gridDataFormats Documentation

Release 1.0.1+2.g069e535.dirty

Oliver Beckstein, Jan Domanski, Jesse Johnson, Max Linke, Tyler

Aug 01, 2022

Contents

Release 1.0.1+2.g069e535.dirty

Date Aug 01, 2022

Citation

The *gridData* module contains a simple class *Grid* that makes it easier to work with data on a regular grid. A limited number of commonly used formats can be read and written as described in *Supported file formats*.

Participating

Ask questions on the mdnalysis-discussion mailing list and join the discussion.

Please report problems and enhancement requests through the issue tracker.

GridDataFormats is open source and welcomes your contributions. Fork the GridDataFormats repository on GitHub and submit a pull request. Participate on the developer mailing list.

CHAPTER 1

Installation

GridDataFormats can be easily installed via the *conda* or *pip* package managers.

It is a pure-python package but it has a few other packages (namely scipy) as dependencies that contain compiled code. For ease of installation we recommend *conda* but *pip* and installation from source are also fully supported.

1.1 Installing GridDataFormats with `conda`

The conda package manager installs, runs, and updates whole environments with all their dependencies.

Installing *GridDataFormats* from the *conda-forge* channel can be achieved by adding "conda-forge" to your channels with:

```
conda config --add channels conda-forge
```

Once the *conda-forge* channel has been enabled, *GridDataFormats* can be installed with:

```
conda install griddataformats
```

Any missing dependencies will be automatically downloaded and installed in the appropriate versions.

You can later update with

```
conda update griddataformats
```

1.2 Installing GridDataFormats with `pip`

Install with pip:

```
pip install gridDataFormats
```

and you can later update with

```
pip install --upgrade gridDataFormats
```

pip also automatically downloads all missing dependencies and will attempt to compile them if necessary; this step can fail if you do not have the correct build environment with the necessary compilers installed. You should then read the pip documentation to learn what is needed or switch to the *conda installation*.

CHAPTER 2

Handling grids of data — `gridData`

2.1 Overview

This module contains classes that allow importing and exporting of simple gridded data, A grid is an N-dimensional array that represents a discrete mesh over a region of space. The array axes are taken to be parallel to the cartesian axes of this space. Together with this array we also store the edges, which are are (essentially) the cartesian coordinates of the intersections of the grid (mesh) lines on the axes. In this way the grid is anchored in space.

The *Grid* object can be resampled at arbitrary resolution (by interpolating the data). Standard algebraic operations are defined for grids on a point-wise basis (same as for `numpy.ndarray`).

2.2 Description

The package reads grid data from files, makes them available as a *Grid* object, and allows one to write out the data again.

A *Grid* consists of a rectangular, regular, N-dimensional array of data. It contains

(1) The position of the array cell edges.

(2) The array data itself.

This is equivalent to knowing

(1) The origin of the coordinate system (i.e. which data cell corresponds to (0,0,...,0)

(2) The spacing of the grid in each dimension.

(3) The data on a grid.

Grid objects have some convenient properties:

- The data is represented as a `numpy.ndarray` in `Grid.grid` and thus can be directly manipulated with all the tools available in NumPy.

- `Grid` instances can be manipulated arithmetically, e.g. one can simply add or subtract two of them and get another one, or multiply by a constant. Note that all operations are defined point-wise (see the `numpy` documentation for details) and that only grids defined on the same cell edges can be combined.
- A *`Grid`* object can also be created from within python code e.g. from the output of the `numpy.histogramdd()` function.
- The representation of the data is abstracted from the format that the files are saved in. This makes it straightforward to add additional readers for new formats.
- The data can be written out again in formats that are understood by other programs such as VMD, ChimeraX or PyMOL.

2.3 Reading grid data files

Some *Formats* can be read directly from a file on disk:

```
g = Grid(filename)
```

filename could be, for instance, "density.dx".

2.4 Constructing a Grid

Data from an n-dimensional array can be packaged as a *`Grid`* for convenient handling (especially export to other formats). The *`Grid`* class acts as a universal constructor:

```
g = Grid(ndarray, edges=edges)                  # from histogramdd
g = Grid(ndarray, origin=origin, delta=delta)   # from arbitrary data

g.export(filename, format)    # export to the desire format
```

See the doc string for *`Grid`* for details.

2.5 Formats

For the available file formats see *Supported file formats*.

CHAPTER 3

Basic use

In most cases, only one class is important, the *Grid*, so we just load this right away:

```
from gridData import Grid
```

3.1 Loading data

From a OpenDX file:

```
g = Grid("density.dx")
```

(See also *Reading and writing OpenDX files* for more information, especially when working with visualization programs such as PyMOL, VMD, or Chimera.)

From a gOpenMol PLT file:

```
g = Grid("density.plt")
```

From the output of `numpy.histogramdd()`:

```
import numpy
r = numpy.random.randn(100,3)
H, edges = np.histogramdd(r, bins = (5, 8, 4))
g = Grid(H, edges=edges)
```

For other ways to load data, see the docs for *Grid*.

3.2 Writing out data

Some formats support writing data (see *Supported file formats* for more details), using the *gridData.core.Grid.export()* method:

```
g.export("density.dx")
```

The format can also be specified explicitly:

```
g.export("density.pkl", file_format="pickle")
```

Some of the exporters (such as for OpenDX, see *Reading and writing OpenDX files*) may take additional, format-specific keywords, which are documented separately.

3.3 Subtracting two densities

Assuming one has two densities that were generated on the same grid positions, stored in files `A.dx` and `B.dx`, one first reads the data into two *Grid* objects:

```
A = Grid('A.dx')
B = Grid('B.dx')
```

Subtract A from B:

```
C = B - A
```

and write out as a dx file:

```
C.export('C.dx')
```

The resulting file `C.dx` can be visualized with any OpenDX-capable viewer, or later read-in again.

3.4 Resampling

Load data:

```
A = Grid('A.dx')
```

Interpolate with a cubic spline to twice the sample density:

```
A2 = A.resample_factor(2)
```

Downsample to half of the bins in each dimension:

```
Ahalf = A.resample_factor(0.5)
```

Resample to the grid of another density, B:

```
B = Grid('B.dx')
A_on_B = A.resample(B.edges)
```

or even simpler

```
A_on_B = A.resample(B)
```

Note: The cubic spline generates region with values that did not occur in the original data; in particular if the original data's lowest value was 0 then the spline interpolation will probably produce some values <0 near regions where the density changed abruptly.

CHAPTER 4

Core functionality for storing n-D grids — `gridData.core`

The `core` module contains classes and functions that are independent of the grid data format. In particular this module contains the *Grid* class that acts as a universal constructor for specific formats:

```
g = Grid(**kwargs)            # construct
g.export(filename, format)    # export to the desired format
```

Some formats can also be read:

```
g = Grid()                    # make an empty Grid
g.load(filename)              # populate with data from filename
```

4.1 Classes and functions

class gridData.core.**Grid**(*grid=None*, *edges=None*, *origin=None*, *delta=None*, *metadata=None*, *interpolation_spline_order=3*, *file_format=None*)

A multidimensional grid object with origin and grid spacings.

Grid objects can be used in arithmetical calculations just like numpy arrays *if* they are compatible, i.e., they have the same shapes and lengths. In order to make arrays compatible, they an be resampled (*resample()*) on a common grid.

The attribute *grid* that holds the data is a standard numpy array and so the data can be directly manipulated.

Data can be read from a number of molecular volume/density formats and written out in different formats with *export()*.

Parameters

- **grid** (*numpy.ndarray or str (optional)*) – Build the grid either from a histogram or density (a numpy nD array) or read data from a filename.
- **edges** (*list (optional)*) – List of arrays, the lower and upper bin edges along the axes (same as the output by `numpy.histogramdd()`)

- **origin** (numpy.ndarray (optional)) – Cartesian coordinates of the center of grid position at index `[0, 0, ..., 0]`.
- **delta** (numpy.ndarray (optional)) – Either `n x n` array containing the cell lengths in each dimension, or `n x 1` array for rectangular arrays.
- **metadata** (*dict (optional)*) – A user defined dictionary of arbitrary key/value pairs associated with the density; the class does not touch *metadata* but stores it with *save()*
- **interpolation_spline_order** (*int (optional)*) – Order of interpolation function for resampling with *resample()*; cubic splines = 3 and the default is 3
- **file_format** (*str (optional)*) – Name of the file format; only necessary when *grid* is a filename (see *load()*) and autodetection of the file format fails. The default is `None` and normally the file format is guessed from the file extension.

Raises

- `TypeError` – If the dimensions of the various input data do not agree with each other.
- `ValueError` – If some of the required data are not provided in the keyword arguments, e.g., if only the *grid* is supplied as an array but not the *edges* or only *grid* and one of *origin* and *delta*.
- `NotImplementedError` – If triclinic (non-orthorhombic) boxes are supplied in *delta*

Note: *delta* can only be a 1D array of length `grid.ndim`

---

grid
This array can be any number of dimensions supported by NumPy in order to represent high-dimensional data. When used with data that represents real space densities then the **axis convention in GridDataFormats** is that axis 0 corresponds to the Cartesian x component, axis 1 corresponds to the y component, and axis 2 to the z component.

Type numpy.ndarray

delta
Length of a grid cell (spacing or voxelsize) in x, y, z dimensions. This is a *1D array* with length `Grid.grid.ndim`.

Type numpy.ndarray

origin
Array with the Cartesian coordinates of the coordinate system origin, the *center* of cell `Grid.grid[0, 0, .., 0]`.

Type numpy.ndarray

edges
List of arrays, one for each axis in *grid*. Each 1D edge array describes the *edges* of the grid cells along the corresponding axis. The length of an edge array for axis `i` is `grid.shape[i] + 1` because it contains the lower boundary for the first cell, the boundaries between all grid cells, and the upper boundary for the last cell. The edges are assumed to be regular with spacing indicated in *delta*, namely `Grid.delta[i]` for axis `i`.

Type list

midpoints
List of arrays, one for each axis in *grid*. Each 1D midpoints array contains the *midpoints* of the grid cells along the corresponding axis.

Type list

metadata
A user-defined dictionary that can be used to annotate the data. The content is not touched by *Grid*. It is saved together with the other data with *save()*.

Type dict

Example

Create a Grid object from data.

From `numpy.histogramdd()`:

```
grid, edges = numpy.histogramdd(...)
g = Grid(grid, edges=edges)
```

From an arbitrary grid:

```
g = Grid(grid, origin=origin, delta=delta)
```

From a saved file:

```
g = Grid(filename)
```

or

```
g = Grid()
g.load(filename)
```

Notes

In principle, the dimension (number of axes) is arbitrary but in practice many formats only support three and almost all functionality is only tested for this special case.

The *export()* method with `format='dx'` always exports a 3D object. Other methods might work for an array of any dimension (in particular the Python pickle output).

Changed in version 0.5.0: New *file_format* keyword argument.

Changed in version 0.7.0: CCP4 files are now read with *gridData.mrc.MRC* and not anymore with the deprecated/buggy *ccp4.CCP4*

centers()
Returns the coordinates of the centers of all grid cells as an iterator.

See also:

`numpy.ndindex()`

check_compatible(*other*)
Check if *other* can be used in an arithmetic operation.

other is compatible if

1) *other* is a scalar

2) *other* is a grid defined on the same edges

In order to make *other* compatible, resample it on the same grid as this one using *resample()*.

Parameters **other** (*Grid or float or int*) – Another object to be used for standard arithmetic operations with this *Grid*

Raises TypeError – if not compatible

See also:

resample()

default_format = 'DX'
Default format for exporting with *export()*.

export (*filename*, *file_format=None*, *type=None*, *typequote='"'*)
export density to file using the given format.

The format can also be deduced from the suffix of the filename although the *file_format* keyword takes precedence.

The default format for *export()* is 'dx'. Use 'dx' for visualization.

Implemented formats:

dx OpenDX

pickle pickle (use *Grid.load()* to restore); *Grid.save()* is simpler than `export(format='python')`.

Parameters

- **filename** (*str*) – name of the output file
- **file_format** (*{'dx', 'pickle', None} (optional)*) – output file format, the default is "dx"
- **type** (*str (optional)*) – for DX, set the output DX array type, e.g., "double" or "float". By default (None), the DX type is determined from the numpy dtype of the array of the grid (and this will typically result in "double").

  New in version 0.4.0.
- **typequote** (*str (optional)*) – For DX, set the character used to quote the type string; by default this is a double-quote character, '"'. Custom parsers like the one from NAMD-GridForces (backend for MDFF) expect no quotes, and typequote='' may be used to appease them.

  New in version 0.5.0.

interpolated
B-spline function over the data grid(x,y,z).

The *interpolated()* function allows one to obtain data values for any values of the coordinates:

```
interpolated([x1,x2,...],[y1,y2,...],[z1,z2,...]) -> F[x1,y1,z1],F[x2,y2,z2],.
↪..
```

The interpolation order is set in *Grid.interpolation_spline_order*.

The interpolated function is computed once and is cached for better performance. Whenever *interpolation_spline_order* is modified, *Grid.interpolated()* is recomputed.

The value for unknown data is set in `Grid.interpolation_cval` (TODO: also recompute when `interpolation_cval` value is changed.)

Example

Example usage for resampling:

```
XX, YY, ZZ = numpy.mgrid[40:75:0.5, 96:150:0.5, 20:50:0.5]
FF = interpolated(XX, YY, ZZ)
```

Note: Values are interpolated with a spline function. It is possible that the spline will generate values that would not normally appear in the data. For example, a density is non-negative but a cubic spline interpolation can generate negative values, especially at the boundary between 0 and high values.

Internally, the function uses `scipy.ndimage.map_coordinates()` with `mode="constant"` whereby interpolated values outside the interpolated grid are determined by filling all values beyond the edge with the same constant value, defined by the `interpolation_cval` parameter, which when not set defaults to the minimum value in the interpolated grid.

Changed in version 0.6.0: Interpolation outside the grid is now performed with `mode="constant"` rather than `mode="nearest"`, eliminating extruded volumes when interpolating beyond the grid.

interpolation_spline_order
Order of the B-spline interpolation of the data.

3 = cubic; 4 & 5 are also supported

Only choose values that are acceptable to `scipy.ndimage.spline_filter()`!

See also:

interpolated

load(*filename*, *file_format=None*)
Load saved grid and edges from *filename*

The *load()* method calls the class's constructor method and completely resets all values, based on the loaded data.

resample(*edges*)
Resample data to a new grid with edges *edges*.

This method creates a new grid with the data from the current grid resampled to a regular grid specified by *edges*. The order of the interpolation is set by *Grid.interpolation_spline_order*: change the value *before* calling *resample()*.

Parameters **edges** (*tuple of arrays or* Grid) – edges of the new grid or a *Grid* instance that provides *Grid.edges*

Returns a new *Grid* with the data interpolated over the new grid cells

Return type *Grid*

Examples

Providing *edges* (a tuple of three arrays, indicating the boundaries of each grid cell):

```
g = grid.resample(edges)
```

As a convenience, one can also supply another *Grid* as the argument for this method

```
g = grid.resample(othergrid)
```

and the edges are taken from *Grid.edges*.

resample_factor (*factor*)
Resample to a new regular grid.

Parameters **factor** (*float*) – The number of grid cells are scaled with *factor* in each dimension, i.e., `factor * N_i` cells along each dimension i. Must be positive, and cannot result in fewer than 2 cells along a dimension.

Returns **interpolated grid** – The resampled data are represented on a *Grid* with the new grid cell sizes.

Return type *Grid*

See also:

resample()

Changed in version 0.6.0: Previous implementations would not alter the range of the grid edges being resampled on. As a result, values at the grid edges would creep steadily inward. The new implementation recalculates the extent of grid edges for every resampling.

save (*filename*)
Save a grid object to *filename* and add ".pickle" extension.

Internally, this calls `Grid.export(filename, format="python")`. A grid can be regenerated from the saved data with

```
g = Grid(filename="grid.pickle")
```

Note: The pickle format depends on the Python version and therefore it is not guaranteed that a grid saved with, say, Python 2.7 can also be read with Python 3.5. The OpenDX format is a better alternative for portability.

gridData.core.**ndmeshgrid**(**arrs*)
Return a mesh grid for N dimensions.

The input are N arrays, each of which contains the values along one axis of the coordinate system. The arrays do not have to have the same number of entries. The function returns arrays that can be fed into numpy functions so that they produce values for *all* points spanned by the axes *arrs*.

Original from http://stackoverflow.com/questions/1827489/numpy-meshgrid-in-3d and fixed.

CHAPTER 5

Formats

A limited number of commonly used formats can be read or written. The formats are particularly suitable to interface with molecular visualization tools such as VMD, PyMOL, or Chimera.

Adding new formats is not difficult and user-contributed format reader/writer can be easily integrated—send a pull request.

5.1 Supported file formats

The package can be easily extended. The OpenDX format is widely understood by many molecular viewers and is sufficient for many applications that were encountered so far. Hence, at the moment only a small number of file formats is directly supported.

Table 1: Available file formats in `gridData`

module or class	format	extension	read	write	remarks
OpenDX	OpenDX	dx	x	x	subset of OpenDX implemented
gOpenMol	gOpenMol	plt	x		
mrc	CCP4	ccp4,mrc	x		subset implemented
Grid	pickle	pickle	x	x	standard Python pickle of the Grid class

5.2 Format-specific modules

5.2.1 `OpenDX` — routines to read and write simple OpenDX files

The OpenDX format for multi-dimensional grid data. OpenDX is a free visualization software, see http://www.opendx.org.

Note: This module only implements a primitive subset, sufficient to represent n-dimensional regular grids.

The OpenDX scalar file format is specified in Appendix B.2 Data Explorer Native Files[1].

If you want to build a dx object from your data you can either use the convenient *Grid* class from the top level module (`gridData.Grid`) or see the lower-level methods described below.

5.2.1.1 Reading and writing OpenDX files

If you have OpenDX files from other software and you just want to **read** it into a Python array then you do not really need to use the interface in *gridData.OpenDX*: just use *Grid* and load the file:

```
from gridData import Grid
g = Grid("data.dx")
```

This should work for files produced by common visualization programs (VMD, PyMOL, Chimera). The documentation for *gridData* tells you more about what to do with the *Grid* object.

If you want to **write** an OpenDX file then you just use the *gridData.core.Grid.export()* method with *file_format="dx"* (or just use a filename with extension ".dx"):

```
g.export("data.dx")
```

However, some visualization programs do not implement full OpenDX specifications and only read very specific, "OpenDX-like" files. *gridData.OpenDX* tries to be compatible with these formats. However, sometimes additional help is needed to write an OpenDX file that can be read by a specific software, as described below:

Known issues for writing OpenDX files

- APBS require the delta to be written to the seventh significant figure. The delta is now written to reflect this increase in precision.

  Changed in version 0.6.0.

- PyMOL requires OpenDX files with the type specification "double" in the *class array* section (see issue #35). By default (since release 0.4.0), the type is set to the one that most closely approximates the dtype of the numpy array `Grid.grid`, which holds all data. This is often `numpy.float64`, which will create an OpenDX type "double", which PyMOL will read.

  However, if you want to *force* a specific OpenDX type (such as "float" or "double", see *gridData.OpenDX.array.dx_types* for available values) then you can use the `type` keyword argument:

  ```
  g.export("for_pymol.dx", type="double")
  ```

  If you always want to be able to read OpenDX files with PyMOL, it is suggested to always export with `type="double"`.

  New in version 0.4.0.

5.2.1.2 Building a dx object from a numpy array `A`

If you have a numpy array `A` that represents a density in cartesian space then you can construct a dx object (named a *field* in OpenDX parlance) if you provide some additional information that fixes the coordinate system in space and defines the units along the axes.

The following data are required:

[1] The original link to the OpenDX file format specs http://opendx.sdsc.edu/docs/html/pages/usrgu068.htm#HDREDF is dead so I am linking to an archived copy at the Internet Archive , B.2 Data Explorer Native Files.

grid numpy nD array (typically a nD histogram)

grid.shape the shape of the array

origin the cartesian coordinates of the center of the (0,0,..,0) grid cell

delta $n \times n$ array with the length of a grid cell along each axis; for regular rectangular grids the off-diagonal elements are 0 and the diagonal ones correspond to the 'bin width' of the histogram, eg `delta[0,0] = 1.0` (Angstrom)

The DX data type ("type" in the DX file) is determined from the `numpy.dtype` of the `numpy.ndarray` that is provided as the *grid* (or with the *type* keyword argument to *gridData.OpenDX.array*).

For example, to build a *field*:

```
dx = OpenDX.field('density')
dx.add('positions', OpenDX.gridpositions(1, grid.shape, origin, delta))
dx.add('connections', OpenDX.gridconnections(2, grid.shape))
dx.add('data', OpenDX.array(3, grid))
```

or all with the constructor:

```
dx = OpenDX.field('density', components=dict(
          positions=OpenDX.gridpositions(1,grid.shape, d.origin, d.delta),
          connections=OpenDX.gridconnections(2, grid.shape),
          data=OpenDX.array(3, grid)))
```

5.2.1.3 Building a dx object from a dx file

One can also read data from an existing dx file:

```
dx = OpenDX.field(0)
dx.read('file.dx')
```

Only simple arrays are read and initially stored as a 1-d `numpy.ndarray` in the *dx.components['data'].array* with the `numpy.dtype` determined by the DX type in the file.

The dx *field* object has a method `histogramdd()` that produces output identical to the `numpy.histogramdd()` function by taking the stored dimension and deltas into account. In this way, one can store nD histograms in a portable and universal manner:

```
histogram, edges = dx.histogramdd()
```

5.2.1.4 Classes and functions

class gridData.OpenDX.**DXInitObject**(*classtype*, *classid*)

Storage class that holds data to initialize one of the 'real' classes such as OpenDX.array, OpenDX.gridconnections, ...

All variables are stored in args which will be turned into the arguments for the DX class.

initialize()

Initialize the corresponding DXclass from the data.

class = DXInitObject.initialize()

exception gridData.OpenDX.**DXParseError**

general exception for parsing errors in DX files

class gridData.OpenDX.**DXParser**(*filename*)
Brain-dead baroque implementation to read a simple (VMD) dx file.

Requires a OpenDX.field instance.

1) scan for 'object' lines: 'object' id 'class' class [data] [data ...]
2) parse data according to class
3) construct dx field from classes

Setup a parser for a simple DX file (from VMD)

```
>>> DXfield_object = OpenDX.field(id)
>>> p = DXparser('bulk.dx')
>>> p.parse(DXfield_object)
```

The field object will be completely rewritten (including the id if one is found in the input file. The input files component layout is currently ignored.

Note that quotes are removed from quoted strings.

apply_parser()
Apply the current parser to the token stream.

parse(*DXfield*)
Parse the dx file and construct a DX field object with component classes.

A *field* instance *DXfield* must be provided to be filled by the parser:

```
DXfield_object = OpenDX.field(*args)
parse(DXfield_object)
```

A tokenizer turns the dx file into a stream of tokens. A hierarchy of parsers examines the stream. The level-0 parser ('general') distinguishes comments and objects (level-1). The object parser calls level-3 parsers depending on the object found. The basic idea is that of a 'state machine'. There is one parser active at any time. The main loop is the general parser.

- Constructing the dx objects with classtype and classid is not implemented yet.
- Unknown tokens raise an exception.

set_parser(*parsername*)
Set parsername as the current parser.

use_parser(*parsername*)
Set parsername as the current parser and apply it.

exception gridData.OpenDX.**DXParserNoTokens**
raised when the token buffer is exhausted

class gridData.OpenDX.**DXclass**(*classid*)
'class' object as defined by OpenDX

id is the object number

ndformat(*s*)
Returns a string with as many repetitions of s as self has dimensions (derived from shape)

write(*stream*, *optstring=''*, *quote=False*)
write the 'object' line; additional args are packed in string

class gridData.OpenDX.**array**(*classid*, *array=None*, *type=None*, *typequote='"'*, ***kwargs*)
OpenDX array class.

See Array Objects for details.

Parameters

- **classid** (*int*) –
- **array** (*array_like*) –
- **type** (*str (optional)*) – Set the DX type in the output file and cast *array* to the closest numpy dtype. *type* must be one of the allowed types in DX files as defined under Array Objects. The default `None` tries to set the type from the `numpy.dtype` of *array*.

  New in version 0.4.0.

Raises `ValueError` – if *array* is not provided; or if *type* is not of the correct DX type

dx_types = {'byte': 'uint8', 'double': 'float64', 'float': 'float32', 'int': 'int3
conversion from OpenDX type to closest `numpy.dtype` (round-tripping is not guaranteed to produce identical types); not all types are supported (e.g., strings and conversion to int64 are missing)

np_types = {'float16': 'float', 'float32': 'float', 'float64': 'double', 'int16':
conversion from `numpy.dtype.name` to closest OpenDX array type (round-tripping is not guaranteed to produce identical types); not all types are supported (e.g., strings are missing)

write(*stream*)
Write the *class array* section.

Parameters **stream** (*stream*) –

Raises `ValueError` – If the *dxtype* is not a valid type, `ValueError` is raised.

class gridData.OpenDX.**field**(*classid='0'*, *components=None*, *comments=None*)
OpenDX container class

The *field* is the top-level object and represents the whole OpenDX file. It contains a number of other objects.

Instantiate a DX object from this class and add subclasses with *add()*.

OpenDX object, which is build from a list of components.

Parameters

- **id** (*str*) – arbitrary string
- **components** (*dict*) – dictionary of DXclass instances (no sanity check on the individual ids!) which correspond to
  - positions
  - connections
  - data
- **comments** (*list*) – list of strings; each string becomes a comment line prefixed with '#'. Avoid newlines.

A field must have at least the components 'positions', 'connections', and 'data'. Those components are associated with objects belonging to the field. When writing a dx file from the field, only the required objects are dumped to the file.

(For a more general class that can use field: Because there could be more objects than components, we keep a separate object list. When dumping the dx file, first all objects are written and then the field object describes its components. Objects are referenced by their unique id.)

Note: uniqueness of the *id* is not checked.

Example

Create a new dx object:

```
dx = OpenDX.field('density',[gridpoints,gridconnections,array])
```

add(*component*, *DXobj*)
add a component to the field

add_comment(*comment*)
add comments

histogramdd()
Return array data as (edges,grid), i.e. a numpy nD histogram.

read(*stream*)
Read DX field from file.

dx = OpenDX.field.read(dxfile)

The classid is discarded and replaced with the one from the file.

sorted_components()
iterator that returns (component,object) in id order

write(*filename*)
Write the complete dx object to the file.

This is the simple OpenDX format which includes the data into the header via the 'object array ... data follows' statement.

Only simple regular arrays are supported.

The format should be compatible with VMD's dx reader plugin.

class gridData.OpenDX.**gridconnections**(*classid*, *shape=None*, ***kwargs*)
OpenDX gridconnections class

write(*stream*)
write the 'object' line; additional args are packed in string

class gridData.OpenDX.**gridpositions**(*classid*, *shape=None*, *origin=None*, *delta=None*, ***kwargs*)
OpenDX gridpositions class.

shape D-tuplet describing size in each dimension origin coordinates of the centre of the grid cell with index 0,0,...,0 delta DxD array describing the deltas

edges()
Edges of the grid cells, origin at centre of 0,0,..,0 grid cell.

Only works for regular, orthonormal grids.

write(*stream*)
write the 'object' line; additional args are packed in string

5.2.2 `gOpenMol` — the gOpenMol plt format

The module provides a simple implementation of a reader for gOpenMol *plt* files. Plt files are binary files. The `Plt` reader tries to guess the endianess of the file, but this can fail (with a `TypeError`); you are on your own in this case.

Only the reader is implemented. If you want to write gridded data use a format that is more standard, such as OpenDX (see `OpenDX`).

5.2.2.1 Background

gOpenMol http://www.csc.fi/english/pages/g0penMol plt format.

Used to be documented at http://www.csc.fi/gopenmol/developers/plt_format.phtml but currently this is only accessible through the internet archive at http://web.archive.org/web/20061011125817/http://www.csc.fi/gopenmol/developers/plt_format.phtml

5.2.2.2 Grid data plt file format

Copyright CSC, 2005. Last modified: September 23, 2003 09:18:50

Plot file (plt) format The plot files are regular 3D grid files for plotting of molecular orbitals, electron densities or other molecular properties. The plot files are produced by several programs. It is also possible to format/unformat plot files using the pltfile program in the utility directory. It is also possible to produce plot files with external (own) programs. Produce first a formatted text file and use then the pltfile program to unformat the file for gOpenMol. The format for the plot files are very simple and a description of the format can be found elsewhere in this manual. gOpenMol can read binary plot files from different hardware platforms independent of the system type (little or big endian machines).

Format of the binary `*.plt` file

The `*.plt` file binary and formatted file formats are very simple but please observe that unformatted files written with a FORTRAN program are not pure binary files because there are file records between the values while pure binary files do not have any records between the values. gOpenMol should be able to figure out if the file is pure binary or FORTRAN unformatted but it is not very well tested.

Binary `*.plt` (grid) file format

Record number and meaning:

```
#1: Integer, rank value must always be = 3
#2: Integer, possible values are 1 ... 50. This value is not used but
it can be used to define the type of surface!
Values used (you can use your own value between 1... 50):

1:   VSS surface
2:   Orbital/density surface
3:   Probe surface
200: Gaussian 94/98
201: Jaguar
202: Gamess
203: AutoDock
204: Delphi/Insight
205: Grid
```

(continues on next page)

(continued from previous page)

```
Value 100 is reserved for grid data coming from OpenMol!

#3: Integer, number of points in z direction
#4: Integer, number of points in y direction
#5: Integer, number of points in x direction
#6: Float, zmin value
#7: Float, zmax value
#8: Float, ymin value
#9: Float, ymax value
#10: Float, xmin value
#11: Float, xmax value
#12 ... Float, grid data values running (x is inner loop, then y and last z):
```

1. Loop in the z direction
2. Loop in the y direction
3. Loop in the x direction

Example:

```
nx=2  ny=1  nz=3

0,0,0   1,0,0     y=0, z=0
0,0,1   1,0,0     y=0, z=1
0,0,2   1,0,2     y=0, z=2
```

The formatted (the first few lines) file can look like:

```
3 2
65 65 65
-3.300000e+001 3.200000e+001 -3.300000e+001 3.200000e+001 -3.300000e+001 3.200000e+001
-1.625609e+001 -1.644741e+001 -1.663923e+001 -1.683115e+001 -1.702274e+001 -1.
↪721340e+001
-1.740280e+001 -1.759018e+001 -1.777478e+001 -1.795639e+001 -1.813387e+001 -1.
↪830635e+001
...
```

Formatted `*.plt` (grid) file format

Line numbers and variables on the line:

```
line #1: Integer, Integer. Rank and type of surface (rank is always = 3)
line #2: Integer, Integer, Integer. Zdim, Ydim, Xdim (number of points in the z,y,x␣
↪directions)
line #3: Float, Float, Float, Float, Float, Float. Zmin, Zmax, Ymin, Ymax, Xmin,Xmax␣
↪(min and max values)
line #4: ... Float. Grid data values running (x is inner loop, then y and last z)␣
↪with one or several values per line:

 1. Loop in the z direction
 2. Loop in the y direction
 3. Loop in the x direction
```

5.2.2.3 Classes

class gridData.gOpenMol.**Plt**(*filename=None*)
A class to represent a gOpenMol plt file.

Only reading is implemented; either supply a filename to the constructor

```
>>> G = Plt(filename)
```

or load the file with the read method

```
>>> G = Plt()
>>> G.read(filename)
```

The data is held in `GOpenMol.array` and all header information is in the dict `GOpenMol.header`.

Plt.shape D-tuplet describing size in each dimension

Plt.origin coordinates of the centre of the grid cell with index 0,0,...,0

Plt.delta DxD array describing the deltas

edges
Edges of the grid cells, origin at centre of 0,0,..,0 grid cell.

Only works for regular, orthonormal grids.

histogramdd()
Return array data as (edges,grid), i.e. a numpy nD histogram.

read(*filename*)
Populate the instance from the plt file *filename*.

5.2.3 `mrc` — the MRC/CCP4 volumetric data format

New in version 0.7.0.

Reading of MRC/CCP4 volumetric files (MRC2014 file format) using the mrcfile library [Burnley2017].

References

5.2.3.1 Classes

class gridData.mrc.**MRC**(*filename=None*)
Represent a MRC/CCP4 file.

Load MRC/CCP4 2014 3D volumetric data with the mrcfile library.

Parameters **filename** (*str (optional)*) – input file (or stream), can be compressed

Raises `ValueError` – If the unit cell is not orthorhombic or if the data are not volumetric.

header
Header data from the MRC file as a numpy record array.

Type numpy.recarray

array
Data as a 3-dimensional array where axis 0 corresponds to X, axis 1 to Y, and axis 2 to Z. This order is always enforced, regardless of the order in the mrc file.

Type numpy.ndarray

delta
Diagonal matrix with the voxel size in X, Y, and Z direction (taken from the `mrcfile.mrcfile.voxel_size` attribute)

Type numpy.ndarray

origin
numpy array with coordinates of the coordinate system origin (computed from `header.origin`, the offsets `header.origin.nxstart`, `header.origin.nystart`, `header.origin.nzstart` and the spacing *delta*)

Type numpy.ndarray

rank
The integer 3, denoting that only 3D maps are read.

Type int

Notes

- Only volumetric (3D) densities are read.
- Only orthorhombic unitcells supported (other raise `ValueError`)
- Only reading is currently supported.

New in version 0.7.0.

edges
Edges of the grid cells, origin at centre of 0,0,0 grid cell.

Only works for regular, orthonormal grids.

histogramdd()
Return array data as (edges,grid), i.e. a numpy nD histogram.

read(*filename*)
Populate the instance from the MRC/CCP4 file *filename*.

shape
Shape of the *array*

Bibliography

[Burnley2017] Burnley T, Palmer C and Winn M (2017) Recent developments in the CCP-EM software suite. *Acta Cryst.* D73:469-477. doi: 10.1107/S2059798317007859

Python Module Index

g

Index

R

S

U

W