

# I/O OPERATIONS

KONRAD HINSEN

CENTRE DE BIOPHYSIQUE MOLÉCULAIRE (ORLÉANS)

AND

SYNCHROTRON SOLEIL (ST AUBIN)



# TEXT AND BINARY FILES

There are two big families of file format:

- text files store data as human-readable text
- binary files store data as compactly as possible

Word-processor files  
are often binary files!

Advantages of text files:

- you can just look at them to see what's inside
- there are thousands of tools that work with them

Advantages of binary files:

- they typically use less space
- they are handled faster by the computer

We will look at binary  
files in part 2!

*Convenience/performance tradeoff*

*Example:* in 2 bytes, you can store integers  
up to  $2^{16} = 65\,536$  in a binary file,  
but only up to 99 in a text file.



# ASCII

Codes 20 to 7E (in decimal: 32 to 126) are standard characters

Codes 00 to 1F (in decimal: 0 to 31) are control characters,  
originally used in the communications protocol for teletype printers

Code 7F (in binary: 1111111) was used to erase data on punched  
tapes by overwriting.

**ASCII Code Chart**

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL



# ASCII CONTROL CODES

Most ASCII control codes were designed for teletypes and are obsolete.

Two control codes are frequent in text files:

- LF (line feed, code 10)
- CR (carriage return, code 13)

One or the other, or both (in the order CR, LF) are used to indicate the end of a line. Unix uses LF, MacOS (before MacOS X) used CR, DOS and Windows use CR+LF.

More rarely used control codes:

- BS (backspace, code 8)
- HT (horizontal tab, code 9)
- FF (feed forward, code 12)



# UNICODE

The Unicode standard aims at covering *all* the world's languages in a single file. This is not possible with just 256 values for a byte. Unicode currently defines 1 114 112 characters, each of which requires 21 bits of storage.

The first 127 codes are equal to ASCII.

Unicode is most commonly stored in files using UTF-8 encoding (Unicode Transformation Format using 8 bits). In UTF-8, the first 127 characters (ASCII) are stored as a single byte. Other characters are stored as a sequence of two or three bytes.

Every valid ASCII file is a valid UTF-8 file.

UTF-8 is on its way to becoming the most widely used character encoding system, in particular on the World-Wide Web.



# BINARY REPRESENTATIONS FOR NUMBERS

## Integers:

- each number is represented by N bits (N=8, 16, 32, 64)
- *unsigned* integers: range from 0 to  $2^N-1$
- *signed* integers: range from  $-2^{N-1}$  to  $2^{N-1}-1$

Order of the individual bytes in memory or in a file:

- highest-value byte first: “big-endian”
- lowest-value byte first: “little-endian”

bits	unsigned	signed
00	0	0
01	1	1
10	2	-2
11	3	-1



# BINARY REPRESENTATIONS FOR NUMBERS

## Floating-point numbers (IEEE 754):

- general form of a float:  $(-1)^s \cdot c \cdot b^q$
- *sign*  $s$ : 0 or 1 (one bit)
- *base*  $b$ : 2 or 10
- *significand*  $c$ : integer in the range 0 to  $b^p-1$
- *precision*  $p$ : positive integer
- *exponent*  $q$ : integer with range  $1-e_{\max} \leq q+p-1 \leq e_{\max}$

## Most common base-2 varieties:

- single-precision:  $p = 24$ ,  $e_{\max} = 127$ , stored in 4 bytes
- double-precision:  $p = 53$ ,  $e_{\max} = 1023$ , stored in 8 bytes

## Special values:

- NaN (not a number): result of undefined operations, e.g. logarithm of a negative number
- positive and negative infinity



# BINARY FILES

Traditionally:

- each program defines its own binary file formats, arranged for efficiency and the programmer's convenience
- binary formats tend to be undocumented
- some binary formats are platform-dependent (size of integers, byte order)
- data often becomes unusable over time, as platforms and programs evolve

Evolution towards standardized binary formats in science:

- platform-independent binary representations for integers (XDR) and floats (IEEE) since the 1980s
- standardized special-purpose formats (e.g. FITS)
- standardized general-purpose formats: CDF, netCDF, HDF



# NON-STANDARD BINARY FILES

**Avoid them if you can!**

If you can't:

- get the documentation if there is one
- get the source code of the program otherwise
- get a collection of example files
- use `od` to look at the examples, and check they agree with the documentation
- make sure you know what each bit means!
- find out how your favorite programming languages handles binary file I/O

Examples

- Python: use modules `struct` and `ctypes`
- C/C++: `fopen()` / `fread()` / `fwrite()` / `fclose()`
- Fortran: unformatted I/O



# FILE FORMATS



# WHAT'S A FILE FORMAT?

In theory: a precise definition of

- what data is stored
- how it is represented
- in which order items are stored

In practice: a more or less precise definition

Ideally, a file format should allow *validation*: running a program that decides if a given file implements the format correctly or not. Validation requires a precise definition of the file format.

Good file formats are important for:

- exchanging data between different programs
- archiving data for long-term use



# WHAT'S A DATA MODEL?

A definition of how complex data is represented in terms of basic data (numbers, text, tables, ...). A data model can be realized in many different concrete file formats.

Example: the PDB data model

- a list of atoms with for each atom
  - ♦ a residue name and residue number
  - ♦ an atom name
  - ♦ the chemical element symbol
  - ♦ a position (x, y, z coordinates)
  - ♦ the uncertainty of the position (B-factor or ADP)
- nature of the molecules
- experimental conditions
- ...

File formats: PDB, mmCIF, PDBML



# PROPRIETARY FILE FORMATS

Proprietary file formats are undocumented file formats used exclusively by a set of programs and inaccessible to others.

Some vendors use proprietary formats out of laziness, but usually the goal is to force you to use their products exclusively.

*Don't use proprietary file formats for scientific data.*

It's you who is responsible for your data, so you must be able to control it to the last bit if necessary.



# MARKUP LANGUAGES

Convention for indicating *semantic* or *presentation* features in human-readable text.

Examples: HTML, TeX, Markdown, reST, ...

Application: text that is processed by computer programs

Examples: Web pages, scientific articles, software documentation, comments in scientific data sets, ...



# FULLY STRUCTURED DATA

Fully structured data has a completely predefined structure. The structure is defined by the file format, the file contains only the data.

Examples: tables, arrays, database records, ...

Example file formats: CSV, PDB, FASTA

Fully structured data is the easiest to handle in computer programs, but also the least flexible.



# SEMI-STRUCTURED DATA

Semi-structured data *contains* tags that define its own structure. There is a general convention that specifies how to represent basic data items (numbers, text, ...), and for each format a specific convention that specifies the valid tags and the rules for combining tagged items.

General conventions: XML, SGML, s-expressions,  
JSON, ...

Specific file formats: PDBML (XML), HTML (SGML),  
Lisp (s-expressions), ...

Semi-structured data is much more flexible than fully structured data and allows in particular future extensions. Most recently defined file formats are based on semi-structured formats, in particular XML.



# XML: EXTENSIBLE MARKUP LANGUAGE

Published in 1996 as a simplified version of SGML (Standard Generalized Markup Language). Became rapidly popular for Web technologies.

An XML file consists of Unicode characters.

Tags are surrounded by < > and can contain attributes.

Elements are delimited by a start tag and a matching end tag:

```
<pdb-id> 1IEE </pdb-id>
```

Empty elements use special tags:

```

```

A specific XML-based format is defined by a *Document Type Definition* (DTD) or a *Schema* in order to allow validation.

Scientific XML formats: PDBML (PDB), CML (chemistry),  
SBML (systems biology), MathML (maths),  
XSIL (tables and arrays), ...



FITS



# FITS IN PYTHON

- Formerly PyFITS, now part of astropy
- Tutorial at  
<http://www.astropython.org/tutorial/2010/10/PyFITS-FITS-files-in-Python>



# HDF5

(HIERARCHICAL DATA FORMAT 5)



# HDF5 OVERVIEW

## Data model: **arrays and tables in a file**

- Widely used in different communities
- Basis of other formats (NeXus, netCDF4)
- Works on all current computing platforms
- Portable data files of unlimited size
- Many tools available
- Library written in C
- Bindings for C++, Fortran 90, and Java
- Supports parallel I/O

<http://www.hdfgroup.org/HDF5/>

Third-party bindings:

Python (h5py, PyTables), Matlab, IDL, Mathematica



# STRUCTURE OF A HDF5 FILE

**Dataset:** multidimensional array of data elements plus metadata (name, description, time stamps, ...)

**Group:** a named grouping of datasets and other groups

**Node:** a group or a dataset

**HDF5 file:** contains the “root group” of a group hierarchy

**Analogy to a file system:** datasets  $\leftrightarrow$  files, groups  $\leftrightarrow$  directories.

**Path notation** to specify groups and datasets:

/group1/group2/dataset



# DATASETS

**Name:** any text string

**Datatype:**

- *atomic*: integer, float, string of various sizes
- *compound*:
  - multidimensional array, elements of any datatype
  - variable length 1D array, elements of any datatype
  - record with named fields of any datatype

Compounds can be recursive: a record with a field that is an array of records whose fields are integers...

**Dataspace:** describes the arrangement of elements

- *scalar*: one element
- *simple*: multidimensional array of elements

**Storage layout:** contiguous, chunked, compact



# ATTRIBUTES

**Small datasets** attached to **primary datasets** or to **groups**.  
Typically used to store **metadata**.

**Name:** any text string

**Datatype:** as for datasets

**Dataspace:** as for datasets

**Storage layout:** always compact

Restrictions:

- can't have attributes
- no partial I/O



# HDF5 TOOLS

## Command line tools:

- **h5ls:** show names of groups and datasets in an HDF5 file
- **h5dump:** show contents of datasets
- **h5diff:** compare datasets and files
- **h5import:** import data from text and platform-specific binary files into an HDF5 file

**GUI browser and editor:** hdfview



# HDF5 LIBRARY

## Sublibraries by function category:

- H5: general library functions
- H5F: files
- H5D: datasets
- H5T: datatypes
- H5S: dataspace
- H5A: attributes

plus many more!

## OO-like structure:

- **open** (file, dataset, datatype, ...) creates an in-memory data structure and returns a unique handle to it
- **close** deallocates the data structure and makes the handle invalid



# HDF5 DOCUMENTATION

## **Introduction:**

<http://www.hdfgroup.org/HDF5/doc/H5.intro.html>

## **User's Guide:** (more detailed)

<http://www.hdfgroup.org/HDF5/doc/UG/>

## **Function reference:**

[http://www.hdfgroup.org/HDF5/doc/RM/RM\\_H5Front.html](http://www.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html)

## **Examples:**

<http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/>



# HDF5 IN PYTHON

**PyTables:**

<http://www.pytables.org/>

database of tables stored in HDF5 files

**h5py:**

<http://h5py.org/>

closer to the HDF5 C/Fortran interface

arrays in a file



# HDF5 IN PYTHON: WRITING

```
import h5py
import numpy as np

h5file = h5py.File('test.h5', 'w')
h5file.attrs['version'] = 42

foo = h5file.create_dataset('foo', (20, 3), dtype=np.float64)
foo[:, :] = 0.
foo[0, :] = [42., 42., 42.]
foo[:, 1] = 1.
foo.attrs['units'] = "arbitrary"

bar = h5file.create_dataset('bar', (0, 20), dtype=np.int32,
                             chunks=(10, 10), maxshape=(None, 20))
for i in range(10):
    bar.resize((i+1, 20))
    bar[i, :] = i*np.ones((1, 20), np.int)

h5file.close()
```



# HDF5 IN PYTHON: READING

```
import h5py
import numpy as np

h5file = h5py.File('test.h5', 'r')

for item in h5file:
    print item

foo = h5file['foo'][...]
foo_units = h5file['foo'].attrs['units']
print foo[0]

h5file.close()
```



# WRITING ARRAYS

```
import tables as T
import numpy as N
```

```
h5file = T.openFile('toto.h5', 'w', 'My first HDF5 file')
```

```
my_arrays = h5file.createGroup(h5file.root, 'my_arrays',  
                                'all my arrays')
```

create a group

```
prime_numbers = N.array([2, 3, 5, 7, 11, 13])
h5file.createArray(my_arrays, 'prime_numbers',  
                  prime_numbers,  
                  'some prime numbers')
```

group and name of array  
array data  
title (optional)

```
big_array = h5file.createCArray(my_arrays, 'big_array',  
                                T.Float32Atom(),  
                                (1000, 3, 10),  
                                'a big array')
```

group and name of array  
element type  
shape  
title (optional)

```
big_array[:, :, :5] = 0.  
big_array[:, :, 5:] = 1.
```

set values

```
h5file.close()
```



# WRITING TABLES

```
import tables as T
```

```
h5file = T.openFile('toto.h5', 'a')
```

```
my_tables = h5file.createGroup(h5file.root, 'my_tables',  
                                'all my tables')
```

create a group

```
class AtomicMasses(T.IsDescription):
```

```
    element = T.StringCol(2)
```

column template definition

```
    mass = T.Float32Col()
```

```
atomic_masses = h5file.createTable(my_tables, 'atomic_masses',  
                                   AtomicMasses,  
                                   'a table of atomic masses')
```

group and name  
column template  
title (optional)

```
row = atomic_masses.row
```

get row object for the table

```
mass_table = {'H': 1., 'C': 12., 'O': 16.}
```

```
for e, m in mass_table.items():
```

```
    row['element'] = e
```

```
    row['mass'] = m
```

set column values

```
    row.append()
```

append row to table

```
atomic_masses.flush()
```

write table data to file

```
h5file.close()
```



# HDF5 IN C: WRITING DATA

```
#include "hdf5.h"
```

```
#define FILE          "demo.h5"
```

```
#define DATASETNAME "IntArray"
```

```
#define NX          5
```

```
#define NY          6
```

```
#define RANK        2
```

```
int main (void) {  
    hid_t      file, dataset;  
    hid_t      datatype, dataspace;  
    hsize_t     dims[2];  
    herr_t      status;  
    int         data[NX][NY];  
    int         i, j;  
  
    for (j = 0; j < NX; j++) {  
        for (i = 0; i < NY; i++)  
            data[j][i] = i + j;  
    }
```

```
    file = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
```

```
    dims[0] = NX;
```

```
    dims[1] = NY;
```

```
    dataspace = H5Screate_simple(RANK, dims, NULL);
```

```
    datatype = H5Tcopy(H5T_NATIVE_INT);
```

```
    status = H5Tset_order(datatype, H5T_ORDER_LE);
```

```
    dataset = H5Dcreate(file, DATASETNAME, datatype, dataspace,  
                        H5P_DEFAULT);
```

```
    status = H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL,  
                      H5P_DEFAULT, data);
```

```
    H5Sclose(dataspace);
```

```
    H5Tclose(datatype);
```

```
    H5Dclose(dataset);
```

```
    H5Fclose(file);
```

```
    return 0;
```

```
}
```



# HDF5 IN FORTRAN 90: WRITING DATA

```
PROGRAM main
```

```
USE HDF5
```

```
IMPLICIT NONE
```

```
CHARACTER(LEN=14), PARAMETER :: filename = "demo.h5"
```

```
CHARACTER(LEN=3) , PARAMETER :: dataset = "DS1"
```

```
INTEGER          , PARAMETER :: dim0      = 4
```

```
INTEGER          , PARAMETER :: dim1      = 7
```

```
INTEGER :: hdferr
```

```
INTEGER(HID_T) :: file, space, dset
```

```
INTEGER(HSIZE_T), DIMENSION(1:2)          :: dims = (/dim0, dim1/)
```

```
INTEGER          , DIMENSION(1:dim0,1:dim1) :: wdata
```

```
INTEGER :: i, j
```

```
CALL h5open_f(hdferr)
```

```
DO i = 1, dim0
```

```
    DO j = 1, dim1
```

```
        wdata(i,j) = (i-1)*(j-1)-(j-1)
```

```
    ENDDO
```

```
ENDDO
```

```
CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F,  
                  file, hdferr)
```

```
CALL h5screate_simple_f(2, dims, space, hdferr)
```

```
CALL h5dcreate_f(file, dataset, H5T_STD_I32LE,  
                 space, dset, hdferr)
```

```
CALL h5dwrite_f(dset, H5T_NATIVE_INTEGER,  
               wdata, dims, hdferr)
```

```
CALL h5dclose_f(dset , hdferr)
```

```
CALL h5sclose_f(space, hdferr)
```

```
CALL h5fclose_f(file , hdferr)
```

```
END PROGRAM main
```



# EXERCISES



# EXPLORING A NEXUS FILE

The NeXus file format is a set of conventions for storing experimental data (X-ray diffraction and neutron scattering experiments) in a HDF5 file.

- 1) Inspect the example file `Co_edge_CnrX_2008-09-17_22-12-12.nxs` using HDFView and/or the HDF5 command line tools.  
(Thanks to the Proxima 1 team at Synchrotron SOLEIL for providing this file!)
- 2) Which user recorded the data? Find his name and telephone number.
- 3) Plot the `fitted_escan_curves: raw_data` vs. `x-ray_energy`, using whatever programs you are familiar with.



# EXPLORING METEOROLOGICAL DATA

Inspect the example file

`OMI.L2.CloudOMCLDO2Strip200kmAlongCloudSat.2011.06.22.050738Z.v003.he5`

(that's the original name as obtained from the NASA Earth Observatory!)

What can you say about the data in that file with no additional information?



# HDF5 IN PYTHON

Write a Python script to explore the contents of the NeXus example file.  
Make it print out as much useful information as you can find.