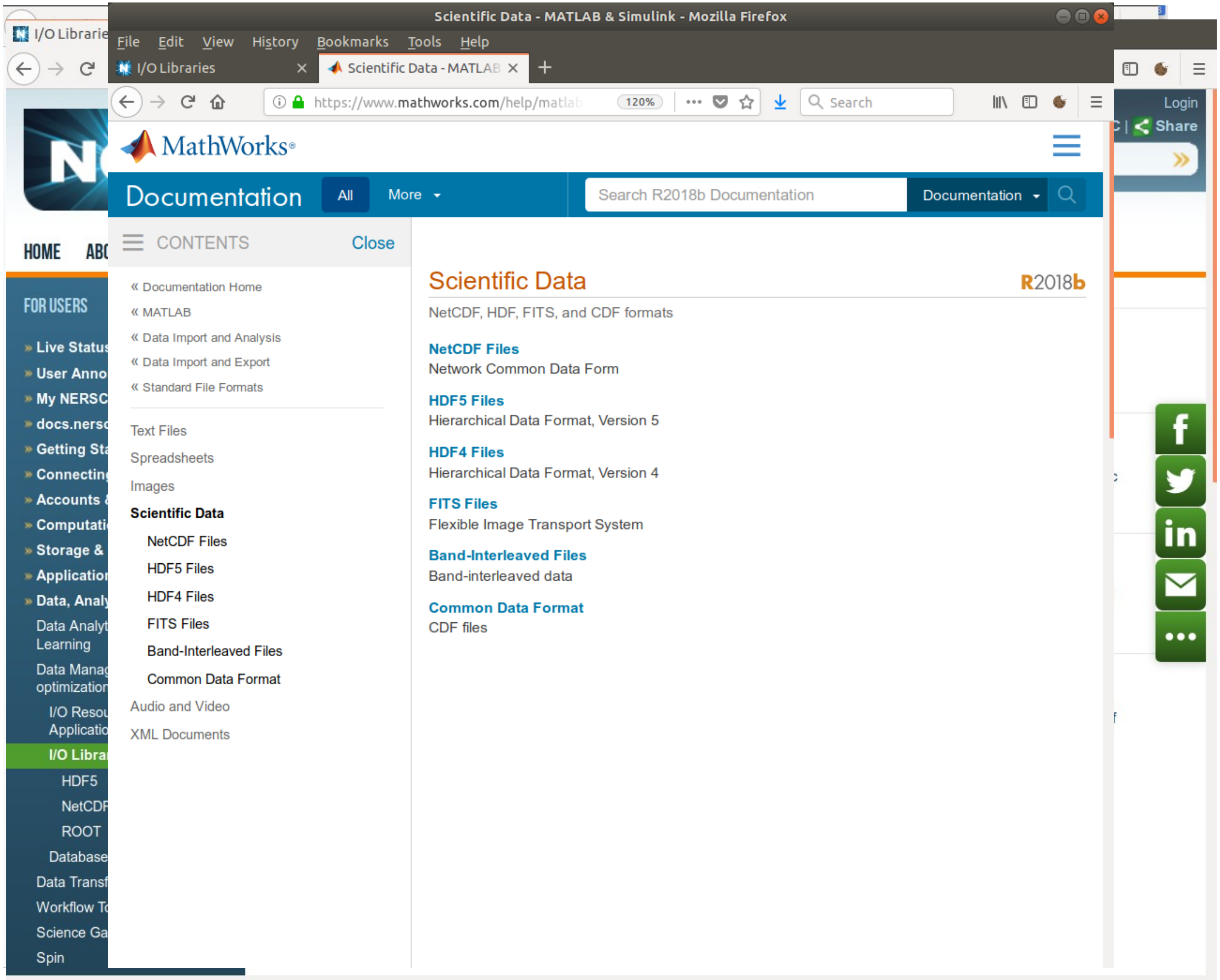
Lecture 10 – Scientific data formats

Open Data Management & the Cloud

(Data Science & Scientific Computing / UniTS – DMG)

Scientific data formats



Scientific I/O goals

- I/O is commonly used by scientific applications to achieve **goals** like
  - **storing** numerical output from simulations for later analysis
  - implementing 'out-of-core' techniques for algorithms that process **more data than can fit in system memory** and must page data in from disk
  - **checkpointing** to files that save the state of an application in case of system failure
- Provide a digital archival format **portable** and **self-describing**, on the assumption that neither the software nor the hardware that wrote the data will be available when the data are read
  - To be supported by an open format specification
  - Application programming interface available for several programming languages (C, C++, Java, Python, R, Fortran, Julia, Ruby, etc.) and on different operating systems and hardware architectures.

Data formats adoption

- **HDF5**
  - used in several research areas, including earth sciences, computational fluid dynamics, astronomy, astrophysics, but also financial services and industry
- **NetCDF** is a set of interfaces for array-oriented data access. Starting with version 4, the netCDF library can use HDF5 files as its base format
  - Used in climatology, meteorology and oceanography applications (e.g., weather forecasting, climate change) and GIS applications
- **FITS** is the standard data format used in Astronomy
  - ESA and NASA developed FITS in the late 1970s, stemming from radio astronomy (FITS is always backward compatible)
  - The Vatican Library has adopted the FITS data format for the long-term digital preservation of the books, manuscripts, and other objects in its vast collection
- **ROOT**
  - Originally designed for particle physics (at CERN), its usage has extended to other data-intensive fields like astrophysics and neuroscience

File formats features

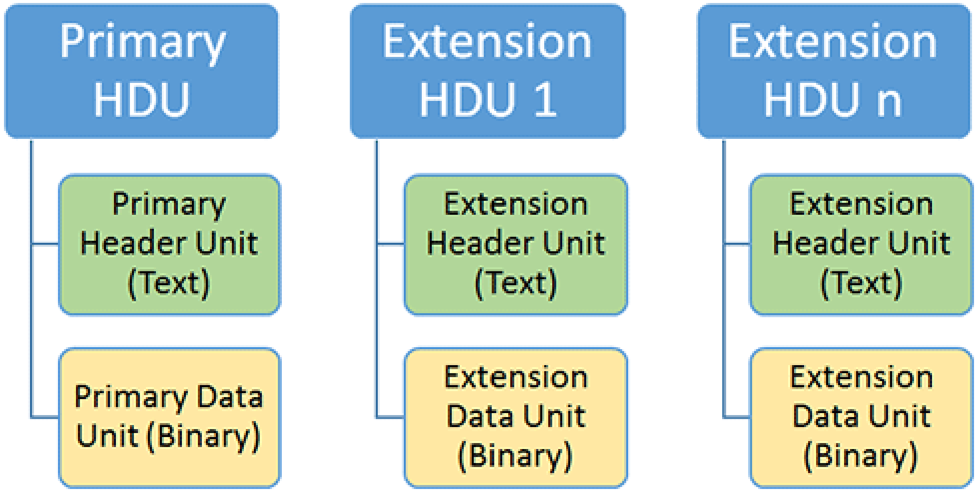
- Self-description (i.e. metadata)
  - Human-readable metadata availability
- Open-format, i.e. with a public specification maintained by a standards organization
- Machine independence
- Storage efficiency
- Data structures: images, n-dimensional arrays, tables, objects sequences, hierarchical structures
- Internal data compression (e.g. tile compression)
- Data access
  - read/write a portion of the n-dimensional arrays (hyperslabs) or tables

FITS format

- Even if mainly used in Astronomy, it is useful to start with a quick view of the FITS standard, in order to highlight some concepts and data structures
- The first **FITS** (**Flexible Image Transport System**) standard was published in 1981. The most recent version (4.0) has been standardized in 2016
  - Ref: https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf
- It is primarily designed to store scientific data sets consisting of **multidimensional arrays** (images) and **2-dimensional tables** organized into rows and columns of information
- In few words a FITS file is composed by **two distinct parts**, which can be repeated **several times**:
  - the first part (**header**) is formed by easily viewable ASCII text elements providing metadata information
  - in the second part there are the data in **binary format** (a multi-dimensional array or a table)

The FITS HDU

- The header and the binary part together are called Header Data Unit (**HDU**)
  - The binary part (data unit) is always optional
  - The first HDU is called **primary HDU** or primary array and its binary part can only be an image (n-dimensional array)
  - Any number of additional HDUs may follow the primary array. These additional HDUs are referred to as FITS ‘extensions’
  - The binary part of a fits extension can contain either an n-dimensional array or a table
    - To be precise, the data unit can also contain an ASCII table, so it is not always binary



FITS header example

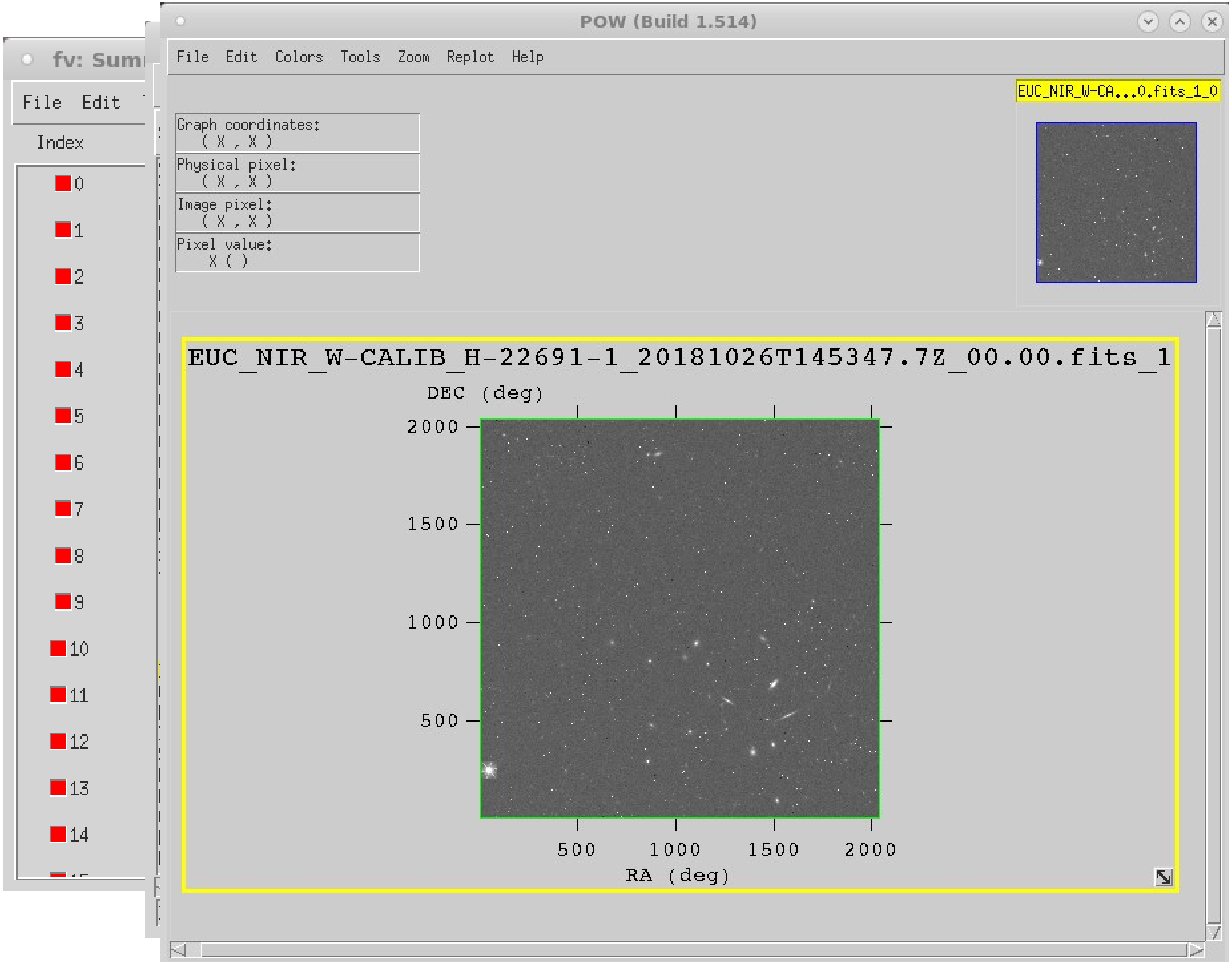
fv: Header of EUC_NIR_W-CALIB_H-22691-1_20181026T145347.7Z_00.00.fi

File Edit Tools Help

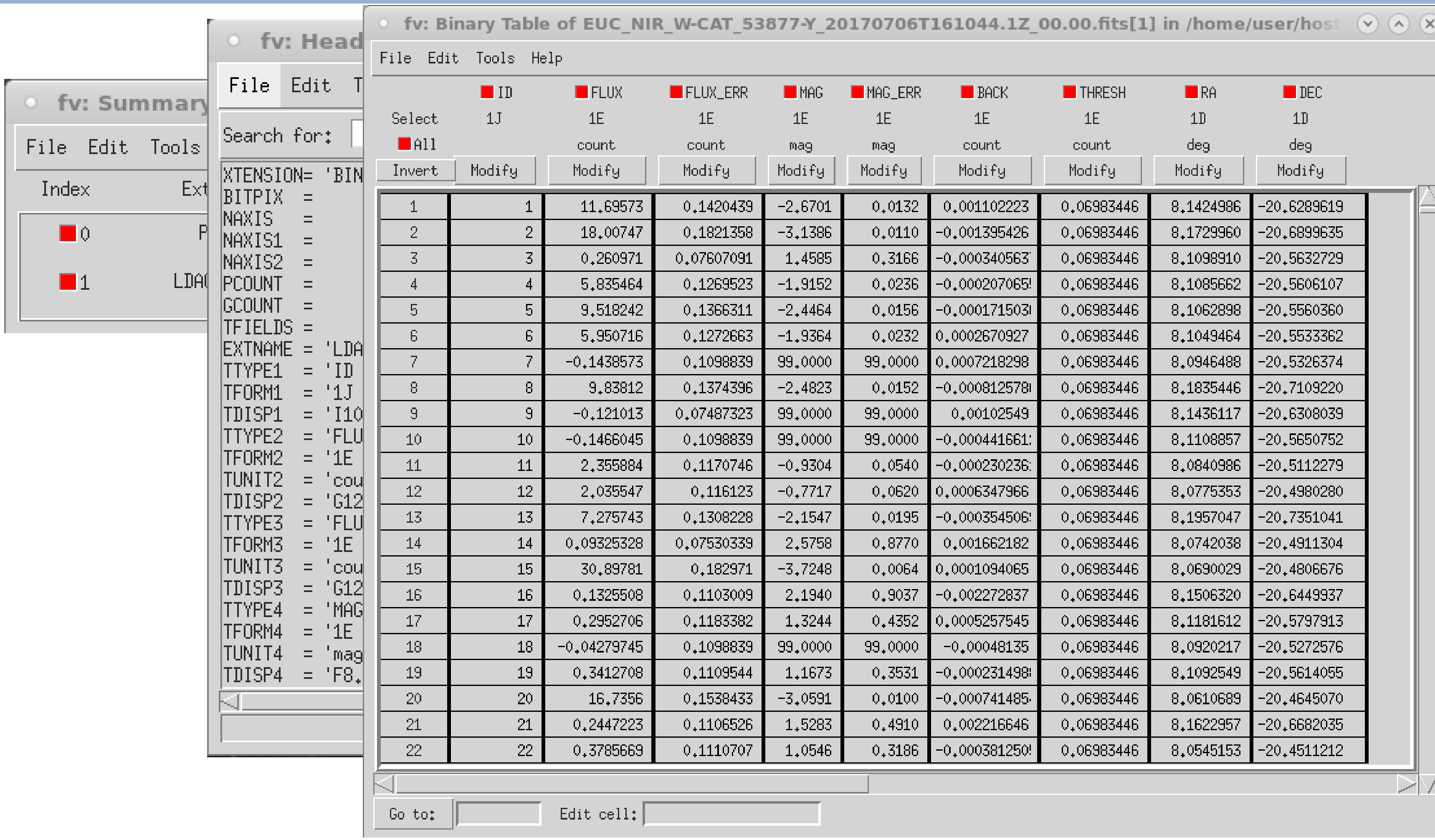
Search for: Find Case sensitive? No

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                    8 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
EXTEND  =                    T / FITS dataset may contain extensions
COMMENT   FITS (Flexible Image Transport System) format is defined in 'Astronomy
COMMENT   and Astrophysics', volume 376, page 359; bibcode: 2001A&A...376..359H
TELESCOP= 'EUCLID  '
INSTRUME= 'NISP    '
VERSION = 'SC456-NIP-C7a_T2'
DATE    = '2018-10-17T15:18:43.456Z'
ORIGIN  = 'OU-SIM  '
PROCVER = '3.0     '
DATE-OBS= '2025-06-28T17:47:16.000Z'
DATE-UTC= '2025-06-28T17:47:16.000001+00:00'
MJD-OBS =     9310.74193500207
EXPTIME =                  66.
RA      =     10.8434101026019
DEC     =    -18.5875042774849
LEDID   =                    0
LEDLVL  =                   0.
PA      =     65.2070574970681
EQUINOX =                2000.
RADECSYS= 'FK5     '
OBSID   =                22691
DITHOBS =                    1
PTGID   =                22691
EXPNUM  =                    5
TOTEXP  =                    4
FILTER  = 'H       '
GRISM   = 'OPEN    '
IMG_CAT = 'SCIENCE '
IMG_T1  = 'OBJECT  '
IMG_T2  = 'SKY     '
OBSMODE = 'WIDE    '
OBSTYPE = 'IMAGE   '
READMODE= 'UpTheRamp'
NG      =                    3
NR      =                   16
ND      =                    4
FRTIME  =                 1.41
FLTCORR = 'True    '           / True => flat field Corrected, False => no
CONTINUE  'terflat_H.xml'      / calibration file used
PHRELEX =                   1. / relative photometric offset for exposure
PHRELEXE=    0.101118742080783 / error in PHRELEX determination
```

Euclid example: NISP detectors in FITS



FITS binary table



Select	ID	FLUX	FLUX_ERR	MAG	MAG_ERR	BACK	THRESH	RA	DEC
All	1J	1E	1E	1E	1E	1E	1E	1D	1D
		count	count	mag	mag	count	count	deg	deg
Invert	Modify	Modify	Modify	Modify	Modify	Modify	Modify	Modify	Modify
1	1	11.69573	0.1420439	-2.6701	0.0132	0.001102223	0.06983446	8.1424986	-20.6289619
2	2	18.00747	0.1821358	-3.1386	0.0110	-0.001395426	0.06983446	8.1729960	-20.6899635
3	3	0.260971	0.07607091	1.4585	0.3166	-0.000340563	0.06983446	8.1098910	-20.5632729
4	4	5.835464	0.1269523	-1.9152	0.0236	-0.000207065	0.06983446	8.1085662	-20.5606107
5	5	9.518242	0.1366311	-2.4464	0.0156	-0.000171503	0.06983446	8.1062898	-20.5560360
6	6	5.950716	0.1272663	-1.9364	0.0232	0.0002670927	0.06983446	8.1049464	-20.5533362
7	7	-0.1438573	0.1098839	99.0000	99.0000	0.0007218298	0.06983446	8.0946488	-20.5326374
8	8	9.83812	0.1374396	-2.4823	0.0152	-0.000812578	0.06983446	8.1835446	-20.7109220
9	9	-0.121013	0.07487323	99.0000	99.0000	0.00102549	0.06983446	8.1436117	-20.6308039
10	10	-0.1466045	0.1098839	99.0000	99.0000	-0.000441661	0.06983446	8.1108857	-20.5650752
11	11	2.355884	0.1170746	-0.9304	0.0540	-0.000230236	0.06983446	8.0840986	-20.5112279
12	12	2.035547	0.116123	-0.7717	0.0620	0.0006347966	0.06983446	8.0775353	-20.4980280
13	13	7.275743	0.1308228	-2.1547	0.0195	-0.000354506	0.06983446	8.1957047	-20.7351041
14	14	0.09325328	0.07530339	2.5758	0.8770	0.001662182	0.06983446	8.0742038	-20.4911304
15	15	30.89781	0.182971	-3.7248	0.0064	0.0001094065	0.06983446	8.0690029	-20.4806676
16	16	0.1325508	0.1103009	2.1940	0.9037	-0.002272837	0.06983446	8.1506320	-20.6449937
17	17	0.2952706	0.1183382	1.3244	0.4352	0.0005257545	0.06983446	8.1181612	-20.5797913
18	18	-0.04279745	0.1098839	99.0000	99.0000	-0.00048135	0.06983446	8.0920217	-20.5272576
19	19	0.3412708	0.1109544	1.1673	0.3531	-0.000231498	0.06983446	8.1092549	-20.5614055
20	20	16.7356	0.1538433	-3.0591	0.0100	-0.000741485	0.06983446	8.0610689	-20.4645070
21	21	0.2447223	0.1106526	1.5283	0.4910	0.002216646	0.06983446	8.1622957	-20.6682035
22	22	0.3785669	0.1110707	1.0546	0.3186	-0.000381250	0.06983446	8.0545153	-20.4511212

- The header of a binary table specifies also each column name, its type and the unit of measurement
- Cells can also contain fixed or variable length arrays

FITS metadata and data

- FITS keywords are defined by a **keyword name**, a **value** (string, logical, int, float, complex) and an **optional comment**
  - The comment is used to further document the metadata information, e.g. indicating the unit of measure and purpose or, for date time values, the epoch used
  - Keyword names are limited to 8 characters, but a widely used standard extension allows longer names
- The FITS standard also fixes a dictionary of keyword names and corresponding value type and format for representation of World Coordinate Systems and time coordinates
- Additional dictionaries are defined by astronomy organizations such as the European Southern Observatory (ESO) and the National Optical Astronomy Observatory (NOAO)

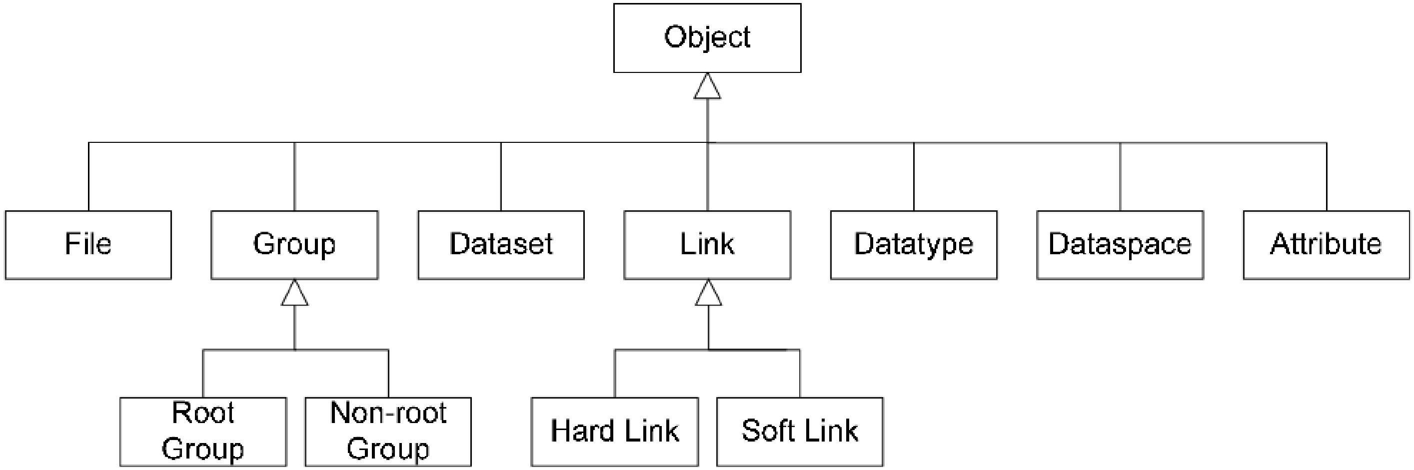
FITS Keyword Dictionaries

The following data dictionaries contain compilations of the FITS header keywords that have been defined and used within various contexts.

- Keywords defined in the FITS Standard
- Other commonly used keywords
- UCO/Lick keyword dictionary
- STScI keyword dictionary
- NOAO keyword dictionary
- ESO keyword dictionary

HDF5 data model

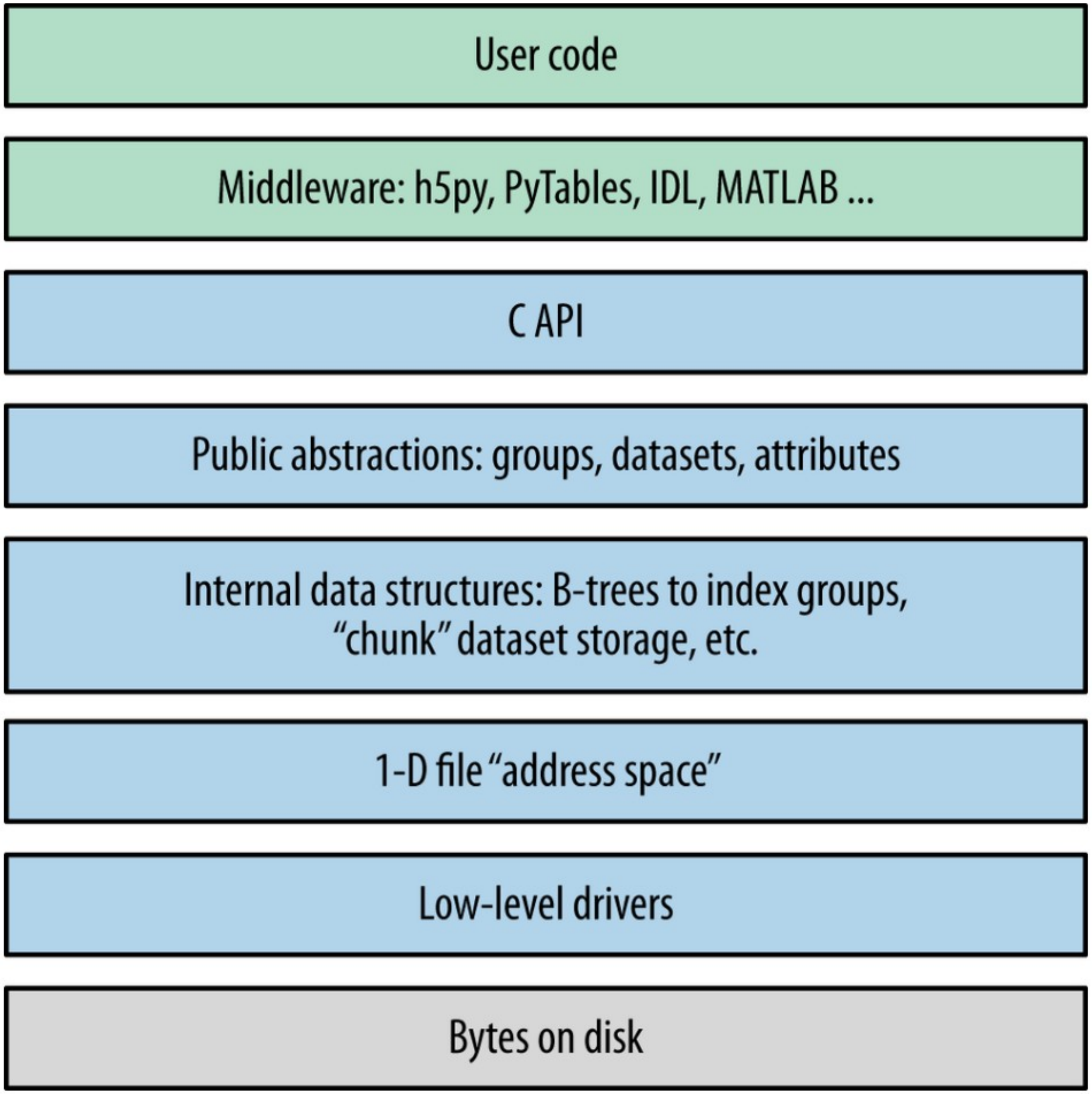
- The Heararchical Data Format (HDF5) data model defines 7 classes of objects:
  - A **file** is a container for HDF5 objects. Default file storage layout: single, contiguous file on local disk
    - Alternative layouts are designed to suit the needs of a variety of systems
  - A **dataset** contains an array of data elements, together with supporting metadata
    - **Dataspaces** describe the rank and dimensions of a data object array.
    - **Datatypes** describe the data elements in a data object array



HDF5 data model and library

- **Groups** and **links** are used to organize objects in a file as a **directed graph** with a single designated entry node, called the **root group**
  - In other words, groups are hierarchical containers that store datasets and other groups
- An **attribute** is a means of attaching content metadata to an object (i.e. datasets and groups)
- The HDF5 file specification and open source library is maintained by the HDF Group
- The HDF Group's primary product is the HDF5 software library, written in C, with additional bindings for C++ and Java
  - The python interfaces, e.g. h5py and PyTables, are designed to use the C library

HDF5 library



HDF5 and Python

- The HDF Group provides a software library in C, C++, Fortran and Java
- It also provides a graphical viewer for HDF5 files, named **HDFView**, and some command line tools:
  - **h5ls**: lists the metadata content of an HDF5 file
  - **h5dump**: prints both metadata and data content of an HDF5 file
- One of the Python modules available for read and write HDF5 files is **h5py**. We will use this module in the following examples
- The easiest way to install the HDF5 libraries and python module is again the Anaconda python distribution, which installs them by default:

  https://www.anaconda.com/download
- Example project available at: https://www.ict.inaf.it/gitlab/odmc/hdf5_example

HDF5 datasets

- The **Datasets** are the central feature of HDF5. We can consider them as multi-dimensional arrays that live on disk
- Every dataset in HDF5 has a name, a type, a shape, and supports random access
- When using the h5py python module, the datasets API is close to the standard python n-dimensional array module, **numpy**

```
import h5py
import numpy as np

f = h5py.File("testdata.hdf5","w")

# Empty dataset creation: dataset name, shape and type
f.create_dataset("test1", (20,15), dtype=np.float32)
# The dataset is filled with zero by default

# We can also pass another fill value
f.create_dataset("test2", (25,), dtype=np.int32, fillvalue=42)

# Or we can pass directly the data array as a numpy array
bigdata = np.ones((100, 1000), dtype=np.float64)
f.create_dataset("test3", data=bigdata, dtype=np.float32)
```

Casting to a 32 bit floating point to save space on disk

Datasets indexing and boolean indexing

- Datasets permit slicing operations analogous to numpy arrays
- However, for performance reasons, the dataset should be accessed by blocks of values instead of single or few values
- If you need to access repeatedly few values at a time, it is better to retrieve an entire dataset or at least a block, so that it is returned as a numpy array in memory, and then access such numpy array

```
# random 2d distribution in the range (-1,1)
data = np.random.rand(15, 10)*2 - 1

dset = f.create_dataset('random', data=data)

# print the first 5 even rows and the first two columns
out = dset[0:10:2, :2]
print(out)

# clipping to zero all negative values
dset[data<0] = 0
```

- But also avoid explicit loops in python on huge arrays

Appending new data

- Until now, we have created datasets with a fixed shape
- However, often we don't know in advance the size of a dataset and we need to append new data to it
- First we have to create a resizable dataset, then we have to append data in a scalable way
  - Datasets, by default, store data in row-major order

```
dset = f.create_dataset('dataset_two', (1,1000), dtype=np.float32,
                        maxshape=(None, 1000))

a = np.ones((1000,1000))

num_rows = dset.shape[0]
dset.resize((num_rows+a.shape[0], 1000))

dset[num_rows:] = a
```

First axis is now unlimited

We resize the dataset before performing the bulk insertion

HDF5 Groups

- **Groups** are the HDF5 **container object**, analogous to folders in a filesystem
- They can hold datasets and other groups, allowing you to build up a **hierarchical structure** with objects neatly organized in groups and subgroups
- The **File object** is itself a group. In this case, it also serves as the **root group**, **named /**, our entry point into the file
- Groups work mostly like dictionaries; groups are iterable, and have a subset of the normal Python dictionary API

```
grp = f.create_group('nisp_frame/detectors/det11')
grp['sci_image'] = np.zeros((2040,2040))

print(grp.name)      # the group name property
print(grp.parent)    # the parent group property
print(grp.file)      # the file property
print(grp)           # prints some group information.
```

output

```
/nisp_frame/detectors/det11
<HDF5 group "/nisp_frame/detectors" (1 members)>
<HDF5 file "testdata.hdf5" (mode r+)>
<HDF5 group "/nisp_frame/detectors/det11" (1 members)>
```

HDF5 attributes

- **Attributes** are pieces of metadata you can stick on objects in the file. They're a key mechanism for making self-describing files.
- You can attach attributes to any kind of object that is linked into the HDF5 tree structure: groups, datasets and other objects not mentioned in this introduction
- Both groups and datasets provide a "**`.attrs`**" property in h5py. This is a little proxy object that works mostly like a Python dictionary

```
grp = f['nisp_frame']
grp.attrs['telescope'] = 'Euclid'
grp.attrs['instrument'] = 'NISP'
grp.attrs['pointing'] = np.array([8.48223045516, -20.4610801911, 64.8793517547])
grp.attrs.create('detector_id', '11', dtype="|S2")

print(grp.attrs['pointing'])
print(grp.attrs['detector_id'])
```

output

```
[  8.48223046 -20.46108019  64.87935175]
b'11'
```

HDF5 types

Native HDF5 type	NumPy equivalent
Integer	`dtype("i")`
Float	`dtype("f")`
Strings (fixed width)	`dtype("S10")`
Strings (variable width)	`h5py.special_dtype(vlen=bytes)`
Compound	`dtype([ ("field1": "i"), ("field2": "f") ])`
Enum	`h5py.special_dtype(enum=("i",{"RED":0, "GREEN":1, "BLUE":2}))`
Array	`dtype("(2,2)f")`
Opaque	`dtype("V10")`
Reference	`h5py.special_dtype(ref=h5py.Reference)`

DEPRECATED

HDF5 special types

- HDF5 supports a few types which have no direct NumPy equivalent. Among the most useful and widely used are:
  - Variable length (VL) types: variable length strings, “ragged” arrays
  - Enumerated types
- Before version 2.10 of h5py the API was providing h5py.spedial_dtype(**kwds) function, now deprecated
- Now h5py provides dedicated functions

```
# Variable length strings
dt = h5py.string_dtype(encoding='utf-8')
ds = f.create_dataset('VLDS', (100,100), dtype=dt)
# Ragged arrays of integers
dt = h5py.vlen_dtype(np.dtype('int32'))
dset = f.create_dataset('vlen_int', (100,), dtype=dt)
dset[0] = [1,2,3]
dset[1] = [1,2,3,4,5]
# Enum types
dt = h5py.enum_dtype({"RED": 0, "GREEN": 1, "BLUE": 42}, basetype='i')
```

Tables = Datasets and compound types

- Table can be stored using datasets and the compound types (see below)
- NumPy supports this feature through structured arrays. The dtype for these arrays contains a series of fields, each of which has a name and its own sub-dtype

```
f = h5py.File("testdata.hdf5",”w”)

dt = np.dtype([('source_id', np.uint32), ('ra', np.float32), ('dec', np.float32),
               ('magnitude', np.float64)])
grp = f.create_group('source_catalog/det11')
dset = grp.create_dataset('star_catalog', (100,), dtype=dt)

dset['source_id', 0] = 1
print(dset['source_id', 'ra', :20])
print(dset[0])
```

Compound type

HDF5 object references

- Additional useful features in HDF5 are those that help you to express relationships between pieces of your data
- For instance, we may want to relate a dataset containing a catalog of sources with the image where the catalog was extracted
- Or, given a specific astronomical source, we may want to quickly find the cutout (region) of the source in the original image
- In HDF5, an **object reference** is basically a pointer to object in the file
- A reference to an object, e.g. a group or a dataset, can be obtained through its '`.ref`' property, which in h5py as type **h5py.Reference**
- Since the reference is an "absolute" way of locating an object, you can use any group in the file for dereferencing it, not just the root group
- Object references can be stored as data, and they're independent of later renaming of the objects involved (almost unbreakable links)

References and Region References

- References are full-fledged types in HDF5; we can use them in both attributes and datasets

```
sci_image = f['/nisp_frame/detectors/det11/sci_image']
sci_image.attrs['star_catalog'] = dset.ref
cat_ref = sci_image.attrs['star_catalog']

print(cat_ref)
dset = f[cat_ref]
print(dset[0])
```

- **Region references** let you store a reference to part of a dataset, e.g. a region of interest (ROI) on images stored in an HDF5 file
  - Datasets provide a property named '**`.regionref`**', to create a region reference by applying the standard NumPy slicing syntax to the object

```
roi = sci_image.regionref[15:20, 36:78]
print(sci_image[roi])
```

Chunked storage 1/2

- By default, all but the smallest HDF5 datasets use contiguous storage
- Applications reading a whole image, or a series of whole images, will be efficient at reading the data
- But suppose that we have a sequence of images of the same size, e.g. 100 images of 2048x2048 pixels, and that we have to compute the median of each pixel along the sequence of images
  - We can process small blocks of 64x64 pixels for each image in the sequence
  - For each image in the sequence we could start reading data in a 64×64 pixel slice in the corner of the first image

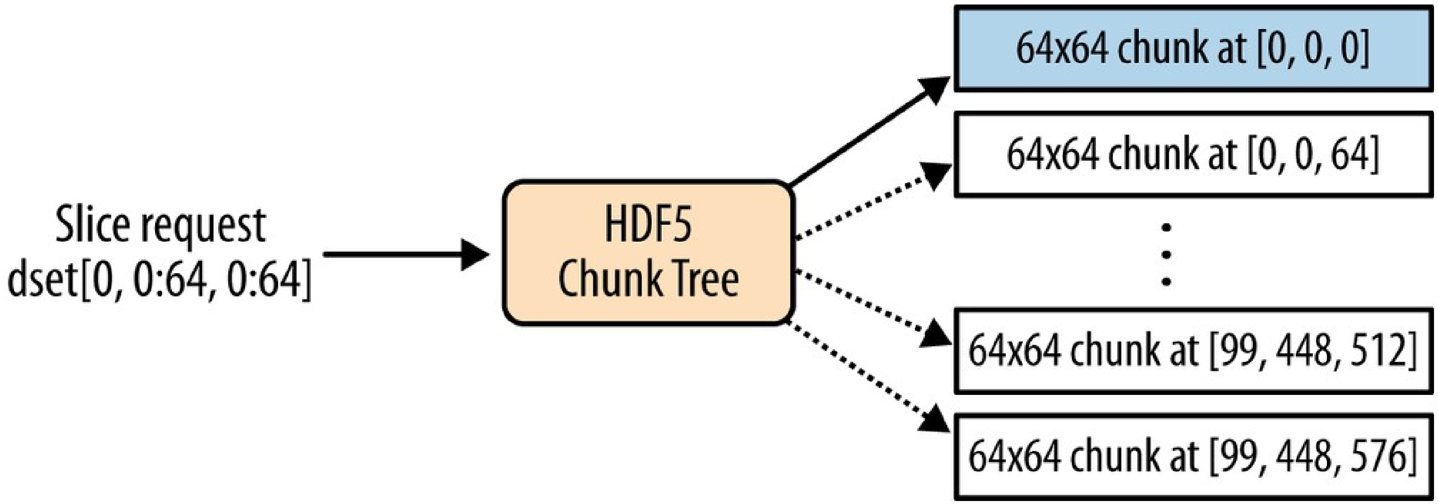
    ```
    dset[0, 0:64, 0:64]
    ```

    and then proceed on the same block for the other images
  - The fundamental problem here is that the default contiguous storage mechanism does not match our access pattern

Chunked storage 2/2

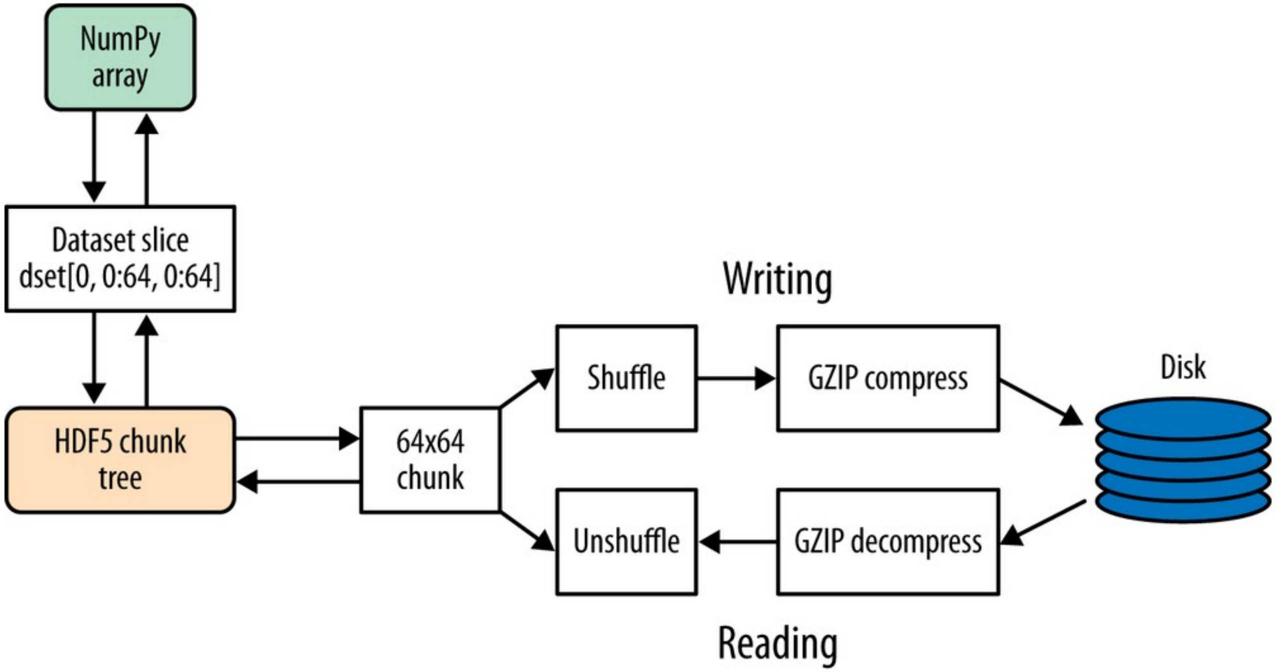
- There is a way to preserve the shape of the dataset but tell HDF5 to optimize the dataset for access in 64×64 pixel blocks
- That's what **chunking** does in HDF5. HDF5 splits the data into "chunks" of the specified shape, flattens them, and writes them to disk
- The chunks are stored in various places in the file and their coordinates are indexed by a B-tree

```
dset = f.create_dataset('chunked', (10,2048,2048), dtype=np.uint16,
                        chunks=(1,64,64))
```



Compression filters 1/2

- HDF5 has the concept of a **filter pipeline**, which is just a series of operations performed on each chunk when it's written
- Each filter is free to do anything it wants to the data in the chunk: compress it, checksum it, add metadata, anything
- When the file is read, each filter is run in "reverse" mode to reconstruct the original data



Compression filters 2/2

- A number of compression filters are available in HDF5. By far the most commonly used is the **GZIP** filter

```
dset = f.require_dataset('auto_chunked', (2048,2048), dtype=np.float32, compression="gzip")
print(dset.compression)
print(dset.compression_opts)
print(dset.chunks)
```

```
gzip
4
(64, 128)
```

- You'll notice that the auto-chunker has selected a chunk shape for us: (64, 128)
  - Data is broken up into chunks of 64*128*(4 bytes) = 32KiB blocks for the compressor

Some additional comments on HDF5

- Attributes in HDF5 can be considered the analogous of FITS keywords. They are considered the element bringing the self-describing feature in HDF5
- However, the HDF5 standard does not provide an annotation feature for the attributes, i.e. the analogous of keyword comments in FITS
  - But there is an official XML Schema language to describe HDF5 structures: https://support.hdfgroup.org/HDF5/XML/
- The attribute has only two parts, name and value. The value can be also an array or a compound type. This means that attributes cannot be organized in hierarchies (they are flat as the FITS keywords)
- There is no standard mechanism to specify units of measurement for datasets or attributes
- Metadata has to be stored also in a DBMS or XML DB. Consistency has to be maintained between the metadata content of the file and the one in the DBMS
  - Metadata mapping tools are not standard

Small exercise with the HDF5 file

- You can try now to model the NISP frame and source catalog in a single HDF5 file
- Some suggestions:
  - Use a top level group, has shown in the hdf5_example project, to store the NISP frame common metadata
  - Use a subgroup for each detector, in order to store the attributes specific for each detector and its three images (science, DQ and RMS)
  - Define a separate group for the source catalog, using a separate dataset for each detector (it will contain the sources detected on the detector image)
  - Create a reference between the detector group or the detector science image and the corresponding source catalog
  - Obviously, you can use dummy or random data to fill the datasets and the attributes