dataspace_id = H5D_GET_SPACE(dataset_id1)
; Retrieve the dimensions so we can set the window size.
dimensions = H5S_GET_SIMPLE_EXTENT_DIMS(dataspace_id)
; Now open and read the color palette associated with this image.
dataset_id2 = H5D_OPEN(file_id, '/images/Eskimo_palette')
palette = H5D_READ(dataset_id2)
H5S_CLOSE, dataspace_id
H5D_CLOSE, dataset_id1
H5D_CLOSE, dataset_id2
H5F_CLOSE, file_id
; Display the data.
DEVICE, DECOMPOSED=0
WINDOW, XSIZE=dimensions[0], YSIZE=dimensions[1]
TVLCT, palette[0,*], palette[1,*], palette[2,*]
TV, image, /ORDER
END
```

(from http://www.exelisvis.com/docs/HDF5_Overview.html)

HDF5 - writing

```
PRO ex_create_hdf5
file = filepath('hdf5_out.h5')
fid = H5F_CREATE(file)
;; create data
data = hanning(100,150)
;; get data type and space, needed to create the dataset
datatype_id = H5T_IDL_CREATE(data)
dataspace_id = H5S_CREATE_SIMPLE(size(data,/DIMENSIONS))
;; create dataset in the output file
dataset_id = H5D_CREATE(fid,$
'Sample data',datatype_id,dataspace_id)
;; write data to dataset
H5D_WRITE,dataset_id,data
;; close all open identifiers
H5D_CLOSE,dataset_id
H5S_CLOSE,dataspace_id
H5T_CLOSE,datatype_id
H5F_CLOSE,fid
END
```

(from http://www.exelisvis.com/docs/HDF5_Overview.html)

Image formats

JPEG, JPEG200, TIFF, GeoTIFF, GIF, PNG, etc.

Well-supported in many platforms, with libraries ready to read and write them.

Some formats have compression, which may be lossless (PNG, JPEG, JPEG2000, TIFF) or lossy (JPEG, JPEG200, TIFF).

Store images as 2D or 3D arrays (3D often limited to 3 or 4 in one of the dimensions: an image in 3 or 4 bands), of integers or (in only a few formats) reals.

Some formats allow storing metadata. (most importantly, JPEG, GeoTIFF)

JPEG2000 and GeoTIFF (a type of TIFF) common in astronomy, remote sensing, geosciences.

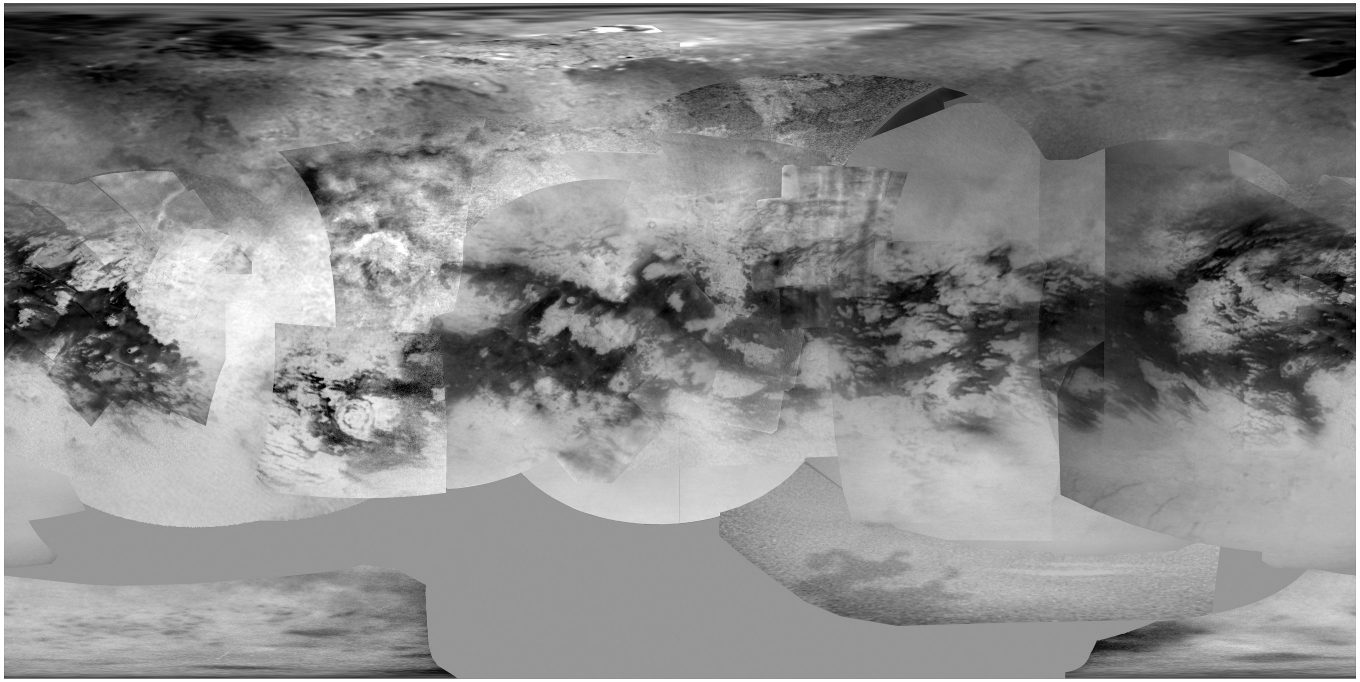


Image reading / writing examples - GeoTIFF

```
IDL> print,query_tiff('issmap_2009.tiff',info,geotiff=geo)
           1
```

Obtain data about the file

```
IDL> help,info
** Structure <d4eca4b8>, 18 tags, length=144, data length=132, refs=1:
   CHANNELS        LONG                     4
   DIMENSIONS      LONG         Array[2]
   IMAGE_INDEX     LONG                     0
   NUM_IMAGES      LONG                     1
   PIXEL_TYPE      INT                  1
   TYPE            STRING       'TIFF'
   BITS_PER_SAMPLE LONG                     8
   POSITION        FLOAT        Array[2]
   RESOLUTION      FLOAT        Array[2]
   UNITS           LONG                     2
   TILE_SIZE       LONG         Array[2]
   DESCRIPTION     STRING       'ISS (2009)'
   DATE_TIME       STRING       '2010:02:18 01:24:36'
(...)

IDL> help,geo
** Structure <d4efa1b8>, 10 tags, length=264, data length=262, refs=1:
   MODELPIXELSCALETAG                  DOUBLE    Array[3]
   MODELTIEPOINTTAG                    DOUBLE    Array[6, 4]
   GEOGRAPHICTYPEGEOKEY                INT           4035
   GEOGSEMIMAJORAXISGEOKEY             DOUBLE           2575000.0
   GEOGSEMIMINORAXISGEOKEY             DOUBLE           2575000.0
(...)

IDL> myimage=read_tiff('issmap_2009.tiff')
```

Reads the image into **myimage** array

Image reading / writing examples – most formats

JPEG, JPEG200, TIFF, GeoTIFF, GIF, PNG, BMP, DICOM. PPM, SRF.

Reading, writing, visualizing:

```
IDL> im=read_image('issmap_2009.tiff')
% Loaded DLM: TIFF.
IDL> help,im
IM              BYTE      = Array[4, 4046, 2023]
IDL> write_image,'issmap_2009.png','png',im
% Compiled module: WRITE_IMAGE.
% Loaded DLM: PNG.
IDL> iopen,'issmap_2009.png',im2,/visualize
IDL> help,im2
IM2             BYTE      = Array[4, 4046, 2023]
IDL> print,array_equal(im,im2)
   1
```

Files – low level processing

- Not always a standard format can do what is necessary.

- Not always there is a choice (you may need to read someone's proprietary format).

- The available support to a standard format might not be enough.

Then you need to write your own low-level routines to process the file.

- Preferably, with a nice high-level interface (interactive or not).

Low level processing - text

Taking the previous example, where columns are of different types:

```
      NAME CALMPOS        FILNAME    ECHLPOS    DISPPOS          TARGNAME              POSDIR          CLASS        MJD-OBS      ITIME     COADDS
dec18s0001       0   NIRSPEC-5-AO    62.6300    36.4500           HD85258     NIRSPEC-5-AO/p1           STAR 54087.57421875 100.00000              1
dec18s0002       0   NIRSPEC-5-AO    62.6300    36.4500           HD85258     NIRSPEC-5-AO/p1           STAR 54087.57421875 100.00000              1
dec18s0003       1   NIRSPEC-5-AO    62.6300    36.4500           HD85258     NIRSPEC-5-AO/p1           FLAT 54087.57812500   4.60000              5
dec18s0004       1   NIRSPEC-5-AO    62.6300    36.4500           HD85258     NIRSPEC-5-AO/p1           DARK 54087.57812500   4.60000              5
```

The easiest way to read it directly is to define a structure to get each line, and an array of structures to get the whole file. :

```
IDL>
record={name:'',calmpos:0,filname:'',echlpos:0d0,disppos:0d0,targname:
'',posdir:'',class:'',mjd_obs:'',itime:0d0,coadds:0}
IDL> nlines=file_lines('filesearch_scam.txt')
IDL> mydata=replicate(record,nlines-1)
IDL> header=''
```

After the variables to receive the data were created, the file can be read:

```
IDL> openr,unit,'filesearch_scam.txt',/get_lun
IDL> readf,unit,header
IDL> readf,unit,mydata,format='(A14,I8,A15,F10.4,F10.4,A18,A19,A15,F15.8,F10.5,I15)'
IDL> free_lun,unit
```

The file is opened to a **unit**, which will be used to specify from which source reading will be done.

At the end, close the file.

Writing the file could have been done in almost the same way, just replacing **`openr`** by **`openw`** (open file for writing, not reading), and **`readf`** by **printf**. (read instead of write the file).

Low level processing - text

Which results in:

```
IDL> print,header
          NAME CALMPOS        FILNAME   ECHLPOS   DISPPOS          TARGNAME
          POSDIR          CLASS        MJD-OBS     ITIME    COADDS

IDL> help,mydata
MYDATA           STRUCT    = -> <Anonymous> Array[4]

IDL> help,mydata[0]
** Structure <d5e38f68>, 11 tags, length=136, data length=124, refs=3:
   NAME            STRING    '    dec18s0001'
   CALMPOS         INT              0
   FILNAME         STRING    '   NIRSPEC-5-AO'
   ECHLPOS         DOUBLE           62.630000
   DISPPOS         DOUBLE           36.450000
   TARGNAME        STRING    '           HD85258'
   POSDIR          STRING    '    NIRSPEC-5-AO/p1'
   CLASS           STRING    '           STAR'
   MJD_OBS         STRING    '       54087.574'
   ITIME           DOUBLE           100.00000
   COADDS          INT              1
```

This case is more complicated because there are columns of strings.

If all columns were numbers, it would not have been necessary to explicitly give the format:

- The column start/end would have been guessed by the reading library.
- Could have been read / written with just an array of numbers (if all are the same type), without need for structures.

Low level processing - binary

Binary formats can be defined in any way. The file is just a set of bytes. **It is up to your program to know how to interpret it.**

A common choice to make them selfodcumented is to include a text header, informing the data characteristics needed to read the data.

Ex: Create a file that stores an array of doubles, with a header informing the dimensions, so that the reading program knows how to read the data:

Make up some data and open the file:

```
IDL> data_to_write=dindgen(3,4)
IDL> openw,unit,'binary_example.dat',/get_lun
```

Write the dimensions, as text:

```
IDL> printf,unit,'dimensions of the double array stored below:'
IDL> printf,unit,size(data_to_write,/dimensions)
```

Write the data, in binary:

```
IDL> writeu,unit,data_to_write
```

Writes in binary (**unformatted** write).

Close the file:

```
IDL> free_lun,unit
```

Low level processing - binary

What the file looks like, if seen in a common text editor (which assumes text coding):

```
dimensions of the double array stored below:
           3           4
^@^@^@^@^@^@^@^@^@^@^@^@^@^@ð^@^@^@^@^@^@^@@^@^@^@^@^@^@^H@^@^@^@^@^
@^@^P@^@^@^@^@^@^T@^@^@^@^@^@^X@^@^@^@^@^@^\@^@^@^@^@^@^@@^@^@
^@^@^@^@"@^@^@^@^@^@^@$@^@^@^@^@^@^@&@
```

Result of interpreting the binary data as if it was text.

Low level processing - binary

Reading this file is equally simple:

```
IDL> header=''
IDL> dims=[0,0]
IDL> openr,unit,'binary_example.dat',/get_lun
```

Reading the dimensions (stored as text):

```
IDL> readf,unit,header
IDL> readf,unit,dims
```

Reading the binary part, after knowing the dimensions:

```
IDL> data_read=dblarr(dims)
IDL> readu,unit,data_read
```

Read binary data (**unformatted** read)

Close the file:

```
IDL> free_lun,unit
```

Verify that the data were written and read correctly:

```
IDL> print,array_equal(data_to_write,data_read)
   1
```

References

FITS
http://idlastro.gsfc.nasa.gov/contents.html

ISIS Cubes
http://ppenteado.net/idl/pp_lib/doc/index.html

CDF / NetCDF
http://www.exelisvis.com/docs/routines-100.html
http://www.exelisvis.com/docs/routines-101.html
http://www.unidata.ucar.edu/software/netcdf/

HDF/HDF5
http://www.exelisvis.com/docs/routines-103.html
http://www.exelisvis.com/docs/routines-102.html
http://www.hdfgroup.org/HDF5/

EOS
http://www.exelisvis.com/docs/routines-138.html

GRIB
http://www.exelisvis.com/docs/routines-104.html

GeoTIFF
http://www.idlcoyote.com/map_tips/autogeoreg.html

Other formats (including text, binary, image, sound, video, Google Maps, shapefile)
http://www.exelisvis.com/docs/routines-1.html