PyMOC Documentation

Release 0.4.2

Graham Bell

May 25, 2017

Contents

PyMOC is a module for manipulating Multi-Order Coverage (MOC) maps. It includes support for reading and writing the three encodings mentioned in the IVOA recommendation: FITS, JSON and ASCII.

PyMOC also includes a utility program `pymoctool` to allow MOC files to be manipulated from the command line.

CHAPTER 1

Contents

Installation

The module can be installed using the `setup.py` script:

```
python setup.py install
```

Unit Tests

Prior to installation, the unit tests can be run using:

```
PYTHONPATH=lib python3 -m unittest
```

or:

```
PYTHONPATH=lib python2 -m unittest discover
```

The *test-extra* directory contains additional tests which may take longer to perform. You can exclude these by specifying just the plain *test* directory, for example with:

```
PYTHONPATH=lib python -m unittest discover -s test
```

The routines included in the doctests should also be covered by the unit tests. However to ensure the documentation is correct, they can be checked with:

```
sphinx-build -b doctest doc doc/_build/doctest
```

Requirements

For reading and writing data in FITS format, the `astropy` library is required.

`Healpy` is needed for some of the utility functions such as `plot_moc` and `catalog_to_moc`.

The MOC Class

pymoc

The `pymoc` module imports the *`MOC`* class from the *`pymoc.moc`* module, allowing it to be imported as follows:

```
from pymoc import MOC
```

pymoc.moc

class `pymoc.moc.`**MOC**(*order=None*, *cells=None*, *filename=None*, *filetype=None*, *name=None*, *mocid=None*, *origin=None*, *moctype=None*)

Class representing Multi-Order Coverage maps.

Apart from the properties listed below, the MOC also includes the following attributes:

- id
- name
- origin

__init__(*order=None*, *cells=None*, *filename=None*, *filetype=None*, *name=None*, *mocid=None*, *origin=None*, *moctype=None*)

Construct new MOC object.

By default the new MOC will be empty, but if an order and a collection of cells are specified, then these will be added to the new MOC. If a filename is specified then data from the given file will be read into the new object, and if it is a FITS file, the metadata will also be read, although this can be overridden by values given explicitly as constructor arguments.

Additional metadata can be added to the MOC using the `name`, `mocid`, `origin` and `moctype` arguments, or added at a later time using the corresponding attributes and properties. The metadata values can be read from and written to FITS files, but are not included when a MOC is written to the other formats (JSON or ASCII).

```
>>> from pymoc import MOC
>>> m = MOC()
>>> m.cells
0
```

```
>>> m = MOC(10, (1234, 4321))
>>> m.cells
2
```

```
>>> m = MOC(name='example', moctype='IMAGE')
>>> m.name
'example'
>>> m.type
'IMAGE'
```

__iter__()

Iterator for MOC objects.

This yields an (order, cell collection) pair for each order at which there are cells. The results will be returned in ascending order of the order number.

```
>>> m = MOC(0, (1, 2))
>>> m.add(1, (0,))
>>> for (order, cells) in m:
...     print(str(order) + ' ' + str(sorted(cells)))
0 [1, 2]
1 [0]
```

__len__()
Length operator for MOC objects.

Returns the number of orders at which the MOC has cells.

```
>>> m = MOC(0, (1, 2))
>>> len(m)
1
```

__getitem__(*order*)
Subscripting operator for MOC objects.

This retrieves a collection of cells at the given order of the MOC.

```
>>> m = MOC(5, (6, 7))
>>> sorted(m[5])
[6, 7]
```

__eq__(*other*)
Equality test operator.

```
>>> MOC(1, (4, 5, 6, 7)) == MOC(0, (1,))
True
>>> MOC(2, (0, 1)) == MOC(0, (0,))
False
>>> MOC(1, (5, 6)) != MOC(1, (1, 2))
True
>>> MOC(2, (8, 9, 10, 11)) != MOC(1, (2,))
False
```

__iadd__(*other*)
In-place addition operator.

Updates the MOC to represent the union of itself and the other MOC.

```
>>> p = MOC(4, (5, 6))
>>> p += MOC(4, (7, 8))
>>> repr(p)
'<MOC: [(4, [5, 6, 7, 8])]>'
```

__add__(*other*)
Addition operator.

Returns a MOC which is the union of two MOCs.

```
>>> MOC(4, (5, 6)) + MOC(4, (7, 8))
<MOC: [(4, [5, 6, 7, 8])]>
```

__sub__(*other*)
Subtraction operator.

Returns a MOC which is the copy of the first MOC with the intersection with the second MOC removed.

```
>>> MOC(0, (0,)) - MOC(2, (15,))
<MOC: [(1, [0, 1, 2]), (2, [12, 13, 14])]>
```

__isub__(*other*)
In-place subtraction operator.

Removes areas which overlap with the given MOC.

__repr__()
Generate printable representation.

Since the constructor only accepts cells at one order and we may be generating a representation for a MOC with cells at multiple orders we can't try to give an expression which would construct the object. Instead show a description in angle brackets.

order
The highest order at which the MOC has cells.

```
>>> m = MOC(4, (3, 2, 1))
>>> m.order
4
```

type
The type of MOC (IMAGE or CATALOG).

```
>>> m = MOC(moctype='IMAGE')
>>> m.type
'IMAGE'
```

```
>>> m.type = 'CATALOG'
>>> m.type
'CATALOG'
```

normalized
Whether the MOC has been normalized or not.

```
>>> m = MOC()
>>> m.add(1, (0,))
>>> m.add(2, (1,))
>>> m.normalized
False
```

```
>>> m.normalize()
>>> m.normalized
True
```

area
The area enclosed by the MOC, in steradians.

```
>>> m = MOC(0, (0, 1, 2))
>>> round(m.area, 2)
3.14
```

area_sq_deg
The area enclosed by the MOC, in square degrees.

```
>>> from math import sqrt
>>> m = MOC(0, (0,))
>>> round(sqrt(m.area_sq_deg), 2)
58.63
```

cells
The number of cells in the MOC.

This gives the total number of cells at all orders, with cells from every order counted equally.

```
>>> m = MOC(0, (1, 2))
>>> m.cells
2
```

add(*order*, *cells*, *no_validation=False*)
Add cells at a given order to the MOC.

The cells are inserted into the MOC at the specified order. This leaves the MOC in an un-normalized state. The cells are given as a collection of integers (or types which can be converted to integers).

```
>>> m = MOC()
>>> m.add(4, (20, 21))
>>> m.cells
2
```

```
>>> m.add(5, (88, 89))
>>> m.cells
4
```

The *no_validation* option can be given to skip validation of the cell numbers. They must already be integers in the correct range.

remove(*order*, *cells*)
Remove cells at a given order from the MOC.

clear()
Clears all cells from a MOC.

```
>>> m = MOC(4, (5, 6))
>>> m.clear()
>>> m.cells
0
```

copy()
Return a copy of a MOC.

```
>>> p = MOC(4, (5, 6))
>>> q = p.copy()
>>> repr(q)
'<MOC: [(4, [5, 6])]>'
```

contains(*order*, *cell*, *include_smaller=False*)
Test whether the MOC contains the given cell.

If the include_smaller argument is true then the MOC is considered to include a cell if it includes part of that cell (at a higher order).

```
>>> m = MOC(1, (5,))
>>> m.contains(0, 0)
False
>>> m.contains(0, 1, True)
True
>>> m.contains(0, 1, False)
False
>>> m.contains(1, 4)
False
>>> m.contains(1, 5)
True
>>> m.contains(2, 19)
False
>>> m.contains(2, 21)
True
```

intersection(*other*)
Returns a MOC representing the intersection with another MOC.

```
>>> p = MOC(2, (3, 4, 5))
>>> q = MOC(2, (4, 5, 6))
>>> p.intersection(q)
<MOC: [(2, [4, 5])]>
```

normalize(*max_order=29*)
Ensure that the MOC is "well-formed".

This structures the MOC as is required for the FITS and JSON representation. This method is invoked automatically when writing to these formats.

The number of cells in the MOC will be minimized, so that no area of the sky is covered multiple times by cells at different orders, and if all four neighboring cells are present at an order (other than order 0), they are merged into their parent cell at the next lower order.

```
>>> m = MOC(1, (0, 1, 2, 3))
>>> m.cells
4
```

```
>>> m.normalize()
>>> m.cells
1
```

flattened(*order=None*, *include_smaller=True*)
Return a flattened pixel collection at a single order.

read(*filename*, *filetype=None*, *include_meta=False*, ***kwargs*)
Read data from the given file into the MOC object.

The cell lists read from the file are added to the current object. Therefore if the object already contains some cells, it will be updated to represent the union of the current coverge and that from the file.

The file type can be specified as "fits", "json" or "ascii", with "text" allowed as an alias for "ascii". If the type is not specified, then an attempt will be made to guess from the file name, or the contents of the file.

Note that writing to FITS and JSON will cause the MOC to be normalized automatically.

Any additional keyword arguments (kwargs) are passed on to the corresponding pymoc.io read functions (read_moc_fits, read_moc_json or read_moc_ascii).

write (*filename*, *filetype=None*, ***kwargs*)
Write the coverage data in the MOC object to a file.

The filetype can be given or left to be inferred as for the read method.

Any additional keyword arguments (kwargs) are passed on to the corresponding pymoc.io write functions (write_moc_fits, write_moc_json or write_moc_ascii). This can be used, for example, to set clobber=True when writing FITS files.

I/O Functions

Input and output (I/O) functions for the three encodings used by MOC are located in a separate part of the package. This is to allow you to use the *MOC* class itself without needing the requirements of the I/O routines.

For general file handling, the giving the `filename` argument to the *MOC* constructor, or using the *read()* and *write()* methods may be sufficient. The dedicated I/O functions described here can be used in more specialized situations, such as interacting with FITS HDU objects or reading and writing already-open file objects.

pymoc.io.ascii

pymoc.io.ascii.**write_moc_ascii** (*moc*, *filename=None*, *file=None*)
Write a MOC to an ASCII file.

Either a filename, or an open file object can be specified.

pymoc.io.ascii.**read_moc_ascii** (*moc*, *filename=None*, *file=None*)
Read from an ASCII file into a MOC.

Either a filename, or an open file object can be specified.

pymoc.io.fits

pymoc.io.fits.**write_moc_fits_hdu** (*moc*)
Create a FITS table HDU representation of a MOC.

pymoc.io.fits.**write_moc_fits** (*moc*, *filename*, ***kwargs*)
Write a MOC as a FITS file.

Any additional keyword arguments are passed to the astropy.io.fits.HDUList.writeto method.

pymoc.io.fits.**read_moc_fits** (*moc*, *filename*, *include_meta=False*, ***kwargs*)
Read data from a FITS file into a MOC.

Any additional keyword arguments are passed to the astropy.io.fits.open method.

pymoc.io.fits.**read_moc_fits_hdu** (*moc*, *hdu*, *include_meta=False*)
Read data from a FITS table HDU into a MOC.

pymoc.io.json

pymoc.io.json.**write_moc_json** (*moc*, *filename=None*, *file=None*)
Write a MOC in JSON encoding.

Either a filename, or an open file object can be specified.

pymoc.io.json.**read_moc_json**(*moc*, *filename=None*, *file=None*)
Read JSON encoded data into a MOC.

Either a filename, or an open file object can be specified.

Utilities

pymoc.util.catalog

pymoc.util.catalog.**catalog_to_moc**(*catalog*, *radius*, *order*, ***kwargs*)
Convert a catalog to a MOC.

The catalog is given as an Astropy SkyCoord object containing multiple coordinates. The radius of catalog entries can be given as an Astropy Quantity (with units), otherwise it is assumed to be in arcseconds.

Any additional keyword arguments are passed on to *catalog_to_cells*.

pymoc.util.catalog.**catalog_to_cells**(*catalog*, *radius*, *order*, *include_fallback=True*, ***kwargs*)
Convert a catalog to a set of cells.

This function is intended to be used via *catalog_to_moc* but is available for separate usage. It takes the same arguments as that function.

This function uses the Healpy *query_disc* function to get a list of cells for item in the catalog in turn. Additional keyword arguments, if specified, are passed to *query_disc*. This can include, for example, *inclusive* (set to *True* to include cells overlapping the radius as well as those with centers within it) and *fact* (to control sampling when *inclusive* is specified).

If cells at the given order are bigger than the given radius, then *query_disc* may find no none inside the radius. In this case, if *include_fallback* is *True* (the default), the cell at each position is included.

pymoc.util.catalog.**read_ascii_catalog**(*filename*, *format_*, *unit=None*)
Read an ASCII catalog file using Astropy.

This routine is used by pymoctool to load coordinates from a catalog file in order to generate a MOC representation.

pymoc.util.plot

pymoc.util.plot.**plot_moc**(*moc*, *order=None*, *antialias=0*, *filename=None*, *projection='cart'*, *color='blue'*, *title=''*, *coord_sys='C'*, *graticule=True*, ***kwargs*)
Plot a MOC using Healpy.

This generates a plot of the MOC at the specified order, or the MOC's current order if this is not specified. The MOC is flattened at an order of *order* + *antialias* to generate intermediate color levels.

Parameters

- **order** – HEALPix order at which to generate the plot.
- **antialias** – number of additional HEALPix orders to use for intermediate color levels. (There can be *4 ** antialias* levels.)
- **filename** – file in which to save plot. If not specified then the plot is shown with *plt.show()*.
- **projection** – map projection to be used — can be shortened to 4 characters. One of:

  - *'cart[esian]'* (uses *healpy.visufunc.cartview*)
  - *'moll[weide]'* (uses *healpy.visufunc.mollview*)
  - *'gnom[onic]'* (uses *healpy.visufunc.gnomview*)
- **color** – color scheme. One of:
  - *'blue'*
  - *'green'*
  - *'red'*
  - *'black'*
- **title** – title of the plot.
- **coord_sys** – Healpy coordinate system code for the desired plot coordinates. One of:
  - *'C'* — Celestial (equatorial)
  - *'G'* — Galactic
  - *'E'* — Ecliptic
- **graticule** – whether or not to draw a graticule.
- ****kwargs** – passed to the selected Healpy plotting function.

pymoc.util.tool

pymoctool - PyMOC utility program

Usage:

```
pymoctool [INPUT]... [COMMAND [FILE]]... [--output OUTPUT]
```

This program can be used to manipulate MOC files. All formats handled by PyMOC (FITS, JSON and ASCII) are supported. The program maintains a "running" MOC into which all input files are merged. Each command operates on the "running" MOC. Input files and commands are processed in the order given to the command.

For example, this command:

```
pymoctool a.fits --output a.json b.fits --output merged.txt
```

would load a MOC "a.fits", re-write it as "a.json", merge (forming the union with) "b.fits" and write the combined MOC to "merged.txt".

class pymoc.util.tool.**CommandDict**
Decorator to record commands in a dictionary.

exception pymoc.util.tool.**CommandError**
Class representing expected errors from PyMOC tool commands.

class pymoc.util.tool.**MOCTool**
Class implementing a basic tool to manipulate MOC files.

run(*params*)
Main run method for PyMOC tool.

Takes a list of command line arguments to process.

Each operation is performed on a current "running" MOC object.

read_moc (*filename*)
Read a file into the current running MOC object.

If the running MOC object has not yet been created, then it is created by reading the file, which will import the MOC metadata. Otherwise the metadata are not imported.

catalog()
Create MOC from catalog of coordinates.

This command requires that the Healpy and Astropy libraries be available. It attempts to load the given catalog, and merges it with the running MOC.

The name of an ASCII catalog file should be given. The file should contain either "RA" and "Dec" columns (for ICRS coordinates) or "Lon" and "Lat" columns (for galactic coordinates). The MOC order and radius (in arcseconds) can be given with additional options.

```
pymoctool --catalog coords.txt
    [order 12]
    [radius 3600]
    [unit (hour | deg | rad) (deg | rad)]
    [format commented_header]
    [inclusive]
```

Units (if not specified) are assumed to be hours and degrees for ICRS coordinates and degrees for galactic coordinates. The format, if not specified (as an Astropy ASCII table format name) is assumed to be commented header, e.g.:

```
# RA Dec
01:30:00 +45:00:00
22:30:00 +45:00:00
```

help()
Display command usage information.

identifier()
Set the identifier of the current MOC.

The new identifier should be given after this option.

```
pymoctool ... --id 'New MOC identifier' --output new_moc.fits
```

display_info()
Display basic information about the running MOC.

intersection()
Compute the intersection with the given MOC.

This command takes the name of a MOC file and forms the intersection of the running MOC with that file.

```
pymoctool a.fits --intersection b.fits --output intersection.fits
```

name()
Set the name of the current MOC.

The new name should be given after this option.

```
pymoctool ... --name 'New MOC name' --output new_moc.fits
```

normalize()
Normalize the MOC to a given order.

This command takes a MOC order (0-29) and normalizes the MOC so that its maximum order is the given order.

```
pymoctool a.fits --normalize 10 --output a_10.fits
```

write_moc()
Write the MOC to a given file.

subtract()
Subtract the given MOC from the running MOC.

This command takes the name of a MOC file to be subtracted from the running MOC.

```
pymoctool a.fits --subtract b.fits --output difference.fits
```

plot()
Show the running MOC on an all-sky map.

This command requires that the Healpy and matplotlib libraries be available. It plots the running MOC, which should be normalized to a lower order first if it would generate an excessively large pixel array.

```
pymoctool a.moc --normalize 8 --plot
```

It also accepts additional arguments which can be used to control the plot. The 'order' option can be used instead of normalizing the MOC before plotting. The 'antialias' option specifies an additional number of MOC orders which should be used to smooth the edges as plotted – 1 or 2 is normally sufficient. The 'file' option can be given to specify a file to which the plot should be saved.

```
pymoctool ... --plot [order <order>] [antialias <level>] [file <filename>] ...
```

version()
Show PyMOC version number.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

C

D

F

H

I

M

N

O

P

R

S

T

V

W