
gridDataFormats Documentation

Release 0.3.3

Oliver Beckstein

May 13, 2016

1	gridData – Handling grids of data	3
2	Examples	5
3	Formats	7
4	gridData.core — Core functionality for storing n-D grids	17
5	Indices and tables	19
	Python Module Index	21

Release 0.3.3

Date May 13, 2016

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*.

The code is available under the [Lesser GNU General Public License, version 3](#) (see also the files `COPYING` and `COPYING.LESSER` in the source distribution). Releases are available from the Python Package index under [GridDataFormats](#) and source code is available from the GitHub repository <https://github.com/MDAnalysis/GridDataFormats>. Please report problems and enhancement requests through the [issue tracker](#).

Contents

gridData – Handling grids of data

1.1 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 (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`).

1.2 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 $(0,0,\dots,0)$)
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` and thus shares all the advantages coming with this sophisticated and powerful library.
- They 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 or PyMOL.

1.3 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”.

1.4 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.

1.5 Formats

The following formats are available (*Supported file formats*):

OpenDX IBM’s Data Explorer, <http://www.opendx.org/>

gOpenMol <http://www.csc.fi/gopenmol/>

pickle python pickle file (`pickle`)

Examples

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

```
from gridData import Grid
```

2.1 Loading data

From a OpenDX file:

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

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*.

2.2 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.

2.3 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.

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](#) or [PyMOL](#).

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

3.1 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 3.1: Available file formats in `gridData`

module or class	format	extension	read	write	remarks
<i>OpenDX</i>	OpenDX	dx	x	x	subset of OpenDX implemented
<i>gOpenMol</i>	gOpenMol	plt	x		
<i>CCP4</i>	CCP4	ccp4	x		subset implemented
<i>Grid</i>	pickle	pickle	x	x	standard Python pickle of the Grid class

3.2 Format-specific modules

3.2.1 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.

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](#).

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:

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)

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)))
```

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')
```

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

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

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

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`

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

write (*file*, *optstring*=‘’, *quote*=*False*)
write the ‘object’ line; additional args are packed in string

class `gridData.OpenDX.array` (*classid*, *array*=*None*, ***kwargs*)
`OpenDX` array class

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.

Methods overview:

add () add a component to the field

add_comments () add comments

write () write `OpenDX` file to file descriptor; only simple regular arrays are supported. File should be readable by VMD.

read() construct the field from a dx file

Instantiated a DX object from this class and add subclasses with `append()`.

histogramdd()

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

read(file)

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.

`write(filename)`

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

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

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

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.

3.2.2 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 endianness 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).

Background

`gOpenMol` http://www.csc.fi/english/pages/gOpenMol_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

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

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 cell

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

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*.

3.2.3 CCP4 — the CCP4 volumetric data format

New in version 0.3.0.

The module provides a simple implementation of a reader for CCP4 *ccp4* files. CCP4 files are binary files. The *CCP4* reader tries to guess the endianness 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`).

Background

CCP4 format: <http://www.ccp4.ac.uk/html/maplib.html#description>

Used to be more carefully documented at http://lsbr.niams.nih.gov/3demc/3demc_maplib.html but currently this is only accessible through the Google cache <http://webcache.googleusercontent.com/search?q=cache:KRSvXB0S3dsJ:lsbr.niams.nih.gov/3>

Grid data CCP4 file format

Copyright Science and Technologies Facilities Council, 2015.

The overall layout of the file is as follows:

```
File header (256 longwords)
Symmetry information
Map, stored as a 3-dimensional array
```

The header is organised as 56 words followed by space for ten 80 character text labels as follows:

```
1      NC          # of Columns    (fastest changing in map)
2      NR          # of Rows
3      NS          # of Sections    (slowest changing in map)
4      MODE        Data type
                    0 = envelope stored as signed bytes (from
                    -128 lowest to 127 highest)
                    1 = Image      stored as Integer*2
                    2 = Image      stored as Reals
                    3 = Transform stored as Complex Integer*2
                    4 = Transform stored as Complex Reals
                    5 == 0

                    Note: Mode 2 is the normal mode used in
                    the CCP4 programs. Other modes than 2 and 0
                    may NOT WORK

5      NCSTART     Number of first COLUMN in map
6      NRSTART     Number of first ROW in map
7      NSSTART     Number of first SECTION in map
8      NX          Number of intervals along X
9      NY          Number of intervals along Y
10     NZ          Number of intervals along Z
11     X length    Cell Dimensions (Angstroms)
12     Y length    "
13     Z length    "
```

14	Alpha	Cell Angles	(Degrees)
15	Beta		"
16	Gamma		"
17	MAPC	Which axis corresponds to Cols.	(1,2,3 for X,Y,Z)
18	MAPR	Which axis corresponds to Rows	(1,2,3 for X,Y,Z)
19	MAPS	Which axis corresponds to Sects.	(1,2,3 for X,Y,Z)
20	AMIN	Minimum density value	
21	AMAX	Maximum density value	
22	AMEAN	Mean density value	(Average)
23	ISPG	Space group number	
24	NSYMBT	Number of bytes used for storing symmetry operators	
25	LSKFLG	Flag for skew transformation, =0 none, =1 if foll	
26-34	SKWMAT	Skew matrix S (in order S11, S12, S13, S21 etc) if	LSKFLG .ne. 0.
35-37	SKWTRN	Skew translation t if LSKFLG .ne. 0.	Skew transformation is from standard orthogonal coordinate frame (as used for atoms) to orthogonal map frame, as
			$Xo(\text{map}) = S * (Xo(\text{atoms}) - t)$
38	future use	(some of these are used by the MSUBSX routines	
.	"	in MAPBRICK, MAPCONT and FRODO)	
.	"	(all set to zero by default)	
.	"		
52	"		
53	MAP	Character string 'MAP '	to identify file type
54	MACHST	Machine stamp indicating the machine type	which wrote file
55	ARMS	Rms deviation of map from mean density	
56	NLABL	Number of labels being used	
57-256	LABEL(20,10)	10 80 character text labels (ie. A4 format)	

Symmetry records follow - if any - stored as text as in International Tables, operators separated by * and grouped into 'lines' of 80 characters (i.e. symmetry operators do not cross the ends of the 80-character 'lines' and the 'lines' do not terminate in a *).

Map data array follows.

Note on the machine stamp: The machine stamp (word 54) is a 32-bit quantity containing a set of four 'nibbles' (half-bytes) - only half the space is used. Each nibble is a number specifying the representation of (in C terms) double (d), float (f), int (i) and unsigned char (c) types. Thus each stamp is of the form 0xdfic0000. For little endian hardware the stamp is 0x44, 0x41, 0x00, 0x00 while the big endian stamp is 0x11, 0x11, 0x00, 0x00.

Classes

class `gridData.CCP4.CCP4` (*filename=None*)

A class to represent a **CCP4** file.

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

```
>>> G = CCP4(filename)
```

or load the file with the read method

```
>>> G = CCP4()
>>> G.read(filename)
```

The data is held in `CCP4.array` and all header information is in the dict `CCP4.header`.

CCP4.shape D-tuplet describing size in each dimension

CCP4.origin coordinates of the centre of the grid cell with index 0,0,...,0

CCP4.delta DxD array describing the deltas

Note: The following features of the CCP4 format are *not* implemented: * triclinic boxes * symmetry records * index ordering besides standard column-major and row-major * non-standard fields, such any in filed in future use block

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 ccp4 file *filename*.

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

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={}, interpolation_spline_order=3*)

Class to manage a multidimensional grid object.

The `export(format='dx')` method always exports a 3D object, the rest should work for an array of any dimension.

The grid (`Grid.grid`) can be manipulated as a standard numpy array.

The attribute `Grid.metadata` holds a user-defined dictionary that can be used to annotate the data. It is saved with `save()`.

centers ()

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

check_compatible (*other*)

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

1. *other* is a scalar

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

Raises `TypeError` if not compatible.

export (*filename, file_format=None*)

export density to file using the given format.

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

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

Implemented formats:

dx OpenDX

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

interpolated

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

`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 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)
```

load (*filename*, *file_format=None*)

Load saved (pickled or dx) grid and edges from `<filename>.pickle`

`Grid.load(<filename>.pickle)` `Grid.load(<filename>.dx)`

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*.

`resample(edges) -> Grid`

or

`resample(otherGrid) -> Grid`

The order of the interpolation is set by `Grid.interpolation_spline_order`.

resample_factor (*factor*)

Resample to a new regular grid with `factor*oldN` cells along each dimension.

save (*filename*)

Save a grid object to `<filename>.pickle`

`Grid.save(filename)`

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

`g = Grid(filename=<filename>)`

`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.

Indices and tables

- `genindex`
- `modindex`
- `search`

g

gridData, 1
gridData.CCP4, 13
gridData.core, 15
gridData.gOpenMol, 10
gridData.OpenDX, 7

A

apply_parser() (gridData.OpenDX.DXParser method), 9
array (class in gridData.OpenDX), 9

C

CCP4 (class in gridData.CCP4), 14
centers() (gridData.core.Grid method), 17
check_compatible() (gridData.core.Grid method), 17

D

DXclass (class in gridData.OpenDX), 9
DXInitObject (class in gridData.OpenDX), 8
DXParseError, 8
DXParser (class in gridData.OpenDX), 9
DXParserNoTokens, 9

E

edges (gridData.CCP4.CCP4 attribute), 15
edges (gridData.gOpenMol.Plit attribute), 12
edges() (gridData.OpenDX.gridpositions method), 10
export() (gridData.core.Grid method), 17

F

field (class in gridData.OpenDX), 9

G

Grid (class in gridData.core), 17
gridconnections (class in gridData.OpenDX), 10
gridData (module), 1
gridData.CCP4 (module), 13
gridData.core (module), 15
gridData.gOpenMol (module), 10
gridData.OpenDX (module), 7
gridpositions (class in gridData.OpenDX), 10

H

histogramdd() (gridData.CCP4.CCP4 method), 15
histogramdd() (gridData.gOpenMol.Plit method), 12
histogramdd() (gridData.OpenDX.field method), 10

I

initialize() (gridData.OpenDX.DXInitObject method), 8
interpolated (gridData.core.Grid attribute), 18

L

load() (gridData.core.Grid method), 18

N

ndformat() (gridData.OpenDX.DXclass method), 9
ndmeshgrid() (in module gridData.core), 18

P

parse() (gridData.OpenDX.DXParser method), 9
Plt (class in gridData.gOpenMol), 12

R

read() (gridData.CCP4.CCP4 method), 15
read() (gridData.gOpenMol.Plit method), 12
read() (gridData.OpenDX.field method), 10
resample() (gridData.core.Grid method), 18
resample_factor() (gridData.core.Grid method), 18

S

save() (gridData.core.Grid method), 18
set_parser() (gridData.OpenDX.DXParser method), 9
sorted_components() (gridData.OpenDX.field method),
10

U

use_parser() (gridData.OpenDX.DXParser method), 9

W

write() (gridData.OpenDX.DXclass method), 9
write() (gridData.OpenDX.field method), 10