

IDL Week 2:

What we'll cover today

- IDL data structures
- File I/O in IDL
- Reading and writing ASCII (text) files
- Reading and writing unformatted binary files
- Reading and writing to scientific data formats
 - netCDF
 - HDF
- Reading data from images

Getting information about a file

There are two routines which are useful for getting information about files before opening them:

file_lines(*filename*) – returns the number of lines in an ASCII file

file_info(*filename*) – returns an IDL data structure of file information (existence, readability, writeability, time of creation, size, and more)

IDL Data structures

A structure is a way of encapsulating different data types into a single variable. Fields of a structure are accessed as *structname.fieldname*. If the field is an array, its elements may be accessed using the usual bracket notation:

structname.fieldname[index]

To discover what fields are in a structure, use the

TAG_NAMES function:

print, TAG_NAMES(structname)

To create a structure:

structname = **CREATE_STRUCT**(*fieldname1*,
data1, *fieldname2*, *data2*,...)

Opening and closing files

There are three commands to open files, depending on the intended use (this applies to ASCII and binary files). They are:

openr – open for reading

openw – open for writing (erases file if it already exists)

openu – open for updating

Syntax: **openr**, *lun*, *filename*[**/get_lun**]

lun stands for **L**ogical **U**nit **N**umber. It's IDL's way of identifying the file internally, and is just a long integer. To have IDL automatically assign a number (recommended), use the **/get_lun** keyword.

To close a file: **free_lun**, *lun*

ASCII text: Pros and Cons



- Human readable
- Universal common format
- Can have a mix of text and numbers



- Lots of I/O overhead
- Numbers use unnecessary amounts of disk space
- Subject to formatting errors

Reading in a free-format file

Free-format ASCII files use whitespace or commas to separate variables. Because there is no explicit format, IDL uses 7 rules to interpret the data using the **readf** command.

Syntax: **readf**, *lun*, *variable*

Important Note: *variable* can't be a subscript of an array – this is one case where IDL passes variables by value, and not by reference. If you try to pass **readf** an array subscript, the array will not be modified!

Free-format rules

Rule #1: If reading into a string variable, all characters remaining on the current line are read into the variable.

Rule #2: Input data must be separated by commas or whitespace (spaces or tabs).

Rule #3: When reading into an array of n elements, IDL will try to read n separate values from the file.

Rule #4: If the current input line is empty, and there are still variables left requiring input, read another line.

Rule #5: If the current input line is not empty, but there are no variables left requiring input, ignore the remainder of the line.

Rule #6: Data is converted into the type specified by the variable (or assumed, if variable has not been initialized)

Rule #7: Complex numbers must be in the (r,i) format. Attempts to read numbers in other formats will be read into the real part, while the imaginary part is set to zero.

Example: Reading a free-format file

File contents:

20.2	22.93
22.23	26.1
19.92	23.55
19.33	24.7
20.68	24.25
24.83	29.47
23.88	29.53
24.55	29.1
22.48	27.17
22.32	27.28
25.62	30.08
23.3	28.75
20.98	24.55
22.68	28.08
23.92	26.52
19.73	24.37
24.88	29.03
27.12	32.48
20.92	26.25
27.55	33.52
27.58	35.18
27.4	31.47
25.87	30.52
25	29.5

Sample code:

```
IDL> print, file_lines('sample1.txt')
                24
IDL> openr, lun, 'sample1.txt', /get_lun
IDL> readf, lun, data1
IDL> print, data1
                20.2000
IDL> readf, lun, data2
IDL> print, data2
                22.2300
IDL> data = fltarr(2,24)
IDL> readf,lun,data
% READF: End of file encountered. Unit: 100, File: sample1.txt
% Execution halted at: $MAIN$
What happened? Read beyond the end of the file(because we had already
read in the first 2 lines)
IDL> point_lun, lun, 0
IDL> readf,lun,data
IDL> print, min(data), max(data)
                19.3300    35.1800
```

Reading explicitly-formatted text

If you know ahead of time that a line of data follows a certain format, specifying that format will resolve some of the ambiguities that come with free-formatted files.

Format codes are the same as in fortran:

$Cw[.x]$, where C specifies the type, w is the field width, and x is optional additional number whose meaning depends on the type.

Format codes

Type	Code	x specifies:
Integer	I	pad with 0 up to x digits
Float	F	number of digits after the decimal point
Double	D	number of digits after the decimal point
String	A	not used
Exponential	E	number of digits after the decimal point

Consider the number 9438.68221

I4 will display as 9438

I6.5 will display as b09438 (b is a blank)

F8.2 will display as b9438.68

E12.5 will display as b9.43868E+03

Example: Reading a formatted file

File contents:

020102.23577.output.250	4058	6489	6600	1635.22	1605.93
020103.23586.output.250	1901	2587	2684	109.90	148.08
020106.23632.output.250	1896	2707	2805	506.67	506.06
020106.23638.output.250	3014	6462	6569	1081.27	1078.18
020110.23693.output.250	3514	2666	2777	69.45	34.06
020113.23745.output.250	2487	6550	6653	4.66	2.07
020117.23815.output.250	3314	2610	2721	49.42	27.21
020120.23861.output.250	817	2741	2805	6.49	3.35
020121.23867.output.250	3970	6486	6596	14.89	2.83
020125.23928.output.250	2741	6457	6562	64.01	30.20

Sample code:

```
IDL> print, file_lines('sample2.txt')
      10
IDL> openr, lun, 'sample2.txt', /get_lun
IDL> readf, lun, line
% READF: Input conversion error. Unit: 100, File:
      sample.txt
% Execution halted at: $MAIN$
What happened? IDL was expecting a numeric data type
(line was undefined)
IDL> point_lun, lun, 0
IDL> sdata = string(0)
IDL> idata = intarr(3)
IDL> fdata = fltarr(2)
IDL> readf, lun, sdata, idata, fdata,
      format='(A59,3I8,2F9.2)'
IDL> print, sdata
      input/PR_profiles/020102.23577/020102.23577.output.
      250
IDL> print, idata
      4058  6489  6600
IDL> print, fdata
      1635.22  1605.93
```

Another option: READ_ASCII

The **READ_ASCII** function provides more flexibility than the standard free-format or formatted read routines.

Syntax: *result* = **READ_ASCII**(*filename*, *keywords*)

result is a data structure containing a variable *FIELD01*, which contains the columns of data. Keywords include:

COMMENT_SYMBOL: ignore lines that begin with this string

COUNT: number of records read

DATA_START: number of lines to skip

DELIMITER: string that separates columns

HEADER: string array that stores skipped lines

MISSING_VALUE: value to replace missing or invalid data

NUM_RECORDS: number of records to read

RECORD_START: number of records to skip

Writing ASCII text

Use the **printf** command to write data to a file.

Syntax: **printf**, *lun*, *var1*[,*var2*,...][,**format**='*fmt*']

If the format keyword is not specified, IDL will use the same conventions as when writing to the screen (all of the significant digits of a number and the entirety of a string).

Note: IDL assumes an 80 character line width when writing free-formatted text. This can be changed with the **width** keyword of the **openw** command.

Binary files: Pros and Cons



- No overhead – data is stored on disk the same way it is in memory
- Smaller size than the equivalent ASCII



- Not readable in a text editor
- Need to know data type and record length
- Subject to byte-swapping issues
- Can't store text

Reading a binary file

Use the same **openr** command as for text files.
Instead **readf**, use the **readu** command:
readu, *lun*, *var1*, *var2*, ...

Important: the variables must be the same data type
(fix, long, float, double) as they are written!
Otherwise, a number will be read, but it will not
have a meaningful value.

Example: Reading a formatted file

(File 'ned.flt' contains floating point values of elevation near Ft. Collins in meters)

```
IDL> openr, lun, 'ned.flt', /get_lun
IDL> data = fltarr(8)
IDL> readu, lun, data
IDL> print, data
    1596.91    1596.21    1595.50    1594.78    1594.28    1594.28    1594.11    1593.53
IDL> point_lun, lun, 0
IDL> data = intarr(8)
IDL> readu, lun, data
IDL> print, data
   -25290  17607 -31083  17607  28642  17607  22764  17607
IDL> point_lun, lun, 0
IDL> data = lonarr(8)
IDL> readu, lun, data
IDL> print, data
  1153932598 1153926805 1153920994 1153915116 1153911070 1153911039 1153909628 1153904915
IDL> point_lun, lun, 0
IDL> data = dblarr(8)
IDL> readu, lun, data
IDL> print, data
  2.2219410e+23 2.2050954e+23 2.1992198e+23 2.1903942e+23 2.1721016e+23 2.1478211e+23
    2.1224496e+23
    2.0961544e+23
IDL> free_lun, lun
```

Byte-swapping issues

At some point, you may run into a situation where you know you are reading into the correct variable type, but the numbers are still not meaningful. If the file was created on another platform, chances are it was not written in the same byte order as your machine prefers. In this case, use the ***/SWAP_ENDIAN*** keyword with **openr**.

Reading binary files from fortran

Fortran writes data to binary files as records, which have a header and a trailer (which is identical to the header). Depending on the machine it was written on, the header/trailer may consist of one or two short (2-byte) or long(4-byte) integers.

For binary files, the **point_lun** command sets the input pointer to the specified byte in the file, which will be a multiple of the fortran record number (equal to the value in the header plus the space taken by the header+trailer)!

Example: Reading fortran binary

File contents:

```
36
184
-33.7065
-35.2532
1.00000
0.927274
0.232121
0.00390695
0.562236
0.250162
0.250162
1.83375
0.634754
0.00136423
0.00000
0.00000
0.00000
```

Read as long:

```
68 <-header
36
184
-1039740051
-1039334595
1065353216
1064133084
1047376182
998245850
1058008762
1048581448
1048581448
1072347230
1059225411
984797186
0
0
0
68 <-trailer
```

Read as float:

```
9.52883e-44<-header
5.04467e-44
2.57839e-43
-33.7065
-35.2532
1.00000
0.927274
0.232121
0.00390695
0.562236
0.250162
0.250162
1.83375
0.634754
0.00136423
0.00000
0.00000
0.00000
9.52883e-44<-trailer
```

Writing binary files

Use the **writeu** command to write data to a file.

Syntax: **writeu**, *lun*, *var1*[,*var2*,...]

IDL simply dumps the data from memory to the disk without any conversion, or adding a header and footer like fortran does.

Scientific data formats – the best of both worlds

- netCDF and HDF file formats offer the advantages of binary with metadata and portability
 - Some overhead compared to binary, but much less than ASCII
 - Able to store dimensional information (arrays)
- Many scientific data sets intended for wide distribution use one of these formats
- IDL comes with special toolkits to read from and write to these formats

About netCDF files

- netCDF (network Common Data Format) is designed to be simple and flexible.
- The basic building blocks are:
 - Variables
 - scalars or multidimensional arrays
 - string, byte, int, long, float and double
 - Attributes
 - contain supplementary information about a single variable or an entire file
 - Variable attributes include: units, valid range, scaling factor.
 - Global attributes contain information about the file i.e., creation date.
 - Attributes are scalars or 1-D arrays.
 - Dimensions
 - long scalars that record the size of one or more variables

What's in a netCDF file?

Option 1: Use the ncdump command in unix:

```
ncdump -h zg_ukmo.nc
```

```
dimensions:
```

```
    lon = 96 ;
```

```
    lat = 73 ;
```

```
    plev = 15 ;
```

```
    time = UNLIMITED ; // (600 currently)
```

```
    bnds = 2 ;
```

```
variables:
```

```
    double lon(lon) ;
```

```
    lon:standard_name = "longitude" ;
```

```
    lon:long_name = "longitude" ;
```

```
    lon:units = "degrees_east" ;
```

```
    lon:axis = "X" ;
```

```
    lon:bounds = "lon_bnds" ;
```

```
    ...
```

```
// global attributes:
```

```
    :title = "Met Office  model output prepared for IPCC Fourth Assessment  
    climate of the 20th Century experiment (20C3M)" ;
```

```
    ...
```

What's in a netCDF file?

Option 2: Use the **ncdf_inquire()** and **ncdf_varinq()** commands:

```
IDL> ncid = ncdf_open('zg_ukmo.nc')
IDL> ncinfo = ncdf_inquire(ncid)
IDL> print, tag_names(ncinfo)
NDIMS NVARS NGATTS RECDIM
IDL> print, ncinfo.NVARS
      8
IDL> varinfo = ncdf_varinq(ncid,2)
IDL> print, tag_names(varinfo)
NAME DATATYPE NDIMS NATTS DIM
IDL> print, varinfo.NAME
lat
IDL> print, varinfo.DATATYPE
DOUBLE
IDL> print, varinfo.NDIMS
      1
IDL> print, varinfo.NATTS
      5
IDL> print, varinfo.DIM
      1
```

Reading data from netCDF files

Once you know the name (or id number) of a variable, use the **ncdf_varget** command to extract the data you want:

```
IDL> ncid = ncdf_open('zg_ukmo.nc')
IDL> ncdf_varget,ncid,'lat', lat
IDL> help, lat
LAT          DOUBLE   = Array[73]
IDL> ncdf_varget,ncid,2, lat
IDL> help, lat
LAT          DOUBLE   = Array[73]
IDL> ncdf_close, ncid
```

Always close the file with **ncdf_close**!

Reading a netCDF attribute

Use the **ncdf_attname** command to learn attribute names, then use **ncdf_attget** to extract them:

```
IDL> for i=0,6 do print, ncdf_attname(ncid,'zg',i)
standard_name
long_name
units
cell_methods
_FillValue
missing_value
history
IDL> ncdf_attget,ncid,'zg','standard_name',zg_name
IDL> print, zg_name
103 101 111 112 111 116 101 110 116 105 97 108 95 104 101 105 103 104 116
What happened? Was expecting a string
IDL> help, zg_name
ZG_NAME      BYTE      = Array[19]
IDL> print, string(zg_name)
geopotential_height
```

Writing to netCDF

Step 1: create the file using the **ncdf_create** function:

```
ncid = ncdf_create(filename[,/CLOBBER])
```

Step 2: Create global attributes with **ncdf_attput**:

```
ncdf_attput, ncid, name, value,/GLOBAL
```

Step 3: Define variable dimensions (e.g., time and space)

using **ncdf_dimdef** (only 1 dimension may be unlimited):

```
dimid =
```

```
ncdf_dimdef(ncid,dimname,size[,/UNLIMITED])
```

Writing to netCDF

Step 4: Define a variable using the **ncdf_vardef** command:

varid = **ncdf_vardef**(*ncid*, *varname*,
[*dimid1*, *dimid2*, ...], **/TYPE**)

(**TYPE** can be **BYTE**, **CHAR**, **DOUBLE**, **FLOAT**, **LONG**, or **SHORT**)

Step 5: Define variable attributes using **ncdf_attput**:

ncdf_attput, *ncid*, *varid* (or *varname*), *name*, *value*

Step 6: exit define mode:

ncdf_control, *ncid*, **/ENDEF**

Step 7: Write variables using **ncdf_varput**:

ncdf_varput, *ncid*, *varid* (or *varname*), *value(s)*

Step 8: Close netCDF file:

ncdf_close, *ncid*

About HDF files

- HDF stands for **H**ierarchical **D**ata **F**ormat
- Similar concept to netCDF, but with some added functionality
 - With added functionality comes added complexity (150 HDF functions vs. 23 netCDF functions!)
- HDF is the format used by NASA's EOS
- In HDF, data can be stored as raster (image) data, annotation data, scientific data, vdata and vgroups
 - Scientific Data consists of datasets (arrays) and attributes (metadata)
 - Vdata represents vector data (tables)
 - Vgroups consist of one or more fields of Vdata

What's in an HDF file?

For Scientific Datasets, open the HDF file using the Scientific Dataset interface:

```
hdfid = HDF_SD_Start(filename)
```

Next, get number of datasets and attributes using **HDF_SD_fileinfo**:
HDF_SD_Fileinfo, *hdfid*, *n_datasets*, *n_attributes*

List the datasets and attributes using the **HDF_SD_AttrInfo** and **HDF_SD_GetInfo** functions:

```
HDF_SD_ATTRINFO, hdfid, Attr_Index, NAME=variable
```

```
HDF_SD_GETINFO, sdid, NAME=variable
```

Note: The dataset ID (*sdid*) is not the same as the file ID (*hdfid*) – you will need to get it from the **HDF_SD_Select** function:

```
sdid = HDF_SD_Select(hdfid, dataset_index)
```

Example: Reading datasets and attribute names

Example Code (read_hdf.pro):

```
HDF_SD_Fileinfo,hdfid, nd, na
for j=0,na-1 do begin
  HDF_SD_AttrInfo, hdfid,j, Name=thisSDSname
  print, 'Attribute No. ', StrTrim(j, 2), ': ', thisSDSname
endfor
for j=0,nd-1 do begin
  thisSDS = HDF_SD_Select(hdfid, j)
  HDF_SD_GetInfo, thisSDS, Name=thisSDSname
  print, 'Dataset No. ', StrTrim(j, 2), ': ', thisSDSname
endfor
```

Output:

```
IDL> read_hdf, '1B11.000130.12519.5.HDF'
Attribute No. 0: CoreMetadata.0
Attribute No. 1: ArchiveMetadata.0
Dataset No. 0: geolocation
Dataset No. 1: calCounts
Dataset No. 2: satLocZenAngle
Dataset No. 3: lowResCh
Dataset No. 4: highResCh
```

Extracting a Scientific Dataset

Once you know the name of the dataset you wish to extract, use the following sequence of commands to obtain the data:

```
index=hdf_sd_nametoindex(hdfid,name)  
varid=hdf_sd_select(hdfid,index)  
hdf_sd_getdata,varid,varname
```

Always close the HDF file with the following commands:

```
HDF_SD_EndAccess, varid  
HDF_SD_End, hdfid
```

Dataset Extraction Example

```
IDL> hdfid = HDF_SD_Start('1B11.000130.12519.5.HDF')
% Loaded DLM: HDF.
IDL> index = hdf_sd_nametoindex(hdfid,'geolocation')
IDL> varid = hdf_sd_select(hdfid,index)
IDL> hdf_sd_getdata,varid,geoloc
IDL> help, geoloc
GEOLOC      FLOAT    = Array[2, 208, 2985]
IDL> print, geoloc[*,0,0]
   -31.5610   -75.8859
IDL> hdf_SD_EndAccess,varid
IDL> hdf_SD_End,hdfid
```

Exploring Vdata

Use a different open command for Vdata than for SD data:

```
hdfid = HDF_Open(filename)
```

Because Vdata is stored hierarchically in Vgroups, need to use recursive combination of **HDF_VG** and **HDF_VD** commands (see [read_hdf.pro](#)) to see what Vdata tables are available.

Extracting Vdata

Once you know the index of the vdata table you wish to extract, use the following commands:

```
vdid = HDF_VD_Attach(hdfid, vindex)
```

```
nrec = HDF_VD_Read(vdid, var[, FIELDS=string])
```

```
HDF_VD_Detach, vdid
```

```
...
```

```
HDF_Close, hdfid
```

Vdata Extraction Example

```
IDL> hdfid = hdf_open('1B11.000130.12519.5.HDF')
IDL> vdid = HDF_VD_Attach(hdfid, 6)
IDL> nrec = HDF_VD_Read(vdid, satloc, FIELDS='scPosX,scPosY,scPosZ')
IDL> print, nrec
      2985
IDL> help, satloc
SATLOC      FLOAT    = Array[3, 2985]
IDL> hdf_vd_detach, vdid
IDL> hdf_close, hdfid
```

Creating and Writing HDF files

The process of writing to an HDF file is similar to reading. Begin by opening the file with the **/CREATE** keyword:

```
hdfid = HDF_SD_Start(filename,/CREATE)
```

To create a scientific dataset, use the **HDF_SD_Create** and **HDF_SD_AddData** commands:

```
sdid = HDF_SD_Create(hdfid,name,[dimensions],/TYPE  
(TYPE can be FLOAT, DOUBLE, INT, LONG, STRING)  
HDF_SD_AddData, sdid, var
```

Remember to close the dataset and the file:

```
HDF_SD_EndAccess, sdid
```

```
HDF_SD_End, hdfid
```

Reading image data

In adding to ASCII, binary, and scientific data formats, IDL can also read in image data in a variety of formats. These have a common syntax:

Read_XXXX, *filename*, *array*[:,*R,G,B*]

where XXXX is the file type (e.g., GIF, JPEG, PNG, TIFF – see references for a full list), and the returned array has the dimensions of the image, is valued from 0 to 255, and may contain three sub-arrays representing the RGB components of a true color image.

Writing Image Data

Similarly, IDL can write 2D byte arrays to an image file.

The syntax is:

WRITE_XXXX, *filename*, *image_array*[,**/ORDER**]

Note that *image_array* must have either two dimensions or two dimensions and a third with length three (representing RGB components). *image_array* will be converted to type byte (values 0 to 255) if necessary.

The **/ORDER** keyword reverses the vertical dimension of the image array.