mrcfile Documentation

Release 1.5.0

Colin Palmer

Jan 10, 2024

TABLE OF CONTENTS:

Get started by reading the *Overview of mrcfile.py* and the *Usage Guide*.

You can also look at the *Source documentation*.

CHAPTER

ONE

OVERVIEW OF MRCFILE.PY

mrcfile is a Python implementation of the MRC2014 file format, which is used in structural biology to store image and volume data.

It allows MRC files to be created and opened easily using a very simple API, which exposes the file's header and data as numpy arrays. The code runs in Python 2 and 3 and is fully unit-tested.

This library aims to allow users and developers to read and write standard-compliant MRC files in Python as easily as possible, and with no dependencies on any compiled libraries except numpy. You can use it interactively to inspect files, correct headers and so on, or in scripts and larger software packages to provide basic MRC file I/O functions.

1.1 Key Features

- Clean, simple API for access to MRC files
- Easy to install and use
- Validation of files according to the MRC2014 format
- Seamless support for gzip and bzip2 files
- Memory-mapped file option for fast random access to very large files
- Asynchronous opening option for background loading of multiple files
- Runs in Python 2 & 3, on Linux, Mac OS X and Windows

1.2 Installation

The mrcfile library is available from the Python package index:

```
pip install mrcfile
```

Or from conda-forge:

```
conda install --channel conda-forge mrcfile
```

It is also included in the ccpem-python environment in the CCP-EM software suite.

The source code (including the full test suite) can be found on GitHub.

1.3 Basic usage

The easiest way to open a file is with the mrcfile.open and mrcfile.new functions. These return an MrcFile object which represents an MRC file on disk.

To open an MRC file and read a slice of data:

```
>>> import mrcfile
>>> with mrcfile.open('tests/test_data/EMD-3197.map') as mrc:
...     mrc.data[10,10]
...
array([ 2.58179283,  3.1406002 ,  3.64495397,  3.63812137,  3.61837363,
        4.0115056 ,  3.66981959,  2.07317996,  0.1251585 , -0.87975615,
        0.12517013,  2.07319379,  3.66982722,  4.0115037 ,  3.61837196,
        3.6381247 ,  3.64495087,  3.14059472,  2.58178973,  1.92690361], dtype=float32)
```

To create a new file with a 2D data array, and change some values:

```
>>> array = np.zeros((5, 5), dtype=np.int8)
>>> with mrcfile.new('tmp.mrc') as mrc:
...     mrc.set_data(array)
...     mrc.data[1:4,1:4] = 10
...     mrc.data
...
array([[ 0,  0,  0,  0,  0],
       [ 0, 10, 10, 10,  0],
       [ 0, 10, 10, 10,  0],
       [ 0, 10, 10, 10,  0],
       [ 0,  0,  0,  0,  0]], dtype=int8)
```

The data will be saved to disk when the file is closed, either automatically at the end of the `with` block (like a normal Python file object) or manually by calling `close()`. You can also call `flush()` to write any changes to disk and keep the file open.

To validate an MRC file:

```
>>> mrcfile.validate('tests/test_data/EMD-3197.map')
File does not declare MRC format version 20140 or 20141: nversion = 0
False

>>> mrcfile.validate('tmp.mrc')
True
```

1.4 Documentation

Full documentation is available on Read the Docs.

1.5 Citing mrcfile

If you find `mrcfile` useful in your work, please cite:

Burnley T, Palmer C & Winn M (2017) Recent developments in the CCP-EM software suite. *Acta Cryst.* D**73**:469–477. doi: 10.1107/S2059798317007859

1.6 Contributing

Please use the GitHub issue tracker for bug reports and feature requests, or email CCP-EM.

Code contributions are also welcome, please submit pull requests to the GitHub repository.

To run the test suite, go to the top-level project directory (which contains the `mrcfile` and `tests` packages) and run `python -m unittest tests`. (Or, if you have tox installed, run `tox`.)

1.7 Licence

The project is released under the BSD licence.

CHAPTER

TWO

USAGE GUIDE

This is a detailed guide to using the `mrcfile` Python library. For a brief introduction, see the *overview*.

2.1 Opening MRC files

2.1.1 Normal file access

MRC files can be opened using the *`mrcfile.new()`* or *`mrcfile.open()`* functions. These return an instance of the *`MrcFile`* class, which represents an MRC file on disk and makes the file's header, extended header and data available for read and write access as numpy arrays. :

```
>>> # First, create a simple dataset
>>> import numpy as np
>>> example_data = np.arange(12, dtype=np.int8).reshape(3, 4)

>>> # Make a new MRC file and write the data to it:
>>> import mrcfile
>>> with mrcfile.new('tmp.mrc') as mrc:
...     mrc.set_data(example_data)
...
>>> # The file is now saved on disk. Open it again and check the data:
>>> with mrcfile.open('tmp.mrc') as mrc:
...     mrc.data
...
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=int8)
```

2.1.2 Simple data access

Alternatively, for even quicker access to MRC data but with minimal control of the file header, you can use the *`read()`* and *`write()`* functions. These do not return *`MrcFile`* objects but instead work directly with numpy arrays:

```
>>> # First, create a simple dataset
>>> import numpy as np
>>> example_data_2 = np.arange(6, dtype=np.int8).reshape(3, 2)

>>> # Write the data to a new MRC file:
```

(continues on next page)

(continued from previous page)

```
>>> mrcfile.write('tmp2.mrc', example_data_2)

>>> # Read it back:
>>> mrcfile.read('tmp2.mrc')
array([[0, 1],
       [2, 3],
       [4, 5]], dtype=int8)
```

2.1.3 Handling compressed files

All of the functions shown above can also handle gzip- or bzip2-compressed files very easily:

```
>>> # Make a new gzipped MRC file:
>>> with mrcfile.new('tmp.mrc.gz', compression='gzip') as mrc:
...     mrc.set_data(example_data * 2)
...
>>> # Open it again with the normal open function:
>>> with mrcfile.open('tmp.mrc.gz') as mrc:
...     mrc.data
...
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]], dtype=int8)

>>> # Same again for bzip2:
>>> with mrcfile.new('tmp.mrc.bz2', compression='bzip2') as mrc:
...     mrc.set_data(example_data * 3)
...
>>> # Open it again with the normal read function:
>>> mrcfile.read('tmp.mrc.bz2')
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33]], dtype=int8)

>>> # The write function applies compression automatically based on the file name
>>> mrcfile.write('tmp2.mrc.gz', example_data * 4)

>>> # The new file is opened as a GzipMrcFile object:
>>> with mrcfile.open('tmp2.mrc.gz') as mrc:
...     print(mrc)
...
GzipMrcFile('tmp2.mrc.gz', mode='r')
```

2.1.4 Closing files and writing to disk

MrcFile objects should be closed when they are finished with, to ensure any changes are flushed to disk and the underlying file object is closed:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> # do things...
>>> mrc.close()
```

As we saw in the examples above, *MrcFile* objects support Python's `with` statement, which will ensure the file is closed properly after use (like a normal Python file object). It's generally a good idea to use `with` if possible, but sometimes when running Python interactively (as in some of these examples), it's more convenient to open a file and keep using it without having to work in an indented block. If you do this, remember to close the file at the end!

There's also a *flush()* method that writes the MRC data to disk but leaves the file open:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> # do things...
>>> mrc.flush()  # make sure changes are written to disk
>>> # continue using the file...
>>> mrc.close()  # close the file when finished
```

2.1.5 MrcFile subclasses

For most purposes, the top-level functions in *mrcfile* should be all you need to open MRC files, but it is also possible to directly instantiate *MrcFile* and its subclasses, *GzipMrcFile*, *Bzip2MrcFile* and *MrcMemmap*:

```
>>> with mrcfile.mrcfile.MrcFile('tmp.mrc') as mrc:
...     mrc
...
MrcFile('tmp.mrc', mode='r')

>>> with mrcfile.gzipmrcfile.GzipMrcFile('tmp.mrc.gz') as mrc:
...     mrc
...
GzipMrcFile('tmp.mrc.gz', mode='r')

>>> with mrcfile.bzip2mrcfile.Bzip2MrcFile('tmp.mrc.bz2') as mrc:
...     mrc
...
Bzip2MrcFile('tmp.mrc.bz2', mode='r')

>>> with mrcfile.mrcmemmap.MrcMemmap('tmp.mrc') as mrc:
...     mrc
...
MrcMemmap('tmp.mrc', mode='r')
```

2.1.6 File modes

MrcFile objects can be opened in three modes: `r`, `r+` and `w+`. These correspond to the standard Python file modes, so `r` opens a file in read-only mode:

```
>>> # The default mode is 'r', for read-only access:
>>> mrc = mrcfile.open('tmp.mrc')
>>> mrc
MrcFile('tmp.mrc', mode='r')
>>> mrc.set_data(example_data)
Traceback (most recent call last):
  ...
ValueError: MRC object is read-only
>>> mrc.close()
```

`r+` opens it for reading and writing:

```
>>> # Using mode 'r+' allows read and write access:
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> mrc
MrcFile('tmp.mrc', mode='r+')
>>> mrc.set_data(example_data)
>>> mrc.data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=int8)
>>> mrc.close()
```

and `w+` opens a new, empty file (also for both reading and writing):

```
>>> # Mode 'w+' creates a new empty file:
>>> mrc = mrcfile.open('empty.mrc', mode='w+')
>>> mrc
MrcFile('empty.mrc', mode='w+')
>>> mrc.data
array([], dtype=int8)
>>> mrc.close()
```

The *new()* function is effectively shorthand for `open(name, mode='w+')`:

```
>>> # Make a new file
>>> mrc = mrcfile.new('empty.mrc')
Traceback (most recent call last):
  ...
ValueError: File 'empty.mrc' already exists; set overwrite=True to overwrite it
>>> # Ooops, we've already got a file with that name!
>>> # If we're sure we want to overwrite it, we can try again:
>>> mrc = mrcfile.new('empty.mrc', overwrite=True)
>>> mrc
MrcFile('empty.mrc', mode='w+')
>>> mrc.close()
```

2.1.7 Permissive read mode

Normally, if an MRC file is badly invalid, an exception is raised when the file is opened. This can be a problem if we want to, say, open a file and fix a header problem. To deal with this situation, *open()* and *mmap()* provide an optional `permissive` argument. If this is set to `True`, problems with the file will cause warnings to be issued (using Python's `warnings` module) instead of raising exceptions, and the file will continue to be interpreted as far as possible.

Let's see an example. First we'll deliberately make an invalid file:

```
>>> # Make a new file and deliberately make a mistake in the header
>>> with mrcfile.new('invalid.mrc') as mrc:
...     mrc.header.map = b'map '  # standard requires b'MAP '
...
```

Now when we try to open the file, an exception is raised:

```
>>> # Opening an invalid file raises an exception:
>>> mrc = mrcfile.open('invalid.mrc')
Traceback (most recent call last):
  ...
ValueError: Map ID string not found - not an MRC file, or file is corrupt
```

If we use permissive mode, we can open the file, and we'll see a warning about the problem (except that here, we have to catch the warning and print the message manually, because warnings don't play nicely with doctests!):

```
>>> # Opening in permissive mode succeeds, with a warning:
>>> with warnings.catch_warnings(record=True) as w:
...     mrc = mrcfile.open('invalid.mrc', permissive=True)
...     print(w[0].message)
...
Map ID string not found - not an MRC file, or file is corrupt
```

Now let's fix the file:

```
>>> # Fix the invalid file by correcting the header
>>> with mrcfile.open('invalid.mrc', mode='r+', permissive=True) as mrc:
...     mrc.header.map = mrcfile.constants.MAP_ID
...
```

And now we should be able to open the file again normally:

```
>>> # Now we don't need permissive mode to open the file any more:
>>> mrc = mrcfile.open('invalid.mrc')
>>> mrc.close()
```

The problems that can cause an exception when opening an MRC file are:

1. The header's `map` field is not set correctly to confirm the file type. If the file is otherwise correct, permissive mode should be able to read the file normally.
2. The machine stamp is invalid and so the file's byte order cannot be determined. In this case, permissive mode assumes that the byte order is little-endian and continues trying to read the file. If the file is actually big-endian, the mode and data size checks will also fail because these values depend on the endianness and will be nonsensical.
3. The mode number is not recognised. Currently accepted modes are 0, 1, 2, 4, 6 and 12.
4. The data block is not large enough for the specified data type and dimensions.

In the last two cases, the data block will not be read and the *data* attribute will be set to None.

Fixing invalid files can be quite complicated! This usage guide might be expanded in future to explain how to analyse and fix problems, or the library itself might be improved to fix certain problems automatically. For now, if you have trouble with an invalid file, inspecting the code in this library might help you to work out how to approach the problem (start with *MrcInterpreter._read_header()*), or you could try asking on the CCP-EM mailing list for advice.

2.1.8 A note on axis ordering

`mrcfile` follows the Python / C-style convention for axis ordering. This means that the first index is the slowest axis (typically Z for volume data or Y for images) and the last index is the fastest axis (typically X), and the numpy arrays are C-contiguous:

```
>>> data = mrcfile.read('tmp.mrc')
>>> data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=int8)

>>> data[1, 0]  # x = 0, y = 1
4
>>> data[2, 3]  # x = 3, y = 2
11

>>> data.flags.c_contiguous
True
>>> data.flags.f_contiguous
False
```

Note that this axis order is the opposite of the FORTRAN-style convention that is used by some other software in structural biology. This can cause confusing errors!

2.2 Using MrcFile objects

2.2.1 Accessing the header and data

The header and data arrays can be accessed using the *header*, *extended_header* and *data* attributes:

```
>>> mrc = mrcfile.open('tmp.mrc')
>>> mrc.header
rec.array((4, 3, 1, ...),
          dtype=[('nx', ...)])
>>> mrc.extended_header
array([],
      dtype='|V1')
>>> mrc.data
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=int8)
>>> mrc.close()
```

These attributes are read-only and cannot be assigned to directly, but (unless the file mode is `r`) the arrays can be modified in-place:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> # A new data array cannot be assigned directly to the data attribute
>>> mrc.data = np.ones_like(example_data)
Traceback (most recent call last):
  ...
AttributeError: can't set attribute
>>> # But the data can be modified by assigning to a slice or index
>>> mrc.data[0, 0] = 10
>>> mrc.data
array([[10,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=int8)
>>> # All of the data values can be replaced this way, as long as the data
>>> # size, shape and type are not changed
>>> mrc.data[:] = np.ones_like(example_data)
>>> mrc.data
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]], dtype=int8)
>>> mrc.close()
```

To replace the data or extended header completely, call the *set_data()* and *set_extended_header()* methods:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> data_3d = np.linspace(-1000, 1000, 20, dtype=np.int16).reshape(2, 2, 5)
>>> mrc.set_data(data_3d)
>>> mrc.data
array([[[-1000,  -894,  -789,  -684,  -578],
        [ -473,  -368,  -263,  -157,   -52]],
       [[   52,   157,   263,   368,   473],
        [  578,   684,   789,   894,  1000]]], dtype=int16)
>>> # Setting a new data array updates the header dimensions to match
>>> mrc.header.nx
array(5, dtype=int32)
>>> mrc.header.ny
array(2, dtype=int32)
>>> mrc.header.nz
array(2, dtype=int32)
>>> # We can also set the extended header in the same way
>>> string_array = np.fromstring(b'The extended header can hold any kind of numpy array',
↪ dtype='S52')
>>> mrc.set_extended_header(string_array)
>>> mrc.extended_header
array([b'The extended header can hold any kind of numpy array'],
      dtype='|S52')
>>> # Setting the extended header updates the header's nsymbt field to match
>>> mrc.header.nsymbt
array(52, dtype=int32)
>>> mrc.close()
```

Note that setting an extended header does not automatically set or change the header's `exttyp` field. You should set this yourself to identify the type of extended header you are using.

For a quick overview of the contents of a file's header, call *print_header()*:

```
>>> with mrcfile.open('tmp.mrc') as mrc:
...     mrc.print_header()
...
nx              : 5
ny              : 2
nz              : 2
mode            : 1
nxstart ...
```

2.2.2 Indexing the extended header

The *extended_header* attribute will return an array of the bytes in the extended header. However, some extended headers are structured and consist of a sequence of metadata blocks, where each block corresponds to a single image, or slice, in the data array. The attribute *indexed_extended_header* is intended for more convenient access to the indexed sequence of metadata blocks, for known extended header types. It will return an array with the appropriate numpy dtype set (or `None` in the case of failure) for an indexable extended header array, even if the extended header itself contains trailing padding bytes.

Currently two extended header types (`exttyp`) are recognised as indexable: `'FEI1'` and `'FEI2'`. Other types may be added in future.

2.2.3 Voxel size

The voxel (or pixel) size in the file can be accessed using the *voxel_size* attribute, which returns a `numpy record array` with three fields, `x`, `y` and `z`, for the voxel size in each dimension:

```
>>> with mrcfile.open('tmp.mrc') as mrc:
...     mrc.voxel_size
...
rec.array((0.,  0.,  0.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])
```

In a new file, the voxel size is zero by default. To set the voxel size, you can assign to the *voxel_size* attribute, using a single number (for an isotropic voxel size), a 3-tuple or a single-item record array with `x`, `y` and `z` fields (which must be in that order):

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # Set a new isotropic voxel size:
>>> mrc.voxel_size = 1.0
>>> mrc.voxel_size
rec.array((1.,  1.,  1.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])

>>> # Set an anisotropic voxel size using a tuple:
>>> mrc.voxel_size = (1.0, 2.0, 3.0)
>>> mrc.voxel_size
rec.array((1.,  2.,  3.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])
```

(continues on next page)

(continued from previous page)

```
>>> # And set a different anisotropic voxel size using a record array:
>>> mrc.voxel_size = np.rec.array(( 4.,  5.,  6.), dtype=[('x', '<f4'), ('y', '<f4'), ('z
↪', '<f4')])
>>> mrc.voxel_size
rec.array((4.,  5.,  6.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])
>>> mrc.close()
```

The sizes are not stored directly in the MRC header, but are calculated when required from the header's cell and grid size fields. The voxel size can therefore be changed by altering the cell size:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # Check the current voxel size in X:
>>> mrc.voxel_size.x
array(4., dtype=float32)

>>> # And check the current cell dimensions:
>>> mrc.header.cella
rec.array((20.,  10.,  6.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])

>>> # Now change the cell's X length:
>>> mrc.header.cella.x = 10

>>> # And we see the voxel size has also changed:
>>> mrc.voxel_size.x
array(2., dtype=float32)

>>> mrc.close()
```

Equivalently, the cell size will be changed if a new voxel size is given:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # Check the current cell dimensions:
>>> mrc.header.cella
rec.array((10.,  10.,  6.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])

>>> # Set a new voxel size:
>>> mrc.voxel_size = 1.0

>>> # And our cell size has been updated:
>>> mrc.header.cella
rec.array((5.,  2.,  1.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])

>>> mrc.close()
```

Because the voxel size array is calculated on demand, assigning back to it wouldn't work so it's flagged as read-only:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # This doesn't work
>>> mrc.voxel_size.x = 2.0
Traceback (most recent call last):
  ...
ValueError: assignment destination is read-only

>>> # But you can do this
>>> vsize = mrc.voxel_size.copy()
>>> vsize.x = 2.0
>>> mrc.voxel_size = vsize
>>> mrc.voxel_size
rec.array((2.,  1.,  1.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])
>>> mrc.close()
```

Note that the calculated voxel size will change if the grid size is changed by replacing the data array:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # Check the current voxel size:
>>> mrc.voxel_size
rec.array((2.,  1.,  1.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])
>>> # And the current data dimensions:
>>> mrc.data.shape
(2, 2, 5)

>>> # Replace the data with an array with a different shape:
>>> mrc.set_data(example_data)
>>> mrc.data.shape
(3, 4)

>>> # ...and the voxel size has changed:
>>> mrc.voxel_size
rec.array((2.5, 0.6666667, 1.),
          dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])

>>> mrc.close()
```

2.2.4 Keeping the header and data in sync

When a new data array is given (using *set_data()* or the `data` argument to *mrcfile.new()*), the header is automatically updated to ensure the file is is valid:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # Check the current data shape and header dimensions match
>>> mrc.data.shape
(3, 4)
>>> mrc.header.nx
```

(continues on next page)

(continued from previous page)

```
array(4, dtype=int32)
>>> mrc.header.nx == mrc.data.shape[-1]  # X axis is always the last in shape
True

>>> # Let's also check the maximum value recorded in the header
>>> mrc.header.dmax
array(11., dtype=float32)
>>> mrc.header.dmax == mrc.data.max()
True

>>> # Now set a data array with a different shape, and check the header again
>>> mrc.set_data(data_3d)
>>> mrc.data.shape
(2, 2, 5)
>>> mrc.header.nx
array(5, dtype=int32)
>>> mrc.header.nx == mrc.data.shape[-1]
True

>>> # The data statistics are updated as well
>>> mrc.header.dmax
array(1000., dtype=float32)
>>> mrc.header.dmax == mrc.data.max()
True
>>> mrc.close()
```

If the data array is modified in place, for example by editing values or changing the shape or dtype attributes, the header will no longer be correct:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> mrc.data.shape
(2, 2, 5)

>>> # Change the data shape in-place and check the header
>>> mrc.data.shape = (5, 4)
>>> mrc.header.nx == mrc.data.shape[-1]
False

>>> # We'll also change some values and check the data statistics
>>> mrc.data[2:] = 0
>>> mrc.data.max()
0
>>> mrc.header.dmax == mrc.data.max()
False
>>> mrc.close()
```

Note that the header is deliberately not updated automatically except when `set_data()` is called, so if you need to override any of the automatic header values you can do.

To keep the header in sync with the data, three methods can be used to update the header:

- `update_header_from_data()`: This updates the header's dimension fields, mode, space group and machine stamp to be consistent with the data array. Because it only inspects the data array's attributes, this method is fast even for very large arrays.

- *update_header_stats()*: This updates the data statistics fields in the header (dmin, dmax, dmean and rms). This method can be slow with large data arrays because it has to access the full contents of the array.
- *reset_header_stats()*: If the data values have changed and the statistics fields are invalid, but the data array is very large and you do not want to wait for `update_header_stats()` to run, you can call this method to reset the header's statistics fields to indicate that the values are undetermined.

The file we just saved had an invalid header, but of course, that's what's used by `mrcfile` to work out how to read the file from disk! When we open the file again, our change to the shape has disappeared:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')
>>> mrc.data.shape
(2, 2, 5)

>>> # Let's change the shape again, as we did before
>>> mrc.data.shape = (5, 4)
>>> mrc.header.nx == mrc.data.shape[-1]
False

>>> # Now let's update the dimensions:
>>> mrc.update_header_from_data()
>>> mrc.header.nx
array(4, dtype=int32)
>>> mrc.header.nx == mrc.data.shape[-1]
True

>>> # The data statistics are still incorrect:
>>> mrc.header.dmax
array(1000., dtype=float32)
>>> mrc.header.dmax == mrc.data.max()
False

>>> # So let's update those as well:
>>> mrc.update_header_stats()
>>> mrc.header.dmax
array(0., dtype=float32)
>>> mrc.header.dmax == mrc.data.max()
True
>>> mrc.close()
```

In general, if you're changing the shape, type or endianness of the data, it's easiest to use *set_data()* and the header will be kept up to date for you. If you start changing values in the data, remember that the statistics in the header will be out of date until you call *update_header_stats()* or *reset_header_stats()*.

2.2.5 Data dimensionality

MRC files can be used to store several types of data: single images, image stacks, volumes and volume stacks. These are distinguished by the dimensionality of the data array and the space group number (the header's `ispg` field):

Data type	Dimensions	Space group
Single image	2	0
Image stack	3	0
Volume	3	1–230 (1 for normal EM data)
Volume stack	4	401–630 (401 for normal EM data)

MrcFile objects have methods to allow easy identification of the data type: *is_single_image()*, *is_image_stack()*, *is_volume()* and *is_volume_stack()*.

```
>>> mrc = mrcfile.open('tmp.mrc')

>>> # The file currently contains two-dimensional data
>>> mrc.data.shape
(5, 4)
>>> len(mrc.data.shape)
2

>>> # This is intepreted as a single image
>>> mrc.is_single_image()
True
>>> mrc.is_image_stack()
False
>>> mrc.is_volume()
False
>>> mrc.is_volume_stack()
False

>>> mrc.close()
```

If a file already contains image or image stack data, new three-dimensional data is treated as an image stack; otherwise, 3D data is treated as a volume by default:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # New 3D data in an existing image file is treated as an image stack:
>>> mrc.set_data(data_3d)
>>> len(mrc.data.shape)
3
>>> mrc.is_volume()
False
>>> mrc.is_image_stack()
True
>>> int(mrc.header.ispg)
0
>>> mrc.close()

>>> # But normally, 3D data is treated as a volume:
>>> mrc = mrcfile.new('tmp.mrc', overwrite=True)
>>> mrc.set_data(data_3d)
>>> mrc.is_volume()
True
>>> mrc.is_image_stack()
False
>>> int(mrc.header.ispg)
1
>>> mrc.close()
```

Call *set_image_stack()* and *set_volume()* to change the interpretation of 3D data. (Note: as well as changing `ispg`, these methods also change `mz` to be 1 for image stacks and equal to `nz` for volumes.)

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # Change the file to represent an image stack:
>>> mrc.set_image_stack()
>>> mrc.is_volume()
False
>>> mrc.is_image_stack()
True
>>> int(mrc.header.ispg)
0

>>> # And now change it back to representing a volume:
>>> mrc.set_volume()
>>> mrc.is_volume()
True
>>> mrc.is_image_stack()
False
>>> int(mrc.header.ispg)
1

>>> mrc.close()
```

Note that the MRC format allows the data axes to be swapped using the header's `mapc`, `mapr` and `maps` fields. This library does not attempt to swap the axes and simply assigns the columns to X, rows to Y and sections to Z. (The data array is indexed in C style, so data values can be accessed using `mrc.data[z][y][x]`.) In general, EM data is written using the default axes, but crystallographic data files might use swapped axes in certain space groups – if this might matter to you, you should check the `mapc`, `mapr` and `maps` fields after opening the file and consider transposing the data array if necessary.

2.2.6 Data types

Various numpy data types can be used for MRC data arrays. The conversions to MRC mode numbers are:

Data type	MRC mode
float16	12 (see note below)
float32	2
int8	0
int16	1
uint8	6 (note that data will be widened to 16 bits in the file)
uint16	6
complex64	4

(Mode 3 and the proposed 4-bit mode 101 are not supported since there are no corresponding numpy dtypes.)

Note that mode 12 is a proposed extension to the MRC2014 format and is not yet widely supported by other software. If you need to write float16 data to MRC files in a compatible way, you should cast to float32 first and use mode 2.

No other data types are accepted, including integer types of more than 16 bits, or float types of more than 32 bits. Many numpy array creation routines use int64 or float64 dtypes by default, which means you will need to give a `dtype` argument to ensure the array can be used in an MRC file:

```
>>> mrc = mrcfile.open('tmp.mrc', mode='r+')

>>> # This does not work
>>> mrc.set_data(np.zeros((4, 5)))
Traceback (most recent call last):
  ...
ValueError: dtype 'float64' cannot be converted to an MRC file mode
>>> # But this does
>>> mrc.set_data(np.zeros((4, 5), dtype=np.int16))
>>> mrc.data
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=int16)

>>> mrc.close()
```

Warning: be careful if you have an existing numpy array in float64, int64 or int32 data types. If they try to convert them into one of the narrower types supported by `mrcfile` and they contain values outside the range of the target type, the values will silently overflow. For floating point formats this can lead to `inf` values, and with integers it can lead to entirely meaningless values. A full discussion of this issue is outside the scope of this guide; see the numpy documentation for more information.

2.3 Dealing with large files

`mrcfile` provides two ways of improving performance when handling large files: memory mapping and asynchronous (background) opening. Memory mapping treats the file's data on disk as if it is already in memory, and only actually loads the data in small chunks when it is needed. Asynchronous opening uses a separate thread to open the file, allowing the main thread to carry on with other work while the file is loaded from disk in parallel.

Which technique is better depends on what you intend to do with the file and the characteristics of your computer, and it's usually worth testing both approaches and seeing what works best for your particular task. In general, memory mapping gives better performance when dealing with a single file, particularly if the file is very large. If you need to process several files, asynchronous opening can be faster because you can work on one file while loading the next one.

2.3.1 Memory-mapped files

With very large files, it might be helpful to use the *`mrcfile.mmap()`* function to open the file, which will open the data as a `memory-mapped numpy array`. The contents of the array are only read from disk as needed, so this allows large files to be opened very quickly. Parts of the data can then be read and written by slicing the array:

```
>>> # Let's make a new file to work with (only small for this example!)
>>> mrcfile.write('maybe_large.mrc', example_data)

>>> # Open the file in memory-mapped mode
>>> mrc = mrcfile.mmap('maybe_large.mrc', mode='r+')
>>> # Now read part of the data by slicing
>>> mrc.data[1:3]
memmap([[ 4,  5,  6,  7],
        [ 8,  9, 10, 11]], dtype=int8)
```

(continues on next page)

(continued from previous page)

```
>>> # Set some values by assigning to a slice
>>> mrc.data[1:3,1:3] = 0

>>> # Read the entire array - with large files this might take a while!
>>> mrc.data[:]
memmap([[ 0,  1,  2,  3],
        [ 4,  0,  0,  7],
        [ 8,  0,  0, 11]], dtype=int8)
>>> mrc.close()
```

To create new large, empty files quickly, use the `mrcfile.new_mmap()` function. This creates an empty file with a given shape and data mode. An optional fill value can be provided but filling a very large mmap array can take a long time, so it's best to use this only when needed. If you plan to fill the array with other data anyway, it's better to leave the fill value as `None`. A typical use case would be to create a new file and then fill it slice by slice:

```
>>> # Make a new, empty memory-mapped MRC file
>>> mrc = mrcfile.new_mmap('mmap.mrc', shape=(3, 3, 4), mrc_mode=0)
>>> # Fill each slice with a different value
>>> for val in range(len(mrc.data)):
...     mrc.data[val] = val
...
>>> mrc.data[:]
memmap([[[0, 0, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 0]],

        [[1, 1, 1, 1],
         [1, 1, 1, 1],
         [1, 1, 1, 1]],

        [[2, 2, 2, 2],
         [2, 2, 2, 2],
         [2, 2, 2, 2]]], dtype=int8)
```

2.3.2 Asynchronous opening

When processing several files in a row, asynchronous (background) opening can improve performance by allowing you to open multiple files in parallel. The `mrcfile.open_async()` function starts a background thread to open a file, and returns a `FutureMrcFile` object which you can call later to get the file after it's been opened:

```
>>> # Open the first example file
>>> mrc1 = mrcfile.open('maybe_large.mrc')
>>> # Start opening the second example file before we process the first
>>> future_mrc2 = mrcfile.open_async('tmp.mrc.gz')
>>> # Now we'll do some calculations with the first file
>>> mrc1.data.sum()
36
>>> # Get the second file from its "Future" container ('result()' will wait
>>> # until the file is ready)
>>> mrc2 = future_mrc2.result()
>>> # Before we process the second file, we'll start the third one opening
```

(continues on next page)

(continued from previous page)

```
>>> future_mrc3 = mrcfile.open_async('tmp.mrc.bz2')
>>> mrc2.data.max()
22
>>> # Finally, we'll get the third file and process it
>>> mrc3 = future_mrc3.result()
>>> mrc3.data
array([[ 0,  3,  6,  9],
       [12, 15, 18, 21],
       [24, 27, 30, 33]], dtype=int8)
```

As we saw in that example, calling *result()* will give us the *MrcFile* from the file opening operation. If the file hasn't been fully opened yet, *result()* will simply wait until it's ready. To avoid waiting, call *done()* to check if it's finished.

2.4 Validating MRC files

MRC files can be validated with *mrcfile.validate()*, which prints an explanation of what is happening and also returns `True` if the file is valid or `False` otherwise:

```
>>> mrcfile.validate('tmp.mrc')
Checking if tmp.mrc is a valid MRC2014 file...
File appears to be valid.
True
```

This works equally well for gzip- or bzip2-compressed files:

```
>>> mrcfile.validate('tmp.mrc.gz')
Checking if tmp.mrc.gz is a valid MRC2014 file...
File appears to be valid.
True

>>> mrcfile.validate('tmp.mrc.bz2')
Checking if tmp.mrc.bz2 is a valid MRC2014 file...
File appears to be valid.
True
```

Errors will cause messages to be printed to the console, and *validate()* will return `False`:

```
>>> # Let's make a file which is valid except for the header's mz value
>>> with mrcfile.new('tmp.mrc', overwrite=True) as mrc:
...     mrc.set_data(example_data)
...     mrc.header.mz = -1
...

>>> # Now it should fail validation and print a helpful message
>>> mrcfile.validate('tmp.mrc')
Checking if tmp.mrc is a valid MRC2014 file...
Header field 'mz' is negative
False
```

(More serious errors might also cause warnings to be printed to `sys.stderr`.)

Normally, messages are printed to `sys.stdout` (as normal for Python `print()` calls). *validate()* has an optional `print_file` argument which allows any text stream to be used for the output instead:

```
>>> # Create a text stream to capture the output
>>> import io
>>> output = io.StringIO()

>>> # Now validate the file...
>>> mrcfile.validate('tmp.mrc', print_file=output)
False

>>> # ...and check the output separately
>>> print(output.getvalue().strip())
Checking if tmp.mrc is a valid MRC2014 file...
Header field 'mz' is negative
```

Behind the scenes, *mrcfile.validate()* opens the file in *permissive mode* using *mrcfile.open()* and then calls *MrcFile.validate()*. If you already have an *MrcFile* open, you can call its *validate()* method directly to check the file – but note that the file size test might be inaccurate unless you call *flush()* first. To ensure the file is completely valid, it's best to flush or close the file and then validate it from scratch using *mrcfile.validate()*.

If you find that a file created with this library is invalid, and you haven't altered anything in the header in a way that might cause problems, please file a bug report on the issue tracker!

2.5 Command line usage

Some `mrcfile` functionality is available directly from the command line, via scripts that are installed along with the package, or in some cases by running modules with `python -m`.

(If you've downloaded the source code instead of installing via `pip`, run `pip install <path-to-mrcfile>` or `python setup.py install` to make the command line scripts available.)

2.5.1 Validation

MRC files can be validated with the `mrcfile-validate` script:

```
$ mrcfile-validate tests/test_data/EMD-3197.map
Checking if tests/test_data/EMD-3197.map is a valid MRC2014 file...
File does not declare MRC format version 20140 or 20141: nversion = 0

$ # Exit status is 1 if file is invalid
$ echo $?
1
```

This script wraps the *mrcfile.validator* module, which can also be called directly:

```
$ python -m mrcfile.validator valid_file.mrc
Checking if valid_file.mrc is a valid MRC2014 file...
File appears to be valid.
$ echo $?
0
```

Multiple file names can be passed to either form of the command, and because these commands call `mrcfile.validate()` behind the scenes, gzip- and bzip2-compressed files can be validated as well:

```
$ mrcfile-validate valid_file_1.mrc valid_file_2.mrc.gz valid_file_3.mrc.bz2
Checking if valid_file_1.mrc is a valid MRC2014 file...
File appears to be valid.
Checking if valid_file_2.mrc is a valid MRC2014 file...
File appears to be valid.
Checking if valid_file_3.mrc is a valid MRC2014 file...
File appears to be valid.
```

2.5.2 Examining MRC headers

MRC file headers can be printed to the console with the `mrcfile-header` script:

```
$ mrcfile-header tests/test_data/EMD-3197.map
MRC header for tests/test_data/EMD-3197.map:
nx              : 20
ny              : 20
nz              : 20
mode            : 2
nxstart         : -2
nystart         : 0
nzstart         : 0
mx              : 20
my              : 20
mz              : 20
cella           : (228.0, 228.0, 228.0)
cellb           : (90.0, 90.0, 90.0)
...
...
```

Like `mrcfile-validate`, this also works for multiple files. If you want to access the same functionality from within Python, call `print_header()` on an open `MrcFile` object, or `mrcfile.command_line.print_headers()` with a list of file names.

2.6 API overview

2.6.1 Class hierarchy

The following classes are provided by the mrcfile.py library:

- `MrcObject`: Represents a generic MRC-like data object in memory, and provides header, extended header and data arrays and methods for operating on them.
- `MrcInterpreter`: Subclass of MrcObject that can read and/or write its MRC data from arbitrary byte I/O streams (including Python file objects).
- `MrcFile`: Subclass of MrcInterpreter that opens a file from disk to use as its I/O stream. This is the normal class used for most interactions with MRC files.
- `GzipMrcFile`: Reads and writes MRC data using compressed gzip files.
- `Bzip2MrcFile`: Reads and writes MRC data using compressed bzip2 files.

- *MrcMemmap*: Uses a memory-mapped data array, for fast random access to very large data files. MrcMemmap overrides various parts of the MrcFile implementation to ensure that the memory-mapped data array is opened, closed and moved correctly when the data or extended header array sizes are changed.

2.6.2 MrcFile attributes and methods

Attributes:

- *header*
- *extended_header*
- *indexed_extended_header*
- *data*
- *voxel_size*

Methods:

- *set_extended_header()*
- *set_data()*
- *is_single_image()*
- *is_image_stack()*
- *is_volume()*
- *is_volume_stack()*
- *set_image_stack()*
- *set_volume()*
- *update_header_from_data()*
- *update_header_stats()*
- *reset_header_stats()*
- *print_header()*
- *validate()*
- *flush()*
- *close()*

CHAPTER THREE

SOURCE DOCUMENTATION

3.1 mrcfile – Main package

3.1.1 mrcfile

A pure Python implementation of the MRC2014 file format.

For a full introduction and documentation, see http://mrcfile.readthedocs.io/

Functions

- *new()*: Create a new MRC file.
- *open()*: Open an MRC file.
- *open_async()*: Open an MRC file asynchronously.
- *mmap()*: Open a memory-mapped MRC file (fast for large files).
- *new_mmap()*: Create a new empty memory-mapped MRC file (fast for large files).
- *validate()*: Validate an MRC file

Basic usage

Examples assume that this package has been imported as `mrcfile` and numpy has been imported as `np`.

To open an MRC file and read a slice of data:

```
>>> with mrcfile.open('tests/test_data/EMD-3197.map') as mrc:
...     mrc.data[10,10]
...
array([ 2.58179283,  3.1406002 ,  3.64495397,  3.63812137,  3.61837363,
        4.0115056 ,  3.66981959,  2.07317996,  0.1251585 , -0.87975615,
        0.12517013,  2.07319379,  3.66982722,  4.0115037 ,  3.61837196,
        3.6381247 ,  3.64495087,  3.14059472,  2.58178973,  1.92690361], dtype=float32)
```

To create a new file with a 2D data array, and change some values:

```
>>> with mrcfile.new('tmp.mrc') as mrc:
...     mrc.set_data(np.zeros((5, 5), dtype=np.int8))
...     mrc.data[1:4,1:4] = 10
```

(continues on next page)

(continued from previous page)

```
...     mrc.data
...
array([[ 0,  0,  0,  0,  0],
       [ 0, 10, 10, 10,  0],
       [ 0, 10, 10, 10,  0],
       [ 0, 10, 10, 10,  0],
       [ 0,  0,  0,  0,  0]], dtype=int8)
```

Background

The MRC2014 format was described in the Journal of Structural Biology: http://dx.doi.org/10.1016/j.jsb.2015.04.002

The format specification is available on the CCP-EM website: http://www.ccpem.ac.uk/mrc_format/mrc2014.php

Members

`mrcfile.new`(*name*, *data=None*, *compression=None*, *overwrite=False*)

Create a new MRC file.

Parameters

- **`name`** – The file name to use, as a string or `Path`.
- **`data`** – Data to put in the file, as a `numpy array`. The default is `None`, to create an empty file.
- **`compression`** – The compression format to use. Acceptable values are: `None` (the default; for no compression), `'gzip'` or `'bzip2'`. It's good practice to name compressed files with an appropriate extension (for example, `.mrc.gz` for gzip) but this is not enforced.
- **`overwrite`** – Flag to force overwriting of an existing file. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised.

Returns

An *`MrcFile`* object (or a subclass of it if `compression` is specified).

Raises

- **`ValueError`** – If the file already exists and overwrite is `False`.
- **`ValueError`** – If the compression format is not recognised.

Warns

RuntimeWarning – If the data array contains Inf or NaN values.

`mrcfile.open`(*name*, *mode='r'*, *permissive=False*, *header_only=False*)

Open an MRC file.

This function opens both normal and compressed MRC files. Supported compression formats are: gzip, bzip2.

It is possible to use this function to create new MRC files (using mode `w+`) but the *`new()`* function is more flexible.

This function offers a permissive read mode for attempting to open corrupt or invalid files. In permissive mode, `warnings` are issued instead of exceptions if problems with the file are encountered. See *`MrcInterpreter`* or the *usage guide* for more information.

Parameters

- **`name`** – The file name to open, as a string or `Path`.

- **mode** – The file mode to use. This should be one of the following: `r` for read-only, `r+` for read and write, or `w+` for a new empty file. The default is `r`.
- **permissive** – Read the file in permissive mode. The default is `False`.
- **header_only** – Only read the header (and extended header) from the file. The default is `False`.

Returns
An *MrcFile* object (or a *GzipMrcFile* object if the file is gzipped).

Raises

- `ValueError` – If the mode is not one of `r`, `r+` or `w+`.
- `ValueError` – If the file is not a valid MRC file and `permissive` is `False`.
- `ValueError` – If the mode is `w+` and the file already exists. (Call *new()* with `overwrite=True` to deliberately overwrite an existing file.)
- `OSError` – If the mode is `r` or `r+` and the file does not exist.

Warns

- **RuntimeWarning** – If the file appears to be a valid MRC file but the data block is longer than expected from the dimensions in the header.
- **RuntimeWarning** – If the file is not a valid MRC file and `permissive` is `True`.
- **RuntimeWarning** – If the header's `exttyp` field is set to a known value but the extended header's size is not a multiple of the number of bytes in the corresponding dtype.

`mrcfile.`**`read`**(*name*)

Read an MRC file's data into a numpy array.

This is a convenience function to read the data from an MRC file when there is no need for the file's header information. To read the headers as well, or if you need access to an *MrcFile* object representing the file, use *mrcfile.open()* instead.

Parameters
name – The file name to read, as a string or `Path`.

Returns
A `numpy array` containing the data from the file.

`mrcfile.`**`write`**(*name*, *data=None*, *overwrite=False*, *voxel_size=None*)

Write a new MRC file.

This is a convenience function to allow data to be quickly written to a file (with optional compression) using just a single function call. However, there is no control over the file's metadata except for optionally setting the voxel size. For more control, or if you need access to an *MrcFile* object representing the new file, use *mrcfile.new()* instead.

Parameters

- **name** – The file name to use, as a string or `Path`. If the name ends with `.gz` or `.bz2`, the file will be compressed using gzip or bzip2 respectively.
- **data** – Data to put in the file, as a `numpy array`. The default is `None`, to create an empty file.
- **overwrite** – Flag to force overwriting of an existing file. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised.
- **voxel_size** – float | 3-tuple The voxel size to be written in the file header.

Raises

`ValueError` – If the file already exists and overwrite is `False`.

Warns

RuntimeWarning – If the data array contains Inf or NaN values.

`mrcfile.open_async`(*name*, *mode='r'*, *permissive=False*)

Open an MRC file asynchronously in a separate thread.

This allows a file to be opened in the background while the main thread continues with other work. This can be a good way to improve performance if the main thread is busy with intensive computation, but will be less effective if the main thread is itself busy with disk I/O.

Multiple files can be opened in the background simultaneously. However, this implementation is relatively crude; each call to this function will start a new thread and immediately use it to start opening a file. If you try to open many large files at the same time, performance will decrease as all of the threads attempt to access the disk at once. You'll also risk running out of memory to store the data from all the files.

This function returns a `FutureMrcFile` object, which deliberately mimics the API of the `Future` object from Python 3's `concurrent.futures` module. (Future versions of this library might return genuine `Future` objects instead.)

To get the real `MrcFile` object from a `FutureMrcFile`, call `result()`. This will block until the file has been read and the `MrcFile` object is ready. To check if the `MrcFile` is ready without blocking, call `running()` or `done()`.

Parameters

- **name** – The file name to open, as a string or `Path`.
- **mode** – The file mode (one of `r`, `r+` or `w+`).
- **permissive** – Read the file in permissive mode. The default is `False`.

Returns

A `FutureMrcFile` object.

`mrcfile.mmap`(*name*, *mode='r'*, *permissive=False*)

Open a memory-mapped MRC file.

This allows much faster opening of large files, because the data is only accessed on disk when a slice is read or written from the data array. See the `MrcMemmap` class documentation for more information.

Because the memory-mapped data array accesses the disk directly, compressed files cannot be opened with this function. In all other ways, `mmap()` behaves in exactly the same way as `open()`. The `MrcMemmap` object returned by this function can be used in exactly the same way as a normal `MrcFile` object.

Parameters

- **name** – The file name to open, as a string or `Path`.
- **mode** – The file mode (one of `r`, `r+` or `w+`).
- **permissive** – Read the file in permissive mode. The default is `False`.

Returns

An `MrcMemmap` object.

`mrcfile.new_mmap`(*name*, *shape*, *mrc_mode=0*, *fill=None*, *overwrite=False*, *extended_header=None*, *exttyp=None*)

Create a new, empty memory-mapped MRC file.

This function is useful for creating very large files. The initial contents of the data array can be set with the `fill` parameter if needed, but be aware that filling a large array can take a long time.

If `fill` is not set, the new data array's contents are unspecified and system-dependent. (Some systems fill a new empty mmap with zeros, others fill it with the bytes from the disk at the newly-mapped location.) If you are definitely going to fill the entire array with new data anyway you can safely leave `fill` as `None`, otherwise it is advised to use a sensible fill value (or ensure you are on a system that fills new mmaps with a reasonable default value).

Parameters

- **`name`** – The file name to use, as a string or `Path`.
- **`shape`** – The shape of the data array to open, as a 2-, 3- or 4-tuple of ints. For example, `(nz, ny, nx)` for a new 3D volume, or `(ny, nx)` for a new 2D image.
- **`mrc_mode`** – The MRC mode to use for the new file. One of 0, 1, 2, 4 or 6, which correspond to numpy dtypes as follows:
  - mode 0 -> int8
  - mode 1 -> int16
  - mode 2 -> float32
  - mode 4 -> complex64
  - mode 6 -> uint16

  The default is 0.
- **`fill`** – An optional value to use to fill the new data array. If `None`, the data array will not be filled and its contents are unspecified. Numpy's usual rules for rounding or rejecting values apply, according to the dtype of the array.
- **`overwrite`** – Flag to force overwriting of an existing file. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised.
- **`extended_header`** – The extended header object
- **`exttyp`** – The extended header type

Returns

A new *`MrcMemmap`* object.

Raises

- **`ValueError`** – If the MRC mode is invalid.
- **`ValueError`** – If the file already exists and overwrite is `False`.

`mrcfile.`**`validate`**`(`*name*, *print_file=None*`)`

Validate an MRC file.

This function first opens the file by calling *`open()`* (with `permissive=True`), then calls *`validate()`*, which runs a series of tests to check whether the file complies with the MRC2014 format specification.

If the file is completely valid, this function returns `True`, otherwise it returns `False`. Messages explaining the validation result will be printed to `sys.stdout` by default, but if a text stream is given (using the `print_file` argument) output will be printed to that instead.

Badly invalid files will also cause `warning` messages to be issued, which will be written to `sys.stderr` by default. See the documentation of the `warnings` module for information on how to suppress or capture warning output.

Because the file is opened by calling *`open()`*, gzip- and bzip2-compressed MRC files can be validated easily using this function.

After the file has been opened, it is checked for problems. The tests are:

1. MRC format ID string: The `map` field in the header should contain "MAP ".
2. Machine stamp: The machine stamp should contain one of `0x44 0x44 0x00 0x00`, `0x44 0x41 0x00 0x00` or `0x11 0x11 0x00 0x00`.
3. MRC mode: the `mode` field should be one of the supported mode numbers: 0, 1, 2, 4, 6 or 12. (Note that MRC modes 3 and 101 are also valid according to the MRC 2014 specification but are not supported by mrcfile.)
4. Map and cell dimensions: The header fields `nx`, `ny`, `nz`, `mx`, `my`, `mz`, `cella.x`, `cella.y` and `cella.z` must all be positive numbers.
5. Axis mapping: Header fields `mapc`, `mapr` and `maps` must contain the values 1, 2, and 3 (in any order).
6. Volume stack dimensions: If the spacegroup is in the range 401–630, representing a volume stack, the `nz` field should be exactly divisible by `mz` to represent the number of volumes in the stack.
7. Header labels: The `nlabl` field should be set to indicate the number of labels in use, and the labels in use should appear first in the label array.
8. MRC format version: The `nversion` field should be 20140 or 20141 for compliance with the MRC2014 standard.
9. Extended header type: If an extended header is present, the `exttyp` field should be set to indicate the type of extended header.
10. Data statistics: The statistics in the header should be correct for the actual data in the file, or marked as undetermined.
11. File size: The size of the file on disk should match the expected size calculated from the MRC header.

Parameters

- **`name`** – The file name to open and validate.
- **`print_file`** – The output text stream to use for printing messages about the validation. This is passed directly to the `file` argument of Python's `print()` function. The default is `None`, which means output will be printed to `sys.stdout`.

Returns

`True` if the file is valid, or `False` if the file does not meet the MRC format specification in any way.

Raises

`OSError` – If the file does not exist or cannot be opened.

Warns

RuntimeWarning – If the file is seriously invalid because it has no map ID string, an incorrect machine stamp, an unknown mode number, or is not the same size as expected from the header.

3.2 Submodules

3.3 mrcfile.bzip2mrcfile module

3.3.1 bzip2mrcfile

Module which exports the *`Bzip2MrcFile`* class.

Classes:

Bzip2MrcFile: An object which represents a bzip2-compressed MRC file.

class mrcfile.bzip2mrcfile.**Bzip2MrcFile**(*name*, *mode='r'*, *overwrite=False*, *permissive=False*, *header_only=False*, ***kwargs*)

Bases: *MrcFile*

MrcFile subclass for handling bzip2-compressed files.

Usage is the same as for *MrcFile*.

_open_file(*name*)

Override _open_file() to open a bzip2 file.

_read(*header_only=False*)

Override _read() to ensure bzip2 file is in read mode.

_ensure_readable_bzip2_stream()

Make sure _iostream is a bzip2 stream that can be read.

_get_file_size()

Override _get_file_size() to ensure stream is readable first.

_read_bytearray_from_stream(*number_of_bytes*)

Override because BZ2File in Python 2 does not support `readinto()`.

flush()

Override *flush()* since BZ2File objects need special handling.

3.4 mrcfile.command_line module

3.4.1 command_line

Module for functions used as command line entry points.

The names of the corresponding command line scripts can be found in the `entry_points` section of `setup.py`.

mrcfile.command_line.**print_headers**(*names=None*, *print_file=None*)

Print the MRC header contents from a list of files.

This function opens files in permissive mode to allow headers of invalid files to be examined.

Parameters

- **names** – A list of file names. If not given or `None`, the names are taken from the command line arguments.
- **print_file** – The output text stream to use for printing the headers. This is passed directly to the `print_file` argument of *print_header()*. The default is `None`, which means output will be printed to `sys.stdout`.

3.5 mrcfile.constants module

3.5.1 constants

Constants used by the `mrcfile.py` library.

3.6 mrcfile.dtypes module

3.6.1 dtypes

numpy dtypes used by the `mrcfile.py` library.

The dtypes are defined in a separate module because they do not interact nicely with the `from __future__ import unicode_literals` feature used in the rest of the package.

`mrcfile.dtypes.`**`get_ext_header_dtype`**(*exttyp*, *byte_order='='*)

Get a dtype for an extended header.

Parameters

- **`exttyp`** – One of `b'FEI1'` or `b'FEI2'`, which are currently the only supported extended header types.
- **`byte_order`** – One of =, < or >.

Returns

A `numpy dtype` object for the extended header, or `None`

Raises

`ValueError` – If `byte_order` is not one of =, < or >.

3.7 mrcfile.future_mrcfile module

3.7.1 future_mrcfile

Module which exports the *`FutureMrcFile`* class.

Classes:

`FutureMrcFile`**: An object which represents an MRC file being**
opened asynchronously.

`class` `mrcfile.future_mrcfile.`**`FutureMrcFile`**(*open_function*, *args=()*, *kwargs={}*)

Bases: `object`

Object representing an MRC file being opened asynchronously.

This API deliberately mimics a `Future` object from the `concurrent.futures` module in Python 3.2+ (which we do not use directly because this code still needs to run in Python 2.7).

`__init__`(*open_function*, *args=()*, *kwargs={}*)

Initialise a new *`FutureMrcFile`* object.

This constructor starts a new thread which will invoke the callable given in `open_function` with the given arguments.

Parameters

- **open_function** – The callable to use to open the MRC file. (This will normally be *mrcfile.open()*, but could also be *MrcFile* or any of its subclasses.)
- **args** – A tuple of positional arguments to use when `open_function` is called. (Normally a 1-tuple containing the name of the file to open.)
- **kwargs** – A dictionary of keyword arguments to use when `open_function` is called.

_run(**args*, ***kwargs*)

Call the open function and store the result in the holder list.

(For internal use only.)

cancel()

Return `False`.

(See `concurrent.futures.Future.cancel()` for more details. This implementation does not allow jobs to be cancelled.)

cancelled()

Return `False`.

(See `concurrent.futures.Future.cancelled()` for more details. This implementation does not allow jobs to be cancelled.)

running()

Return `True` if the *MrcFile* is currently being opened.

(See `concurrent.futures.Future.running()` for more details.)

done()

Return `True` if the file opening has finished.

(See `concurrent.futures.Future.done()` for more details.)

result(*timeout=None*)

Return the *MrcFile* that has been opened.

(See `concurrent.futures.Future.result()` for more details.)

Parameters

timeout – Time to wait (in seconds) for the file opening to finish. If `timeout` is not specified or is `None`, there is no limit to the wait time.

Returns

An *MrcFile* object (or one of its subclasses).

Raises

- **RuntimeError** – If the operation has not finished within the time limit set by `timeout`. (Note that the type of this exception will change in future if this class is replaced by `concurrent.futures.Future`.)
- **Exception** – Any exception raised by the *MrcFile* opening operation will be re-raised here.

exception(*timeout=None*)

Return the exception raised by the file opening operation.

(See `concurrent.futures.Future.exception()` for more details.)

Parameters
`timeout` – Time to wait (in seconds) for the operation to finish. If `timeout` is not specified or is `None`, there is no limit to the wait time.

Returns
An `Exception`, if one was raised by the file opening operation, or `None` if no exception was raised.

Raises
`RuntimeError` – If the operation has not finished within the time limit set by `timeout`. (Note that the type of this exception will change in future if this class is replaced by `concurrent.futures.Future`.)

`_get_result`(*timeout*)

Return the result or exception from the file opening operation.

(For internal use only.)

`add_done_callback`(*fn*)

Not implemented.

(See `concurrent.futures.Future.add_done_callback()` for more details.)

3.8 mrcfile.gzipmrcfile module

3.8.1 gzipmrcfile

Module which exports the *`GzipMrcFile`* class.

Classes:
`GzipMrcFile`: An object which represents a gzipped MRC file.

class mrcfile.gzipmrcfile.**GzipMrcFile**(*name*, *mode='r'*, *overwrite=False*, *permissive=False*, *header_only=False*, ***kwargs*)

Bases: *`MrcFile`*

`MrcFile` subclass for handling gzipped files.

Usage is the same as for *`MrcFile`*.

`_open_file`(*name*)

Override _open_file() to open both normal and gzip files.

`_close_file`()

Override _close_file() to close both normal and gzip files.

`_read`(*header_only=False*)

Override _read() to ensure gzip file is in read mode.

`_ensure_readable_gzip_stream`()

Make sure _iostream is a gzip stream that can be read.

`_get_file_size`()

Override _get_file_size() to avoid seeking from end.

`flush`()

Override *`flush()`* since GzipFile objects need special handling.

3.9 mrcfile.load_functions module

3.9.1 load_functions

Module for top-level functions that open MRC files and form the main API of the package.

`mrcfile.load_functions.`**`new`**(*name*, *data=None*, *compression=None*, *overwrite=False*)

Create a new MRC file.

Parameters

- **`name`** – The file name to use, as a string or `Path`.
- **`data`** – Data to put in the file, as a `numpy array`. The default is `None`, to create an empty file.
- **`compression`** – The compression format to use. Acceptable values are: `None` (the default; for no compression), `'gzip'` or `'bzip2'`. It's good practice to name compressed files with an appropriate extension (for example, `.mrc.gz` for gzip) but this is not enforced.
- **`overwrite`** – Flag to force overwriting of an existing file. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised.

Returns

An *`MrcFile`* object (or a subclass of it if `compression` is specified).

Raises

- **`ValueError`** – If the file already exists and overwrite is `False`.
- **`ValueError`** – If the compression format is not recognised.

Warns

RuntimeWarning – If the data array contains Inf or NaN values.

`mrcfile.load_functions.`**`open`**(*name*, *mode='r'*, *permissive=False*, *header_only=False*)

Open an MRC file.

This function opens both normal and compressed MRC files. Supported compression formats are: gzip, bzip2.

It is possible to use this function to create new MRC files (using mode `w+`) but the *`new()`* function is more flexible.

This function offers a permissive read mode for attempting to open corrupt or invalid files. In permissive mode, `warnings` are issued instead of exceptions if problems with the file are encountered. See *`MrcInterpreter`* or the *usage guide* for more information.

Parameters

- **`name`** – The file name to open, as a string or `Path`.
- **`mode`** – The file mode to use. This should be one of the following: `r` for read-only, `r+` for read and write, or `w+` for a new empty file. The default is `r`.
- **`permissive`** – Read the file in permissive mode. The default is `False`.
- **`header_only`** – Only read the header (and extended header) from the file. The default is `False`.

Returns

An *`MrcFile`* object (or a *`GzipMrcFile`* object if the file is gzipped).

Raises

- **`ValueError`** – If the mode is not one of `r`, `r+` or `w+`.

- `ValueError` – If the file is not a valid MRC file and `permissive` is `False`.
- `ValueError` – If the mode is `w+` and the file already exists. (Call *`new()`* with `overwrite=True` to deliberately overwrite an existing file.)
- `OSError` – If the mode is `r` or `r+` and the file does not exist.

Warns

- **RuntimeWarning** – If the file appears to be a valid MRC file but the data block is longer than expected from the dimensions in the header.
- **RuntimeWarning** – If the file is not a valid MRC file and `permissive` is `True`.
- **RuntimeWarning** – If the header's `exttyp` field is set to a known value but the extended header's size is not a multiple of the number of bytes in the corresponding dtype.

`mrcfile.load_functions.`**`read`**(*name*)

Read an MRC file's data into a numpy array.

This is a convenience function to read the data from an MRC file when there is no need for the file's header information. To read the headers as well, or if you need access to an *`MrcFile`* object representing the file, use *`mrcfile.open()`* instead.

Parameters

`name` – The file name to read, as a string or `Path`.

Returns

A `numpy array` containing the data from the file.

`mrcfile.load_functions.`**`write`**(*name*, *data=None*, *overwrite=False*, *voxel_size=None*)

Write a new MRC file.

This is a convenience function to allow data to be quickly written to a file (with optional compression) using just a single function call. However, there is no control over the file's metadata except for optionally setting the voxel size. For more control, or if you need access to an *`MrcFile`* object representing the new file, use *`mrcfile.new()`* instead.

Parameters

- **`name`** – The file name to use, as a string or `Path`. If the name ends with `.gz` or `.bz2`, the file will be compressed using gzip or bzip2 respectively.
- **`data`** – Data to put in the file, as a `numpy array`. The default is `None`, to create an empty file.
- **`overwrite`** – Flag to force overwriting of an existing file. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised.
- **`voxel_size`** – float | 3-tuple The voxel size to be written in the file header.

Raises

`ValueError` – If the file already exists and overwrite is `False`.

Warns

RuntimeWarning – If the data array contains Inf or NaN values.

`mrcfile.load_functions.`**`open_async`**(*name*, *mode='r'*, *permissive=False*)

Open an MRC file asynchronously in a separate thread.

This allows a file to be opened in the background while the main thread continues with other work. This can be a good way to improve performance if the main thread is busy with intensive computation, but will be less effective if the main thread is itself busy with disk I/O.

Multiple files can be opened in the background simultaneously. However, this implementation is relatively crude; each call to this function will start a new thread and immediately use it to start opening a file. If you try to open many large files at the same time, performance will decrease as all of the threads attempt to access the disk at once. You'll also risk running out of memory to store the data from all the files.

This function returns a `FutureMrcFile` object, which deliberately mimics the API of the `Future` object from Python 3's `concurrent.futures` module. (Future versions of this library might return genuine `Future` objects instead.)

To get the real `MrcFile` object from a `FutureMrcFile`, call `result()`. This will block until the file has been read and the `MrcFile` object is ready. To check if the `MrcFile` is ready without blocking, call `running()` or `done()`.

Parameters

- **`name`** – The file name to open, as a string or `Path`.
- **`mode`** – The file mode (one of `r`, `r+` or `w+`).
- **`permissive`** – Read the file in permissive mode. The default is `False`.

Returns
A `FutureMrcFile` object.

`mrcfile.load_functions.`**`mmap`**`(`*name*, *mode='r'*, *permissive=False*`)`

Open a memory-mapped MRC file.

This allows much faster opening of large files, because the data is only accessed on disk when a slice is read or written from the data array. See the `MrcMemmap` class documentation for more information.

Because the memory-mapped data array accesses the disk directly, compressed files cannot be opened with this function. In all other ways, `mmap()` behaves in exactly the same way as `open()`. The `MrcMemmap` object returned by this function can be used in exactly the same way as a normal `MrcFile` object.

Parameters

- **`name`** – The file name to open, as a string or `Path`.
- **`mode`** – The file mode (one of `r`, `r+` or `w+`).
- **`permissive`** – Read the file in permissive mode. The default is `False`.

Returns
An `MrcMemmap` object.

`mrcfile.load_functions.`**`new_mmap`**`(`*name*, *shape*, *mrc_mode=0*, *fill=None*, *overwrite=False*, *extended_header=None*, *exttyp=None*`)`

Create a new, empty memory-mapped MRC file.

This function is useful for creating very large files. The initial contents of the data array can be set with the `fill` parameter if needed, but be aware that filling a large array can take a long time.

If `fill` is not set, the new data array's contents are unspecified and system-dependent. (Some systems fill a new empty mmap with zeros, others fill it with the bytes from the disk at the newly-mapped location.) If you are definitely going to fill the entire array with new data anyway you can safely leave `fill` as `None`, otherwise it is advised to use a sensible fill value (or ensure you are on a system that fills new mmaps with a reasonable default value).

Parameters

- **`name`** – The file name to use, as a string or `Path`.
- **`shape`** – The shape of the data array to open, as a 2-, 3- or 4-tuple of ints. For example, `(nz, ny, nx)` for a new 3D volume, or `(ny, nx)` for a new 2D image.

- **mrc_mode** – The MRC mode to use for the new file. One of 0, 1, 2, 4 or 6, which correspond to numpy dtypes as follows:
  - mode 0 -> int8
  - mode 1 -> int16
  - mode 2 -> float32
  - mode 4 -> complex64
  - mode 6 -> uint16

  The default is 0.
- **fill** – An optional value to use to fill the new data array. If `None`, the data array will not be filled and its contents are unspecified. Numpy's usual rules for rounding or rejecting values apply, according to the dtype of the array.
- **overwrite** – Flag to force overwriting of an existing file. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised.
- **extended_header** – The extended header object
- **exttyp** – The extended header type

Returns
A new *MrcMemmap* object.

Raises

- `ValueError` – If the MRC mode is invalid.
- `ValueError` – If the file already exists and overwrite is `False`.

3.10 mrcfile.mrcfile module

3.10.1 mrcfile

Module which exports the *MrcFile* class.

Classes:
MrcFile: An object which represents an MRC file.

class mrcfile.mrcfile.**MrcFile**(*name*, *mode='r'*, *overwrite=False*, *permissive=False*, *header_only=False*, ***kwargs*)

Bases: *MrcInterpreter*

An object which represents an MRC file.

The header and data are handled as numpy arrays - see *MrcObject* for details.

MrcFile supports a permissive read mode for attempting to open corrupt or invalid files. See *mrcfile.mrcinterpreter.MrcInterpreter* or the *usage guide* for more information.

Usage:
To create a new MrcFile object, pass a file name and optional mode. To ensure the file is written to disk and closed correctly, it's best to use the `with` statement:

```
>>> with MrcFile('tmp.mrc', 'w+') as mrc:
...     mrc.set_data(np.zeros((10, 10), dtype=np.int8))
```

In mode r or r+, the named file is opened from disk and read. In mode w+ a new empty file is created and will be written to disk at the end of the `with` block (or when *`flush()`* or *`close()`* is called).

`__init__`(*name*, *mode='r'*, *overwrite=False*, *permissive=False*, *header_only=False*, ***kwargs*)

Initialise a new *`MrcFile`* object.

The given file name is opened in the given mode. For mode `r` or `r+` the header, extended header and data are read from the file. For mode `w+` a new file is created with a default header and empty extended header and data arrays.

Parameters

- **`name`** – The file name to open, as a string or pathlib Path.
- **`mode`** – The file mode to use. This should be one of the following: `r` for read-only, `r+` for read and write, or `w+` for a new empty file. The default is `r`.
- **`overwrite`** – Flag to force overwriting of an existing file if the mode is `w+`. If `False` and a file of the same name already exists, the file is not overwritten and an exception is raised. The default is `False`.
- **`permissive`** – Read the file in permissive mode. (See *`mrcfile.mrcinterpreter.MrcInterpreter`* for details.) The default is `False`.
- **`header_only`** – Only read the header (and extended header) from the file. The default is `False`.

Raises

- **`ValueError`** – If the mode is not one of `r`, `r+` or `w+`.
- **`ValueError`** – If the file is not a valid MRC file and `permissive` is `False`.
- **`ValueError`** – If the mode is `w+`, the file already exists and overwrite is `False`.
- **`OSError`** – If the mode is `r` or `r+` and the file does not exist.

Warns

- **RuntimeWarning** – If the file appears to be a valid MRC file but the data block is longer than expected from the dimensions in the header.
- **RuntimeWarning** – If the file is not a valid MRC file and `permissive` is `True`.
- **RuntimeWarning** – If the header's `exttyp` field is set to a known value but the extended header's size is not a multiple of the number of bytes in the corresponding dtype.

`_open_file`(*name*)

Open a file object to use as the I/O stream.

`_read`(*header_only=False*)

Override _read() to move back to start of file first.

`_read_data`()

Override _read_data() to check file size matches data block size.

`_get_file_size`()

Return the size of the underlying file object, in bytes.

`close`()

Flush any changes to disk and close the file.

This override calls *`MrcInterpreter.close()`* to ensure the stream is flushed and closed, then closes the file object.

`_close_file``()`

Close the file object.

`validate``(`*print_file=None*`)`

Validate this MRC file.

The tests are:

1. MRC format ID string: The `map` field in the header should contain "MAP ".
2. Machine stamp: The machine stamp should contain one of `0x44 0x44 0x00 0x00`, `0x44 0x41 0x00 0x00` or `0x11 0x11 0x00 0x00`.
3. MRC mode: the `mode` field should be one of the supported mode numbers: 0, 1, 2, 4, 6 or 12. (Note that MRC modes 3 and 101 are also valid according to the MRC 2014 specification but are not supported by mrcfile.)
4. Map and cell dimensions: The header fields `nx`, `ny`, `nz`, `mx`, `my`, `mz`, `cella.x`, `cella.y` and `cella.z` must all be positive numbers.
5. Axis mapping: Header fields `mapc`, `mapr` and `maps` must contain the values 1, 2, and 3 (in any order).
6. Volume stack dimensions: If the spacegroup is in the range 401–630, representing a volume stack, the `nz` field should be exactly divisible by `mz` to represent the number of volumes in the stack.
7. Header labels: The `nlabl` field should be set to indicate the number of labels in use, and the labels in use should appear first in the label array.
8. MRC format version: The `nversion` field should be 20140 or 20141 for compliance with the MRC2014 standard.
9. Extended header type: If an extended header is present, the `exttyp` field should be set to indicate the type of extended header.
10. Data statistics: The statistics in the header should be correct for the actual data in the file, or marked as undetermined.
11. File size: The size of the file on disk should match the expected size calculated from the MRC header.

Parameters

`print_file` – The output text stream to use for printing messages about the validation. This is passed directly to the `file` argument of Python's `print()` function. The default is `None`, which means output will be printed to `sys.stdout`.

Returns

`True` if the file is valid, or `False` if the file does not meet the MRC format specification in any way.

3.11 mrcfile.mrcinterpreter module

3.11.1 mrcinterpreter

Module which exports the *`MrcInterpreter`* class.

Classes:

`MrcInterpreter`: An object which can interpret an I/O stream as MRC data.

`class mrcfile.mrcinterpreter.MrcInterpreter`(*iostream=None*, *permissive=False*, *header_only=False*, ***kwargs*)

Bases: `MrcObject`

An object which interprets an I/O stream as MRC / CCP4 map data.

The header and data are handled as numpy arrays - see `MrcObject` for details.

`MrcInterpreter` can be used directly, but it is mostly intended as a superclass to provide common stream-handling functionality. This can be used by subclasses which will handle opening and closing the stream.

This class implements the `__enter__()` and `__exit__()` special methods which allow it to be used by the Python context manager in a `with` block. This ensures that `close()` is called after the object is finished with.

When reading the I/O stream, a `ValueError` is raised if the data is invalid in one of the following ways:

1. The header's `map` field is not set correctly to confirm the file type.
2. The machine stamp is invalid and so the data's byte order cannot be determined.
3. The mode number is not recognised. Currently accepted modes are 0, 1, 2, 4 and 6.
4. The file is not large enough for the specified extended header size.
5. The data block is not large enough for the specified data type and dimensions.

`MrcInterpreter` offers a permissive read mode for handling problematic files. If `permissive` is set to `True` and any of the validity checks fails, a `warning` is issued instead of an exception, and file interpretation continues. If the mode number is invalid or the data block is too small, the `data` attribute will be set to `None`. In this case, it might be possible to inspect and correct the header, and then call `_read()` again to read the data correctly. See the *usage guide* for more details.

Methods:

- `flush()`
- `close()`

Methods relevant to subclasses:

- `_read()`
- `_read_data()`
- `_read_bytearray_from_stream()`

`__init__`(*iostream=None*, *permissive=False*, *header_only=False*, ***kwargs*)

Initialise a new MrcInterpreter object.

This initialiser reads the stream if it is given. In general, subclasses should call `__init__()` without giving an `iostream` argument, then set the `_iostream` attribute themselves and call `_read()` when ready.

To use the MrcInterpreter class directly, pass a stream when creating the object (or for a write-only stream, create an MrcInterpreter with no stream, call `_create_default_attributes()` and set the `_iostream` attribute directly).

Parameters

- **`iostream`** – The I/O stream to use to read and write MRC data. The default is `None`.
- **`permissive`** – Read the stream in permissive mode. The default is `False`.
- **`header_only`** – Only read the header (and extended header) from the file. The default is `False`.

Raises

`ValueError` – If `iostream` is given, the data it contains cannot be interpreted as a valid MRC file and `permissive` is `False`.

Warns

RuntimeWarning – If `iostream` is given, the data it contains cannot be interpreted as a valid MRC file and `permissive` is `True`.

`_read`(*header_only=False*)

Read the header, extended header and data from the I/O stream.

Before calling this method, the stream should be open and positioned at the start of the header. This method will advance the stream to the end of the data block (or the end of the extended header if `header_only` is `True`.

Parameters

`header_only` – Only read the header and extended header from the stream. The default is `False`.

Raises

`ValueError` – If the data in the stream cannot be interpreted as a valid MRC file and `permissive` is `False`.

Warns

RuntimeWarning – If the data in the stream cannot be interpreted as a valid MRC file and `permissive` is `True`.

`_read_header`()

Read the MRC header from the I/O stream.

The header will be read from the current stream position, and the stream will be advanced by 1024 bytes.

Raises

`ValueError` – If the data in the stream cannot be interpreted as a valid MRC file and `permissive` is `False`.

Warns

RuntimeWarning – If the data in the stream cannot be interpreted as a valid MRC file and `permissive` is `True`.

`_read_extended_header`()

Read the extended header from the stream.

If there is no extended header, a zero-length array is assigned to the extended_header attribute.

The dtype is set as void (`'V1'`).

Raises

`ValueError` – If the stream is not long enough to contain the extended header indicated by the header and `permissive` is `False`.

Warns

RuntimeWarning – If the stream is not long enough to contain the extended header indicated by the header and `permissive` is `True`.

`_read_data`(*max_bytes=0*)

Read the data array from the stream.

This method uses information from the header to set the data array's shape and dtype.

Parameters
`max_bytes` – Read at most this many bytes from the stream. If zero or negative, the full size of the data block as defined in the header will be read, even if this is very large.

Raises
`ValueError` – If the stream is not long enough to contain the data indicated by the header and `permissive` is `False`.

Warns
RuntimeWarning – If the stream is not long enough to contain the data indicated by the header and `permissive` is `True`.

`_read_bytearray_from_stream`(*number_of_bytes*)

Read a `bytearray` from the stream.

This default implementation relies on the stream implementing the `readinto()` method to avoid copying the new array while creating the mutable `bytearray`. Subclasses should override this if their stream does not support `readinto()`.

Returns
A 2-tuple of the `bytearray` and the number of bytes that were read from the stream.

`close`()

Flush to the stream and clear the header and data attributes.

`flush`()

Flush the header and data arrays to the I/O stream.

This implementation seeks to the start of the stream, writes the header, extended header and data arrays, and then truncates the stream.

Subclasses should override this implementation for streams which do not support `seek()` or `truncate()`.

3.12 mrcfile.mrcmemmap module

3.12.1 mrcmemmap

Module which exports the *`MrcMemmap`* class.

Classes:
`MrcMemmap`: An MrcFile subclass that uses a memory-mapped data array.

class `mrcfile.mrcmemmap.`**`MrcMemmap`**(*name*, *mode='r'*, *overwrite=False*, *permissive=False*, *header_only=False*, ***kwargs*)

Bases: *`MrcFile`*

MrcFile subclass that uses a `numpy memmap array` for the data.

Using a memmap means that the disk access is done lazily: the data array will only be read or written in small chunks when required. To access the contents of the array, use the array slice operator.

Usage is the same as for *`MrcFile`*.

Note that memmap arrays use a fairly small chunk size and so performance could be poor on file systems that are optimised for infrequent large I/O operations.

If required, it is possible to create a very large empty file by creating a new MrcMemmap and then calling *`_open_memmap()`* to create the memmap array, which can then be filled slice-by-slice. Be aware that the contents of a new, empty memmap array depend on your platform: the data values could be garbage or zeros.

set_extended_header(*extended_header*)

Replace the file's extended header.

Note that the file's entire data block must be moved if the extended header size changes. Setting a new extended header can therefore be very time consuming with large files, if the new extended header occupies a different number of bytes than the previous one.

flush()

Flush the header and data arrays to the file buffer.

_read_data()

Read the data block from the file.

This method first calculates the parameters needed to read the data (block start position, endian-ness, file mode, array shape) and then opens the data as a numpy memmap array.

_open_memmap(*dtype*, *shape*)

Open a new memmap array pointing at the file's data block.

_close_data()

Delete the existing memmap array, if it exists.

The array is flagged as read-only before deletion, so if a reference to it has been kept elsewhere, changes to it should no longer be able to change the file contents.

_set_new_data(*data*)

Override of `_set_new_data()` to handle opening a new memmap and copying data into it.

3.13 mrcfile.mrcobject module

3.13.1 mrcobject

Module which exports the `MrcObject` class.

Classes:

`MrcObject`: An object representing image or volume data in the MRC format.

class mrcfile.mrcobject.**MrcObject**(***kwargs*)

Bases: `object`

An object representing image or volume data in the MRC format.

The header, extended header and data are stored as numpy arrays and exposed as read-only attributes. To replace the data or extended header, call `set_data()` or `set_extended_header()`. The header cannot be replaced but can be modified in place.

Voxel size is exposed as a writeable attribute, but is calculated on-the-fly from the header's `cella` and `mx/my/mz` fields.

Three-dimensional data can represent either a stack of 2D images, or a 3D volume. This is indicated by the header's `ispg` (space group) field, which is set to 0 for image data and >= 1 for volume data. The `is_single_image()`, `is_image_stack()`, `is_volume()` and `is_volume_stack()` methods can be used to identify the type of information stored in the data array. For 3D data, the `set_image_stack()` and `set_volume()` methods can be used to switch between image stack and volume interpretations of the data.

If the data contents have been changed, you can use the `update_header_from_data()` and `update_header_stats()` methods to make the header consistent with the data. These methods are called automatically if the data array is replaced by calling `set_data()`. `update_header_from_data()`

is fast, even with very large data arrays, because it only examines the shape and type of the data array. *update_header_stats()* calculates statistics from all items in the data array and so can be slow for very large arrays. If necessary, the *reset_header_stats()* method can be called to set the header fields to indicate that the statistics are undetermined.

Attributes:

- *header*
- *extended_header*
- *indexed_extended_header*
- *data*
- *voxel_size*
- *nstart*

Methods:

- *set_extended_header()*
- *set_data()*
- *is_single_image()*
- *is_image_stack()*
- *is_volume()*
- *is_volume_stack()*
- *set_image_stack()*
- *set_volume()*
- *update_header_from_data()*
- *update_header_stats()*
- *reset_header_stats()*
- *print_header()*
- *get_labels()*
- *add_label()*

Attributes and methods relevant to subclasses:

- `_read_only`
- *_check_writeable()*
- *_create_default_attributes()*
- *_close_data()*
- *_set_new_data()*

`__init__`(***kwargs*)

Initialise a new `MrcObject`.

This initialiser deliberately avoids creating any arrays and simply sets the header, extended header and data attributes to `None`. This allows subclasses to call *`__init__()`* at the start of their initialisers and then set the attributes themselves, probably by reading from a file, or by calling *_create_default_attributes()* for a new empty object.

Note that this behaviour might change in future: this initialiser could take optional arguments to allow the header and data to be provided by the caller, or might create the standard empty defaults rather than setting the attributes to `None`.

`_check_writeable()`

Check that this MRC object is writeable.

Raises
`ValueError` – If this object is read-only.

`_create_default_attributes()`

Set valid default values for the header and data attributes.

`_create_default_header()`

Create a default MRC file header.

The header is initialised with standard file type and version information, default values for some essential fields, and zeros elsewhere. The first text label is also set to indicate the file was created by this module.

`property header`

Get the header as a `numpy record array`.

`property extended_header`

Get the extended header as a `numpy array`.

The dtype will be void (raw data, dtype `V'`). If the actual data type of the extended header is known, the dtype of the array can be changed to match. For supported types (e.g. `'FEI1'` and `'FEI2'`), the indexed part of the extended header (excluding any zero padding) can be accessed using *`indexed_extended_header()`*.

The extended header may be modified in place. To replace it completely, call *`set_extended_header()`*.

`property indexed_extended_header`

Get the indexed part of the extended header as a `numpy array` with the appropriate dtype set.

Currently only `'FEI1'` and `'FEI2'` extended headers are supported. Modifications to the indexed extended header will not change the extended header data recorded in this *`MrcObject`*. If the extended header type is unrecognised or extended header data is not of sufficient length a warning will be produced and the indexed extended header will be None.

`set_extended_header`(*extended_header*)

Replace the extended header.

If you set the extended header you should also set the `header.exttyp` field to indicate the type of extended header.

`property data`

Get the data as a `numpy array`.

`set_data`(*data*)

Replace the data array.

This replaces the current data with the given array (or a copy of it), and updates the header to match the new data dimensions. The data statistics (min, max, mean and rms) stored in the header will also be updated.

Warns
RuntimeWarning – If the data array contains Inf or NaN values.

`_close_data()`

Close the data array.

_set_new_data(*data*)

Replace the data array with a new one.

The new data array is not checked - it must already be valid for use in an MRC file.

property voxel_size

Get or set the voxel size in angstroms.

The voxel size is returned as a structured NumPy `record array` with three fields (x, y and z). For example:

```
>>> mrc.voxel_size
rec.array((0.44825, 0.3925, 0.45874998),
  dtype=[('x', '<f4'), ('y', '<f4'), ('z', '<f4')])
>>> mrc.voxel_size.x
array(0.44825, dtype=float32)
```

Note that changing the voxel_size array in-place will *not* change the voxel size in the file – to prevent this being overlooked accidentally, the writeable flag is set to `False` on the voxel_size array.

To set the voxel size, assign a new value to the voxel_size attribute. You may give a single number, a 3-tuple `(x, y ,z)` or a modified version of the voxel_size array. The following examples are all equivalent:

```
>>> mrc.voxel_size = 1.0
```

```
>>> mrc.voxel_size = (1.0, 1.0, 1.0)
```

```
>>> vox_sizes = mrc.voxel_size
>>> vox_sizes.flags.writeable = True
>>> vox_sizes.x = 1.0
>>> vox_sizes.y = 1.0
>>> vox_sizes.z = 1.0
>>> mrc.voxel_size = vox_sizes
```

_set_voxel_size(*x_size*, *y_size*, *z_size*)

Set the voxel size.

Parameters

- **x_size** – The voxel size in the X direction, in angstroms
- **y_size** – The voxel size in the Y direction, in angstroms
- **z_size** – The voxel size in the Z direction, in angstroms

property nstart

Get or set the grid start locations.

This provides a convenient way to get and set the values of the header's `nxstart`, `nystart` and `nzstart` fields. Note that these fields are integers and are measured in voxels, not angstroms. The start locations are returned as a structured NumPy `record array` with three fields (x, y and z). For example:

```
>>> mrc.header.nxstart
array(0, dtype=int32)
>>> mrc.header.nystart
array(-21, dtype=int32)
>>> mrc.header.nzstart
array(-12, dtype=int32)
```

(continues on next page)

(continued from previous page)

```
>>> mrc.nstart
rec.array((0, -21, -12),
  dtype=[('x', '<i4'), ('y', '<i4'), ('z', '<i4')])
>>> mrc.nstart.y
array(-21, dtype=int32)
```

Note that changing the nstart array in-place will *not* change the values in the file – to prevent this being overlooked accidentally, the writeable flag is set to `False` on the nstart array.

To set the start locations, assign a new value to the nstart attribute. You may give a single number, a 3-tuple `(x, y ,z)` or a modified version of the nstart array. The following examples are all equivalent:

```
>>> mrc.nstart = -150
```

```
>>> mrc.nstart = (-150, -150, -150)
```

```
>>> starts = mrc.nstart
>>> starts.flags.writeable = True
>>> starts.x = -150
>>> starts.y = -150
>>> starts.z = -150
>>> mrc.nstart = starts
```

`_set_nstart`(*nxstart*, *nystart*, *nzstart*)

Set the grid start locations.

Parameters

- **`nxstart`** – The location of the first column in the unit cell
- **`nystart`** – The location of the first row in the unit cell
- **`nzstart`** – The location of the first section in the unit cell

`is_single_image`()

Identify whether the file represents a single image.

Returns
`True` if the data array is two-dimensional.

`is_image_stack`()

Identify whether the file represents a stack of images.

Returns
`True` if the data array is three-dimensional and the space group is zero.

`is_volume`()

Identify whether the file represents a volume.

Returns
`True` if the data array is three-dimensional and the space group is not zero.

`is_volume_stack`()

Identify whether the file represents a stack of volumes.

Returns
`True` if the data array is four-dimensional.

`set_image_stack()`

Change three-dimensional data to represent an image stack.

This method changes the space group number (`header.ispg`) to zero.

Raises
`ValueError` – If the data array is not three-dimensional.

`set_volume()`

Change three-dimensional data to represent a volume.

If the space group was previously zero (representing an image stack), this method sets it to one. Otherwise the space group is not changed.

Raises
`ValueError` – If the data array is not three-dimensional.

`update_header_from_data()`

Update the header from the data array.

This function updates the header byte order and machine stamp to match the byte order of the data. It also updates the file mode, space group and the dimension fields `nx`, `ny`, `nz`, `mx`, `my` and `mz`.

If the data is 2D, the space group is set to 0 (image stack). For 3D data the space group is not changed, and for 4D data the space group is set to 401 (simple P1 volume stack) unless it is already in the volume stack range (401–630).

This means that new 3D data will be treated as an image stack if the previous data was a single image or image stack, or as a volume if the previous data was a volume or volume stack.

Note that this function does *not* update the data statistics fields in the header (`dmin`, `dmax`, `dmean` and `rms`). Use the *`update_header_stats()`* function to update the statistics. (This is for performance reasons – updating the statistics can take a long time for large data sets, but updating the other header information is always fast because only the type and shape of the data array need to be inspected.)

`update_header_stats()`

Update the header's `dmin`, `dmax`, `dmean` and `rms` fields from the data.

Note that this can take some time with large files, particularly with files larger than the currently available memory.

Warns
RuntimeWarning – If the data array contains Inf or NaN values.

`reset_header_stats()`

Set the header statistics to indicate that the values are unknown.

`print_header`(*print_file=None*)

Print the contents of all header fields.

Parameters
`print_file` – The output text stream to use for printing the header. This is passed directly to the `file` argument of Python's `print()` function. The default is `None`, which means output will be printed to `sys.stdout`.

`get_labels()`

Get the labels from the MRC header.

Up to ten labels are stored in the header as arrays of 80 bytes. This method returns the labels as Python strings, filtered to remove non-printable characters. To access the raw bytes (including any non-printable characters) use the `header.label` attribute (and note that `header.nlabl` stores the number of labels currently set).

Returns
The labels, as a list of strings. The list will contain between 0 and 10 items, each containing up to 80 characters.

`add_label`(*label*)

Add a label to the MRC header.

The new label will be stored after any labels already in the header. If all ten labels are already in use, an exception will be raised.

Future versions of this method might add checks to ensure that labels containing valid text are not overwritten even if the `nlabl` value is incorrect.

Parameters
`label` – The label value to store, as a string containing only printable ASCII characters.

Raises

- `ValueError` – If the label is longer than 80 bytes or contains non-printable or non-ASCII characters.
- `IndexError` – If the file already contains 10 labels and so an additional label cannot be stored.

`validate`(*print_file=None*)

Validate this MrcObject.

This method runs a series of tests to check whether this object complies strictly with the MRC2014 format specification:

1. MRC format ID string: The header's `map` field must contain "MAP ".
2. Machine stamp: The machine stamp should contain one of `0x44 0x44 0x00 0x00`, `0x44 0x41 0x00 0x00` or `0x11 0x11 0x00 0x00`.
3. MRC mode: the `mode` field should be one of the supported mode numbers: 0, 1, 2, 4, 6 or 12. (Note that MRC modes 3 and 101 are also valid according to the MRC 2014 specification but are not supported by mrcfile.)
4. Map and cell dimensions: The header fields `nx`, `ny`, `nz`, `mx`, `my`, `mz`, `cella.x`, `cella.y` and `cella.z` must all be positive numbers.
5. Axis mapping: Header fields `mapc`, `mapr` and `maps` must contain the values 1, 2, and 3 (in any order).
6. Volume stack dimensions: If the spacegroup is in the range 401–630, representing a volume stack, the `nz` field should be exactly divisible by `mz` to represent the number of volumes in the stack.
7. Header labels: The `nlabl` field should be set to indicate the number of labels in use, and the labels in use should appear first in the label array (that is, there should be no blank labels between text-filled ones).
8. MRC format version: The `nversion` field should be 20140 or 20141 for compliance with the MRC2014 standard.
9. Extended header type: If an extended header is present, the `exttyp` field should be set to indicate the type of extended header.
10. Data statistics: The statistics in the header should be correct for the actual data, or marked as undetermined.

Parameters
`print_file` – The output text stream to use for printing messages about the validation. This

is passed directly to the `file` argument of Python's `print()` function. The default is `None`, which means output will be printed to `sys.stdout`.

Returns
`True` if this MrcObject is valid, or `False` if it does not meet the MRC format specification in any way.

3.14 mrcfile.utils module

3.14.1 utils

Utility functions used by the other modules in the mrcfile package.

3.14.2 Functions

- *`data_dtype_from_header()`*: Work out the data `dtype` from an MRC header.
- *`data_shape_from_header()`*: Work out the data array shape from an MRC header
- *`mode_from_dtype()`*: Convert a `numpy dtype` to an MRC mode number.
- *`dtype_from_mode()`*: Convert an MRC mode number to a `numpy dtype`.
- *`pretty_machine_stamp()`*: Get a nicely-formatted string from a machine stamp.
- *`machine_stamp_from_byte_order()`*: Get a machine stamp from a byte order indicator.
- *`byte_orders_equal()`*: Compare two byte order indicators for equal endianness.
- *`normalise_byte_order()`*: Convert a byte order indicator to < or >.
- *`spacegroup_is_volume_stack()`*: Identify if a space group number represents a volume stack.

`mrcfile.utils.`**`data_dtype_from_header`**`(`*header*`)`

Return the data dtype indicated by the given header.

This function calls *`dtype_from_mode()`* to get the basic dtype, and then makes sure that the byte order of the new dtype matches the byte order of the header's `mode` field.

Parameters
`header` – An MRC header as a `numpy record array`.

Returns
The `numpy dtype` object for the data array corresponding to the given header.

Raises
`ValueError` – If there is no corresponding dtype for the given mode.

`mrcfile.utils.`**`data_shape_from_header`**`(`*header*`)`

Return the data shape indicated by the given header.

Parameters
`header` – An MRC header as a `numpy record array`.

Returns
The shape tuple for the data array corresponding to the given header.

`mrcfile.utils.`**`mode_from_dtype`**`(`*dtype*`)`

Return the MRC mode number corresponding to the given `numpy dtype`.

The conversion is as follows:

- float16 -> mode 12
- float32 -> mode 2
- int8 -> mode 0
- int16 -> mode 1
- uint8 -> mode 6 (data will be widened to 16 bits in the file)
- uint16 -> mode 6
- complex64 -> mode 4

Note that there is no numpy dtype which corresponds to MRC mode 3.

Parameters
`dtype` – A `numpy dtype` object.

Returns
The MRC mode number.

Raises
`ValueError` – If there is no corresponding MRC mode for the given dtype.

`mrcfile.utils.`**`dtype_from_mode`**`(`*mode*`)`

Return the `numpy dtype` corresponding to the given MRC mode number.

The mode parameter may be given as a Python scalar, numpy scalar or single-item numpy array.

The conversion is as follows:

- mode 0 -> int8
- mode 1 -> int16
- mode 2 -> float32
- mode 4 -> complex64
- mode 6 -> uint16
- mode 12 -> float16

Note that modes 3 and 101 are not supported as there is no matching numpy dtype.

Parameters
`mode` – The MRC mode number. This may be given as any type which can be converted to an int, for example a Python scalar (`int` or `float`), a numpy scalar or a single-item numpy array.

Returns
The `numpy dtype` object corresponding to the given mode.

Raises
`ValueError` – If there is no corresponding dtype for the given mode, or if `mode` is an array and does not contain exactly one item.

`mrcfile.utils.`**`pretty_machine_stamp`**`(`*machst*`)`

Return a human-readable hex string for a machine stamp.

`mrcfile.utils.`**`byte_order_from_machine_stamp`**`(`*machst*`)`

Return the byte order corresponding to the given machine stamp.

Parameters
`machst` – The machine stamp, as a `bytearray` or a `numpy array` of bytes.

Returns
< if the machine stamp represents little-endian data, or > if it represents big-endian.

Raises
`ValueError` – If the machine stamp is invalid.

`mrcfile.utils.`**`machine_stamp_from_byte_order`**`(`*byte_order='='*`)`

Return the machine stamp corresponding to the given byte order indicator.

Parameters
`byte_order` – The byte order indicator: one of =, < or >, as defined and used by numpy dtype objects.

Returns
The machine stamp which corresponds to the given byte order, as a `bytearray`. This will be either `(0x44, 0x44, 0, 0)` for little-endian or `(0x11, 0x11, 0, 0)` for big-endian. If the given byte order indicator is =, the native byte order is used.

Raises
`ValueError` – If the byte order indicator is unrecognised.

`mrcfile.utils.`**`byte_orders_equal`**`(`*a*, *b*`)`

Work out if the byte order indicators represent the same endianness.

Parameters

- `a` – The first byte order indicator: one of =, < or >, as defined and used by `numpy dtype` objects.
- `b` – The second byte order indicator.

Returns
`True` if the byte order indicators represent the same endianness.

Raises
`ValueError` – If the byte order indicator is not recognised.

`mrcfile.utils.`**`normalise_byte_order`**`(`*byte_order*`)`

Convert a numpy byte order indicator to one of < or >.

Parameters
`byte_order` – One of =, < or >.

Returns
< if the byte order indicator represents little-endian data, or > if it represents big-endian. Therefore on a little-endian machine, = will be converted to <, but on a big-endian machine it will be converted to >.

Raises
`ValueError` – If `byte_order` is not one of =, < or >.

`mrcfile.utils.`**`spacegroup_is_volume_stack`**`(`*ispg*`)`

Identify if the given space group number represents a volume stack.

Parameters
`ispg` – The space group number, as an integer, numpy scalar or single- element numpy array.

Returns
`True` if the space group number is in the range 401–630.

`mrcfile.utils.`**`is_printable_ascii`**(*string_*)

Check if a string is entirely composed of printable ASCII characters.

`mrcfile.utils.`**`printable_string_from_bytes`**(*bytes_*)

Convert bytes into a printable ASCII string by removing non-printable characters.

`mrcfile.utils.`**`bytes_from_string`**(*string_*)

Convert a string to bytes.

Even though this is a one-liner, the details are tricky to get right so things work properly in both Python 2 and 3. It's broken out as a separate function so it can be thoroughly tested.

Raises
`UnicodeError` – If the input contains non-ASCII characters.

3.15 mrcfile.validator module

3.15.1 validator

Module for top-level functions that validate MRC files.

This module is runnable to allow files to be validated easily from the command line.

`mrcfile.validator.`**`main`**(*args=None*)

Validate a list of MRC files given as command arguments.

The return value is used as the process exit code when this function is called by running this module or from the corresponding `console_scripts` entry point.

Returns
`0` if all command arguments are names of valid MRC files. 1 if no file names are given or any of the files is not a valid MRC file.

`mrcfile.validator.`**`validate_all`**(*names*, *print_file=None*)

Validate a list of MRC files.

This function calls *`validate()`* for each file name in the given list.

Parameters

- **`names`** – A sequence of file names to open and validate.
- **`print_file`** – The output text stream to use for printing messages about the validation. This is passed directly to the `print_file` argument of the *`validate()`* function. The default is `None`, which means output will be printed to `sys.stdout`.

Returns
`True` if all of the files are valid, or `False` if any of the files do not meet the MRC format specification in any way.

Raises
`OSError` – If one of the files does not exist or cannot be opened.

Warns
RuntimeWarning – If one of the files is seriously invalid because it has no map ID string, an incorrect machine stamp, an unknown mode number, or is not the same size as expected from the header.

`mrcfile.validator.validate`(*name*, *print_file=None*)

Validate an MRC file.

This function first opens the file by calling `open()` (with `permissive=True`), then calls `validate()`, which runs a series of tests to check whether the file complies with the MRC2014 format specification.

If the file is completely valid, this function returns `True`, otherwise it returns `False`. Messages explaining the validation result will be printed to `sys.stdout` by default, but if a text stream is given (using the `print_file` argument) output will be printed to that instead.

Badly invalid files will also cause `warning` messages to be issued, which will be written to `sys.stderr` by default. See the documentation of the `warnings` module for information on how to suppress or capture warning output.

Because the file is opened by calling `open()`, gzip- and bzip2-compressed MRC files can be validated easily using this function.

After the file has been opened, it is checked for problems. The tests are:

1. MRC format ID string: The `map` field in the header should contain "MAP ".
2. Machine stamp: The machine stamp should contain one of `0x44 0x44 0x00 0x00`, `0x44 0x41 0x00 0x00` or `0x11 0x11 0x00 0x00`.
3. MRC mode: the `mode` field should be one of the supported mode numbers: 0, 1, 2, 4, 6 or 12. (Note that MRC modes 3 and 101 are also valid according to the MRC 2014 specification but are not supported by mrcfile.)
4. Map and cell dimensions: The header fields `nx`, `ny`, `nz`, `mx`, `my`, `mz`, `cella.x`, `cella.y` and `cella.z` must all be positive numbers.
5. Axis mapping: Header fields `mapc`, `mapr` and `maps` must contain the values 1, 2, and 3 (in any order).
6. Volume stack dimensions: If the spacegroup is in the range 401–630, representing a volume stack, the `nz` field should be exactly divisible by `mz` to represent the number of volumes in the stack.
7. Header labels: The `nlabl` field should be set to indicate the number of labels in use, and the labels in use should appear first in the label array.
8. MRC format version: The `nversion` field should be 20140 or 20141 for compliance with the MRC2014 standard.
9. Extended header type: If an extended header is present, the `exttyp` field should be set to indicate the type of extended header.
10. Data statistics: The statistics in the header should be correct for the actual data in the file, or marked as undetermined.
11. File size: The size of the file on disk should match the expected size calculated from the MRC header.

Parameters

- **`name`** – The file name to open and validate.
- **`print_file`** – The output text stream to use for printing messages about the validation. This is passed directly to the `file` argument of Python's `print()` function. The default is `None`, which means output will be printed to `sys.stdout`.

Returns

`True` if the file is valid, or `False` if the file does not meet the MRC format specification in any way.

Raises

`OSError` – If the file does not exist or cannot be opened.

Warns
: **RuntimeWarning** – If the file is seriously invalid because it has no map ID string, an incorrect machine stamp, an unknown mode number, or is not the same size as expected from the header.

PYTHON MODULE INDEX

INDEX