mpop documentation

Release v1.5.1

SMHI

May 21, 2018

Contents

The Meteorological Post-Processing package is a python library for generating RGB products for meteorological remote sensing. As such it can create RGB composites directly from satellite instrument channels, or take advantage of precomputed PGEs.

Get to the project page, with source and downloads.

It is designed to be easily extendable to support any meteorological satellite by the creation of plugins. In the base distribution, we provide support for Meteosat-7, -8, -9, -10, Himawari-6 (MTSAT-1R), Himawari-7 (MTSAT-2), GOES-11, GOES-12, GOES-13 through the use of mipp, and NOAA-15, -16, -17, -18, -19, Metop-A and -B through the use of AAPP.

Reprojection of data is also available through the use of pyresample.

CHAPTER 1

Installation instructions

1.1 Getting the files and installing them

First you need to get the files from github:

```
cd /path/to/my/source/directory/
git clone git://github.com/mraspaud/mpop.git
```

You can also retreive a tarball from there if you prefer, then run:

```
tar zxvf tarball.tar.gz
```

Then you need to install mpop on you computer:

```
cd mpop
python setup.py install [--prefix=/my/custom/installation/directory]
```

You can also install it in develop mode to make it easier to hack:

```
python setup.py develop [--prefix=/my/custom/installation/directory]
```

1.2 Configuration

1.2.1 Environment variables

Environment variables which are needed for mpop are the *PYTHONPATH* of course, and the *PPP_CONFIG_DIR*, which is the directory where the configuration files are to be found. If the latter is not defined, the *etc* directory of the mpop installation is used.

1.2.2 Input data directories

The input data directories are setup in the satellite configuration files, which can be found in the *PPP_CONFIG_DIR* directory (some template files are provided with mpop in the *etc* directory):

```
[seviri-level1]
format = 'xrit/MSG'
dir='/data/geo_in'
filename='H-000-MSG?__-MSG?_________-%(channel)s-%(segment)s-%Y%m%d%H%M-__'
filename_pro='H-000-MSG?__-MSG?_________-__________-%(segment)s-%Y%m%d%H%M-__'
filename_epi='H-000-MSG?__-MSG?_________-__________-%(segment)s-%Y%m%d%H%M-__'


[seviri-level2]
format='mipp_xrit'
```

The different levels indicate different steps of the reading. The *level2* section gives at least the plugin to read the data with. In some cases, the data is first read from another level, as is this case with HRIT/LRIT data when we use mipp: there we use the *level1* section.

The data location is generally dealt in to parts: the directory and the filename. There can also be additional filenames depending on the reader plugin: here, mipp needs also the filename for prologue and epilogue files.

Note that the section starts with the name of the instrument. This is important in the case where several instruments are available for the same satellite. Note also that the filename can contain wildcards (* and *?*) and optional values (here channel, segment, and time markers). It is up to the input plugin to handle these constructs if needed.

CHAPTER 2

Quickstart

The software uses OOP extensively, to allow higher level metaobject handling.

For this tutorial, we will use the Meteosat plugin and data.

Don't forget to first source the *profile* file of interest located in the source *etc* directory.

2.1 First example

Changed in version 0.10.0: The factory-based loading was added in 0.10.0

Ok, let's get it on:

```
>>> from mpop.satellites import GeostationaryFactory
>>> from mpop.projector import get_area_def
>>> import datetime
>>> time_slot = datetime.datetime(2009, 10, 8, 14, 30)
>>> global_data = GeostationaryFactory.create_scene("Meteosat-9", "", "seviri", time_
↪slot)
>>> europe = get_area_def("EuropeCanary")
>>> global_data.load([0.6, 0.8, 10.8], area_extent=europe.area_extent)
>>> print global_data
'IR_097: (9.380,9.660,9.940)µm, resolution 3000.40316582m, not loaded'
'IR_016: (1.500,1.640,1.780)µm, resolution 3000.40316582m, not loaded'
'VIS008: (0.740,0.810,0.880)µm, shape (1200, 3000), resolution 3000.40316582m'
'VIS006: (0.560,0.635,0.710)µm, shape (1200, 3000), resolution 3000.40316582m'
'WV_062: (5.350,6.250,7.150)µm, resolution 3000.40316582m, not loaded'
'IR_120: (11.000,12.000,13.000)µm, resolution 3000.40316582m, not loaded'
'WV_073: (6.850,7.350,7.850)µm, resolution 3000.40316582m, not loaded'
'IR_087: (8.300,8.700,9.100)µm, resolution 3000.40316582m, not loaded'
'IR_039: (3.480,3.920,4.360)µm, resolution 3000.40316582m, not loaded'
'HRV: (0.500,0.700,0.900)µm, resolution 1000.13434887m, not loaded'
'IR_134: (12.400,13.400,14.400)µm, resolution 3000.40316582m, not loaded'
'IR_108: (9.800,10.800,11.800)µm, shape (1200, 3000), resolution 3000.40316582m'
```

In this example, we create a scene object for the seviri instrument onboard Meteosat-9, specifying the time of the snapshot of interest. The time is defined as a datetime object.

The next step is loading the data. This is done using mipp, which takes care of reading the HRIT data, and slicing the data so that we read just what is needed. Calibration is also done with mipp. In order to slice the data, we retreive the area we will work on, here set to variable *europe*.

Here we call the `load()` function with a list of the wavelengths of the channels we are interested in, and the area extent in satellite projection of the area of interest. Each retrieved channel is the closest in terms of central wavelength, provided that the required wavelength is within the bounds of the channel.

The wavelengths are given in micrometers and have to be given as a floating point number (*i.e.*, don't type '1', but '1.0'). Using an integer number instead returns a channel based on resolution, while using a string retrieves a channels based on its name.

```
>>> img = global_data.image.overview()
>>> img.save("./myoverview.png")
>>>
```

Once the channels are loaded, we generate an overview RGB composite image, and save it as a png image. Instead of `save()`, one could also use `show()` if the only purpose is to display the image on screen.

Available composites are listed in the `mpop.satellites.visir` module in the mpop documentation.

2.2 We want more!

In the last example, the composite generation worked because the channels needed for the overview (0.6, 0.8, 10.8 μm) were loaded. If we try to generate a day natural color composite, which requires also the 1.6 μm channel, it will result in an error:

```
>>> img = global_data.image.natural()
Traceback (most recent call last):
  ...
NotLoadedError: Required channel 1.63 not loaded, aborting.
```

So it means that we have to load the missing channel first. To do this we could enter the channels list to load manually, as we did for the overview, but we provide a way to get the list of channels needed by a given method using the *prerequisites* method attribute:

```
>>> global_data.load(global_data.image.natural.prerequisites, area_extent=europe.area_
↪extent)
>>> img = global_data.image.natural()
>>>
```

Now you can save the image:

```
>>> img.save("./mynaturalcolors.png")
>>>
```

If you want to combine several prerequisites for channel loading, since prerequisites are python sets, you can do:

```
>>> global_data.load(global_data.image.overview.prerequisites |
...                  global_data.image.natural.prerequisites,
...                  area_extent=europe.area_extent)
>>>
```

and add as many | *global_data.image.mymethod.prerequisites* as needed.

2.3 Retrieving channels

Retrieving channels is dead easy. From the center wavelength:

```
>>> print global_data[0.6]
'VIS006: (0.560,0.635,0.710)μm, shape (1200, 3000), resolution 3000.40316582m'
```

or from the channel name:

```
>>> print global_data["VIS006"]
'VIS006: (0.560,0.635,0.710)μm, shape (1200, 3000), resolution 3000.40316582m'
```

or from the resolution:

```
>>> print global_data[3000]
'VIS006: (0.560,0.635,0.710)μm, shape (1200, 3000), resolution 3000.40316582m'
```

or more than one at the time:

```
>>> print global_data[3000, 0.8]
'VIS008: (0.740,0.810,0.880)μm, shape (1200, 3000), resolution 3000.40316582m'
```

The printed lines consists of the following values:

- First the name is displayed,
- then the triplet gives the min-, center-, and max-wavelength of the channel,
- follows the shape of the loaded data, or *None* if the data is not loaded,
- and finally the theoretical resolution of the channel is shown.

The data of the channel can be retrieved as an numpy (masked) array using the data property:

```
>>> print global_data[0.6].data
[[-- -- -- ..., -- -- --]
 [-- -- -- ..., -- -- --]
 [-- -- -- ..., -- -- --]
 ...,
 [7.37684259374 8.65549530999 6.58997938374 ..., 0.29507370375 0.1967158025
  0.1967158025]
 [7.18012679124 7.86863209999 6.19654777874 ..., 0.29507370375
  0.29507370375 0.29507370375]
 [5.80311617374 7.57355839624 6.88505308749 ..., 0.29507370375
  0.29507370375 0.29507370375]]
```

2.4 Channel arithmetics

New in version 0.10.0: Channel arithmetics added.

The common arithmetical operators are supported on channels, so that one can run for example:

```
>>> cool_channel = (global_data[0.6] - global_data[0.8]) * global_data[10.8]
```

2.5 PGEs

From the satellite data PGEs[1] are generated by the accompanying program. The loading procedure for PGEs is exactly the same as with regular channels:

```
>>> global_data.area = "EuropeCanary"
>>> global_data.load(["CTTH"])
>>>
```

and they can be retrieved as simply as before:

```
>>> print global_data["CTTH"]
'CTTH: shape (1200, 3000), resolution 3000.40316582m'
```

2.6 Making custom composites

Building custom composites makes use of the `imageo` module. For example, building an overview composite can be done manually with:

```
>>> from mpop.imageo.geo_image import GeoImage
>>> img = GeoImage((global_data[0.6].data,
...                 global_data[0.8].data,
...                 -global_data[10.8].data),
...                 "EuropeCanary",
...                 time_slot,
...                 mode = "RGB")
>>> img.enhance(stretch="crude")
>>> img.enhance(gamma=1.7)
```

New in version 0.10.0: Custom composites module added.

In order to have mpop automatically use the composites you create, it is possible to write them in a python module which name has to be specified in the *mpop.cfg* configuration file, under the *composites* section:

```
[composites]
module=mpop.smhi_composites
```

The module has to be importable (i.e. it has to be in the pythonpath). Here is an example of such a module:

```
def overview(self):
    """Make an overview RGB image composite.
    """
    self.check_channels(0.635, 0.85, 10.8)

    ch1 = self[0.635].check_range()
    ch2 = self[0.85].check_range()
    ch3 = -self[10.8].data

    img = geo_image.GeoImage((ch1, ch2, ch3),
                             self.area,
                             self.time_slot,
                             fill_value=(0, 0, 0),
                             mode="RGB")
```

(continues on next page)

[1] PGEs in Meteosat : CloudType and CTTH

(continued from previous page)

```
    img.enhance(stretch = (0.005, 0.005))

    return img

overview.prerequisites = set([0.6, 0.8, 10.8])

def hr_visual(self):
    """Make a High Resolution visual BW image composite from Seviri
    channel.
    """
    self.check_channels("HRV")

    img = geo_image.GeoImage(self["HRV"].data,
                             self.area,
                             self.time_slot,
                             fill_value=0,
                             mode="L")
    img.enhance(stretch="crude")
    return img

hr_visual.prerequisites = set(["HRV"])

seviri = [overview,
          hr_visual]
```

2.7 Projections

Until now, we have used the channels directly as provided by the satellite, that is in satellite projection. Generating composites thus produces views in satellite projection, *i.e.* as viewed by the satellite.

Most often however, we will want to project the data onto a specific area so that only the area of interest is depicted in the RGB composites.

Here is how we do that:

```
>>> local_data = global_data.project("eurol")
>>>
```

Now we have projected data onto the "eurol" area in the *local_data* variable and we can operate as before to generate and play with RGB composites:

```
>>> img = local_data.image.overview()
>>> img.save("./local_overview.tif")
>>>
```

The image is saved here in GeoTiff format.

On projected images, one can also add contour overlay with the `imageo.geo_image.add_overlay()`.

CHAPTER 3

Making use of the mpop package

The mpop package is the heart of mpop: here are defined the core classes which the user will then need to build satellite composites.

3.1 Conventions about satellite names

Throughout the document, we will use the following conventions:

- *platform name* is the name of an individual satellite following the OSCAR naming scheme, e.g. "NOAA-19".
- ***variant* will be used to differentiate the same data (from the same** satellite and instrument) coming in different flavours. For example, we use variant to distinguish data coming from the satellite Metop-B from direct readout (no variant), regional coverage (EARS) or global coverage (GDS).

All the satellite configuration files in *PPP_CONFIG_DIR* should be named <*variant*><*platform name*>*.cfg*, e.g. *NOAA-19.cfg* or *GDSMetop-B.cfg*.

3.2 Creating a scene object

Creating a scene object can be done calling the *create_scene* function of a factory, (for example *mpop.satellites.GenericFactory.create_scene()*).

The reader is refered to the documentation of the *mpop.scene.SatelliteInstrumentScene()* for a description of the input arguments.

Such a scene object is roughly a container for *mpop.channel.Channel* objects, which hold the actual data and information for each band.

3.3 Loading the data

Loading the data is done through the *mpop.scene.SatelliteInstrumentScene.load()* method. Calling it effectively loads the data from disk into memory, so it can take a while depending on the volume of data to load and the performance of the host computer. The channels listed as arguments become loaded, and cannot be reloaded: a subsequent call to the method will not reload the data from disk.

3.4 Re-projecting data

Once the data is loaded, one might need to re-project the data. The scene objects can be projected onto other areas if the pyresample software is installed, thanks to the *mpop.scene.SatelliteInstrumentScene.project()* method. As input, this method takes either a Definition object (see pyresample's documentation) or string identificator for the area. In the latter case, the referenced region has to be defined in the area file. The name and location of this file is defined in the *mpop.cfg* configuration file, itself located in the directory pointed by the *PPP_CONFIG_DIR* environment variable.

For more information about the internals of the projection process, take a look at the `mpop.projector` module.

3.5 Geo-localisation of the data

Once the data is loaded, each channel should have an *area* attribute containing a pyresample area object, if the pyresample package is available. These area objects should implement the `get_lonlats()` method, returning the longitudes and latitudes of the channel data. For more information on this matter, the reader is then referred to the documentation of the aforementioned package.

3.6 Image composites

Methods building image composites are distributed in different modules, taking advantage of the hierarchical structure offered by OOP.

The image composites common to all visir instruments are defined in the *mpop.instruments.visir* module. Some instrument modules, like `mpop.instruments.avhrr` or `mpop.instruments.seviri` overload these methods to adapt better for the instrument at hand.

For instructions on how to write a new composites, see *Geographic images*.

3.7 Adding a new satellite: configuration file

A satellite configuration file looks like the following (here Meteosat-7, mviri instrument):

The configuration file must hold a *satellite* section, the list of channels for the needed instruments (here *mviri-n* sections), and how to read the data in mipp (*mviri-level1*) and how to read it in mpop (*mviri-level2*).

Using this template we can define new satellite and instruments.

3.8 Adding a new satellite: python code

Another way of adding satellites and instruments to mpop is to write the correponding python code.

Here are example of such code:

3.9 The mpop API

3.9.1 Satellite scenes

The *mpop.scene* module defines satellite scenes. They are defined as generic classes, to be inherited when needed.

A scene is a set of *mpop.channel* objects for a given time, and sometimes also for a given area.

class mpop.scene.**Satellite**

This is the satellite class. It contains information on the satellite.

classmethod add_method(*func*)

Add a method to the class.

add_method_to_instance(*func*)

Add a method to the instance.

fullname

Full name of the satellite, that is platform name and number (eg "metop02").

classmethod remove_attribute(*name*)

Remove an attribute from the class.

sat_nr(*string=False*)

class mpop.scene.**SatelliteInstrumentScene**(*time_slot=None*, *area_id=None*, *area=None*, *orbit=None*, *satellite=(None*, *None*, *None)*, *instrument=None*)

This is the satellite instrument class. It is an abstract channel container, from which all concrete satellite scenes should be derived.

The constructor accepts as optional arguments the *time_slot* of the scene, the *area* on which the scene is defined (this can be used for slicing of big datasets, or can be set automatically when loading), and *orbit* which is a string giving the orbit number.

add_to_history(*message*)

Adds a message to history info.

channel_list = []

check_channels(**channels*)

Check if the *channels* are loaded, raise an error otherwise.

estimate_cth(*cth_atm='best'*, *time_slot=None*)

Estimation of the cloud top height using the 10.8 micron channel limitations: this is the most simple approach a simple fit of the ir108 to the temperature profile

- no correction for water vapour or any other trace gas
- no viewing angle dependency
- no correction for semi-transparent clouds
- no special treatment of temperature inversions

data.estimate_cth(cth_atm="best")

cth_atm * using temperature profile to estimate the cloud top height

possible choices are (see estimate_cth in mpop/tools.py): "standard", "tropics", "midlatitude summer", "midlatitude winter", "subarctic summer", "subarctic winter" this will choose the corresponding atmospheric AFGL temperature profile

- new choice: "best" -> choose according to central (lon,lat) and time from: "tropics", "midlatitude summer", "midlatitude winter", "subarctic summer", "subarctic winter"

time_slot current observation time as (datetime.datetime() object) time_slot option can be omitted, the function tries to use self.time_slot

get_orbital()

load(*channels=None*, *load_again=False*, *area_extent=None*, ***kwargs*)

Load instrument data into the *channels*. *Channels* is a list or a tuple containing channels we will load data into, designated by there center wavelength (float), resolution (integer) or name (string). If None, all channels are loaded.

The *load_again* boolean flag allows to reload the channels even they have already been loaded, to mirror changes on disk for example. This is false by default.

The *area_extent* keyword lets you specify which part of the data to load. Given as a 4-element sequence, it defines the area extent to load in satellite projection.

The other keyword arguments are passed as is to the reader plugin. Check the corresponding documentation for more details.

loaded_channels()

Return the set of loaded_channels.

parallax_corr(*fill='False'*, *estimate_cth=False*, *cth_atm='best'*, *replace=False*)

perform the CTH parallax corretion for all loaded channels

project(*dest_area*, *channels=None*, *precompute=False*, *mode=None*, *radius=None*, *nprocs=1*)

Make a copy of the current snapshot projected onto the *dest_area*. Available areas are defined in the region configuration file (ACPG). *channels* tells which channels are to be projected, and if None, all channels are projected and copied over to the return snapshot.

If *precompute* is set to true, the projecting data is saved on disk for reusage. *mode* sets the mode to project in: 'quick' which works between cartographic projections, and, as its denomination indicates, is quick (but lower quality), and 'nearest' which uses nearest neighbour for best projection. A *mode* set to None uses 'quick' when possible, 'nearest' otherwise.

radius defines the radius of influence for neighbour search in 'nearest' mode (in metres). Setting it to None, or omitting it will fallback to default values (5 times the channel resolution) or 10,000m if the resolution is not available.

Note: channels have to be loaded to be projected, otherwise an exception is raised.

save(*filename*, *to_format='netcdf4'*, ***options*)

Saves the current scene into a file of format *to_format*. Supported formats are:

- *netcdf4*: NetCDF4 with CF conventions.

unload(**channels*)

Unloads *channels* from memory. *mpop.scene.SatelliteInstrumentScene.load()* must be called again to reload the data.

class mpop.scene.**SatelliteScene**(*time_slot=None*, *area_id=None*, *area=None*, *orbit=None*, *satellite=(None*, *None*, *None)*)
This is the satellite scene class. It is a capture of the satellite (channels) data at given *time_slot* and *area_id/area*.

area
Getter for area.

get_area()
Getter for area.

set_area(*area*)
Setter for area.

mpop.scene.**assemble_segments**(*segments*)
Assemble the scene objects listed in *segment_list* and returns the resulting scene object.

3.9.2 Instrument channels

This module defines satellite instrument channels as a generic class, to be inherited when needed.

class mpop.channel.**Channel**(*name=None, resolution=0, wavelength_range=[0, 0, 0], data=None, calibration_unit=None*)
This is the satellite channel class. It defines satellite channels as a container for calibrated channel data.

The *resolution* sets the resolution of the channel, in meters. The *wavelength_range* is a triplet, containing the lowest-, center-, and highest-wavelength values of the channel. *name* is simply the given name of the channel, and *data* is the data it should hold.

as_image(*stretched=True*)
Return the channel as a *mpop.imageo.geo_image.GeoImage* object. The *stretched* argument set to False allows the data to remain untouched (as opposed to crude stretched by default to obtain the same output as *show()*).

check_range(*min_range=1.0*)
Check that the data of the channels has a definition domain broader than *min_range* and return the data, otherwise return zeros.

data
Getter for channel data.

get_data()
Getter for channel data.

get_reflectance(*tb11*, *sun_zenith=None*, *tb13_4=None*)
Get the reflectance part of an NIR channel

get_viewing_geometry(*orbital*, *time_slot*, *altitude=None*)
Calculates the azimuth and elevation angle as seen by the observer at the position of the current area pixel. inputs:

orbital an orbital object define by the tle file (see pyorbital.orbital import Orbital or mpop/scene.py get_oribtal)

time_slot time object specifying the observation time altitude optinal: altitude of the observer above the earth ellipsoid

outputs: azi azimuth viewing angle in degree (south is 0, counting clockwise) ele elevation viewing angle in degree (zenith is 90, horizon is 0)

is_loaded()
Tells if the channel contains loaded data.

parallax_corr(*cth=None*, *time_slot=None*, *orbital=None*, *azi=None*, *ele=None*, *fill='False'*)
Perform the parallax correction for channel at *time_slot* (datetime.datetime() object), assuming the cloud top height cth and the viewing geometry given by the satellite orbital "orbital" and return the corrected channel. Authors: Ulrich Hamann (MeteoSwiss), Thomas Leppelt (DWD) Example calls:

- **calling this function (using orbital and time_slot)** orbital = data.get_oribtal() data["VIS006"].parallax_corr(cth=data["CTTH"].height, time_slot=data.time_slot, orbital=orbital)
- **calling this function (using viewing geometry)** orbital = data.get_oribtal() (azi, ele) = get_viewing_geometry(self, orbital, time_slot) data["VIS006"].parallax_corr(cth=data["CTTH"].height, azi=azi, ele=ele)

Optional input:

cth The parameter cth is the cloud top height (or the altitude of the object that should be shifted). cth must have the same size and projection as the channel

orbital an orbital object define by the tle file (see pyorbital.orbital import Orbital or mpop/scene.py get_oribtal)

azi azimuth viewing angle in degree (south is 0, counting clockwise) e.g. as given by self.get_viewing_geometry

ele elevation viewing angle in degree (zenith is 90, horizon is 0) e.g. as given by self.get_viewing_geometry

fill specifies the interpolation method to fill the gaps (basically areas behind the cloud that can't be observed by the satellite instrument) "False" (default): no interpolation, gaps are np.nan values and mask is set accordingly "nearest": fill gaps with nearest neighbour "bilinear": use scipy.interpolate.griddata with linear interpolation

to fill the gaps

output:

parallax corrected channel the content of the channel will be parallax corrected. The name of the new channel will be *original_chan.name+'_PC'*, eg. "IR_108_PC". This name is also stored to the info dictionary of the originating channel.

project(*coverage_instance*)
Make a projected copy of the current channel using the given *coverage_instance*.

See also the `mpop.projector` module.

set_data(*data*)
Setter for channel data.

shape
Shape of the channel.

show()
Display the channel as an image.

sunzen_corr(*time_slot*, *lonlats=None*, *limit=80.0*, *mode='cos'*, *sunmask=False*)
Perform Sun zenith angle correction for the channel at *time_slot* (datetime.datetime() object) and return the corrected channel. The parameter *limit* can be used to set the maximum zenith angle for which the correction is calculated. For larger angles, the correction is the same as at the *limit* (default: 80.0 degrees).

Coordinate values can be given as a 2-tuple or a two-element list *lonlats* of numpy arrays; if None, the coordinates will be read from the channel data. Parameter *mode* is a placeholder for other possible illumination corrections. The name of the new channel will be *original_chan.name+'_SZC'*, eg. "VIS006_SZC". This name is also stored to the info dictionary of the originating channel.

viewzen_corr(*view_zen_angle_data*)
Apply atmospheric correction on a copy of this channel data using the given satellite zenith angle data of the same shape. Returns a new channel containing the corrected data. The name of the new channel will be *original_chan.name+'_VZC'*, eg. "IR108_VZC". This name is also stored to the info dictionary of the originating channel.

vinc_vect(*lembda*, *alpha*, *s*, *f=None*, *a=None*, *degree=True*)
Vincenty's Direct formular

Returns the lat and long of projected point and reverse azimuth given a reference point and a distance and azimuth to project. lats, longs and azimuths are passed in radians.

Keyword arguments: phi Latitude in degree/radians lembda Longitude in degree/radians alpha Geodetic azimuth in degree/radians s Ellipsoidal distance in meters f WGS84 parameter a WGS84 parameter degree Boolean if in/out values are in degree or radians.

Default is in degree

Returns: (phiout, lembdaout, alphaout) as a tuple

class mpop.channel.**GenericChannel**(*name=None*)
This is an abstract channel class. It can be a super class for calibrated channels data or more elaborate channels such as cloudtype or CTTH.

area
Getter for area.

exception mpop.channel.**GeolocationIncompleteError**
Exception to try catch cases where the original data have not been read or expanded properly so that each pixel has a geo-location

exception mpop.channel.**NotLoadedError**
Exception to be raised when attempting to use a non-loaded channel.

3.9.3 The VisIr instrument class

This module defines the generic VISIR instrument class.

class mpop.instruments.visir.**VisirCompositer**(*scene*)
Compositer for Visual-IR instruments

airmass(*fill_value=(0, 0, 0)*)
Make an airmass RGB image composite.

Channels	Temp	Gamma
WV6.2 - WV7.3	-25 to 0 K	gamma 1
IR9.7 - IR10.8	-40 to 5 K	gamma 1
WV6.2	243 to 208 K	gamma 1

ash(*fill_value=(0, 0, 0)*)
Make a Ash RGB image composite.

Channels	Temp	Gamma
IR12.0 - IR10.8	-4 to 2 K	gamma 1
IR10.8 - IR8.7	-4 to 5 K	gamma 1
IR10.8	243 to 303 K	gamma 1

channel_image (*channel*, *fill_value=0*)
Make a black and white image of the *channel*.

Linear stretch without clipping is applied by default.

cloudtop (*stretch=(0.005, 0.005)*, *gamma=None*, *fill_value=(0, 0, 0)*)
Make a Cloudtop RGB image composite.

Channels	Gamma
IR3.9 (inverted)	gamma 1
IR10.8 (inverted)	gamma 1
IR12.0 (inverted)	gamma 1

Linear stretch with 0.5 % clipping at both ends.

convection (*fill_value=(0, 0, 0)*)
Make a Severe Convection RGB image composite.

Channels	Span	Gamma
WV6.2 - WV7.3	-30 to 0 K	gamma 1
IR3.9 - IR10.8	0 to 55 K	gamma 1
IR1.6 - VIS0.6	-70 to 20 %	gamma 1

dust (*fill_value=(0, 0, 0)*)
Make a Dust RGB image composite.

Channels	Temp	Gamma
IR12.0 - IR10.8	-4 to 2 K	gamma 1
IR10.8 - IR8.7	0 to 15 K	gamma 2.5
IR10.8	261 to 289 K	gamma 1

fog (*fill_value=(0, 0, 0)*)
Make a Fog RGB image composite.

Channels	Temp	Gamma
IR12.0 - IR10.8	-4 to 2 K	gamma 1
IR10.8 - IR8.7	0 to 6 K	gamma 2.0
IR10.8	243 to 283 K	gamma 1

green_snow (*fill_value=(0, 0, 0)*)
Make a Green Snow RGB image composite.

Channels	Gamma
IR1.6	gamma 1.6
VIS0.6	gamma 1.6
IR10.8 (inverted)	gamma 1.6

Linear stretch without clipping.

ir108()
Make a black and white image of the IR 10.8um channel.

Channel is inverted. Temperature range from -70 °C (white) to +57.5 °C (black) is shown.

natural(*stretch=None*, *gamma=1.8*, *fill_value=(0, 0, 0)*)
Make a Natural Colors RGB image composite.

Channels	Range (reflectance)	Gamma (default)
IR1.6	0 - 90	gamma 1.8
VIS0.8	0 - 90	gamma 1.8
VIS0.6	0 - 90	gamma 1.8

night_fog(*fill_value=(0, 0, 0)*)
Make a Night Fog RGB image composite.

Channels	Temp	Gamma
IR12.0 - IR10.8	-4 to 2 K	gamma 1
IR10.8 - IR3.9	0 to 6 K	gamma 2.0
IR10.8	243 to 293 K	gamma 1

night_overview(*stretch='histogram'*, *gamma=None*)
Make an overview RGB image composite using IR channels.

Channels	Gamma
IR3.9 (inverted)	gamma 1
IR10.8 (inverted)	gamma 1
IR12.0 (inverted)	gamma 1

Histogram equalization is applied for each channel.

overview(*stretch='crude'*, *gamma=1.6*, *fill_value=(0, 0, 0)*)
Make an overview RGB image composite.

Channels	Gamma (default)
VIS0.6	gamma 1.6
VIS0.8	gamma 1.6
IR10.8 (inverted)	gamma 1.6

Linear stretch without clipping is applied.

overview_sun(*stretch='linear'*, *gamma=1.6*, *fill_value=(0, 0, 0)*)
Make an overview RGB image composite normalising with cosine to the sun zenith angle.

red_snow(*fill_value=(0, 0, 0)*)
Make a Red Snow RGB image composite.

Channels	Gamma
VIS0.6	gamma 1.6
IR1.6	gamma 1.6
IR10.8 (inverted)	gamma 1.6

Linear stretch without clipping.

vis06()
Make a black and white image of the VIS 0.635um channel.

Linear stretch without clipping is applied.

wv_high()
Make a black and white image of the IR 6.7um channel.

Channel inverted and a linear stretch is applied with 0.5 % clipping at both ends.

wv_low()
Make a black and white image of the IR 7.3um channel.

Channel data inverted and a linear stretch is applied with 0.5 % clipping at both ends.

3.9.4 Projection facility

3.9.5 Satellite class loader

mpop.satellites is the module englobes all satellite specific modules. In itself, it hold the mighty `mpop.satellites.get_satellite_class()` method.

class mpop.satellites.**GenericFactory**
Factory for generic satellite scenes.

static create_scene(*satname*, *satnumber*, *instrument*, *time_slot*, *orbit*, *area=None*, *variant=''*)
Create a compound satellite scene.

class mpop.satellites.**GeostationaryFactory**
Factory for geostationary satellite scenes.

static create_scene(*satname*, *satnumber*, *instrument*, *time_slot*, *area=None*, *variant=''*)
Create a compound satellite scene.

class mpop.satellites.**PolarFactory**
Factory for polar satellite scenes.

static create_scene(*satname*, *satnumber*, *instrument*, *time_slot*, *orbit=None*, *area=None*, *variant=''*)
Create a compound satellite scene.

mpop.satellites.**build_instrument_compositer**(*instrument_name*)
Automatically generate an instrument compositer class from its *instrument_name*. The class is then filled with custom composites if there are any (see *get_custom_composites()*)

mpop.satellites.**build_sat_instr_compositer**()
Build a compositer class for the given satellite (defined by the three strings *satellite*, *number*, and *variant*) and *instrument* on the fly, using data from a corresponding config file. They inherit from the corresponding instrument class, which is also created on the fly is no predefined module (containing a compositer) for this instrument is available (see *build_instrument_compositer()*).

mpop.satellites.**get_custom_composites**(*name*)
Get the home made methods for building composites for a given satellite or instrument *name*.

mpop.satellites.**get_sat_instr_compositer**()
Get the compositer class for a given satellite, defined by the three strings *satellite*, *number*, and *variant*, and *instrument*. The class is then filled with custom composites if there are any (see *get_custom_composites()*). If no class is found, an attempt is made to build the class from a corresponding configuration file, see *build_sat_instr_compositer()*.

3.9.6 Miscellaneous tools

Helper functions for eg. performing Sun zenith angle correction.

mpop.tools.**estimate_cth**(*IR_108*, *cth_atm='standard'*)

Estimation of the cloud top height using the 10.8 micron channel limitations: this is the most simple approach

a simple fit of the ir108 to the temperature profile * no correction for water vapour or any other trace gas * no viewing angle dependency * no correction for semi-transparent clouds

optional input:

cth_atm * "standard", "tropics", "midlatitude summer", "midlatitude winter", "subarctic summer", "subarctic wint

Matching the 10.8 micron temperature with atmosphere profile (s) AFGL atmospheric constituent profile. U.S. standard atmosphere 1976. (AFGL-TR-86-0110) (t) AFGL atmospheric constituent profile. tropical. (AFGL-TR-86-0110) (mw) AFGL atmospheric constituent profile. midlatitude summer. (AFGL-TR-86-0110) (ms) AFGL atmospheric constituent profile. midlatitude winter. (AFGL-TR-86-0110) (ss) AFGL atmospheric constituent profile. subarctic summer. (AFGL-TR-86-0110) (sw) AFGL atmospheric constituent profile. subarctic winter. (AFGL-TR-86-0110) Ulrich Hamann (MeteoSwiss)

- "tropopause" Assuming a fixed tropopause height and a fixed temperature gradient Richard Mueller (DWD)

output:

parallax corrected channel the content of the channel will be parallax corrected. The name of the new channel will be *original_chan.name+'_PC'*, eg. "IR_108_PC". This name is also stored to the info dictionary of the originating channel.

Versions: 05.07.2016 initial version Ulrich Hamann (MeteoSwiss), Richard Mueller (DWD)

mpop.tools.**sunzen_corr_cos**(*data*, *cos_zen*, *limit=80.0*)

Perform Sun zenith angle correction to the given *data* using cosine of the zenith angle (*cos_zen*). The correction is limited to *limit* degrees (default: 80.0 degrees). For larger zenith angles, the correction is the same as at the *limit*. Both *data* and *cos_zen* are given as 2-dimensional Numpy arrays or Numpy MaskedArrays, and they should have equal shapes.

mpop.tools.**viewzen_corr**(*data*, *view_zen*)

Apply atmospheric correction on the given *data* using the specified satellite zenith angles (*view_zen*). Both input data are given as 2-dimensional Numpy (masked) arrays, and they should have equal shapes. The *data* array will be changed in place and has to be copied before.

CHAPTER 4

Input plugins: the `mpop.satin` package

4.1 Available plugins and their requirements

4.1.1 mipp_xrit

Reader for for hrit/lrit formats. Recommends numexpr and pyresample.

4.1.2 aapp1b

Reader for AAPP level 1b format. Requires numpy, recommends pyresample.

Reader for aapp level 1b data.

Options for loading:

- pre_launch_coeffs (False): use pre-launch coefficients if True, operational otherwise (if available).

http://research.metoffice.gov.uk/research/interproj/nwpsaf/aapp/ NWPSAF-MF-UD-003_Formats.pdf

class mpop.satin.aapp1b.**AAPP1b**(*fname*)

AAPP-level 1b data reader

calibrate(*chns=('1', '2', '3A', '3B', '4', '5')*, *calibrate=1*, *pre_launch_coeffs=False*, *calib_coeffs=None*)

Calibrate the data

navigate()

Return the longitudes and latitudes of the scene.

read()

Read the data.

mpop.satin.aapp1b.**load**(*satscene*, **args*, ***kwargs*)

Read data from file and load it into *satscene*. A possible *calibrate* keyword argument is passed to the AAPP reader. Should be 0 for off (counts), 1 for default (brightness temperatures and reflectances), and 2 for radiances only.

If *use_extern_calib* keyword argument is set True, use external calibration data.

mpop.satin.aapp1b.**load_avhrr**(*satscene*, *options*)
Read avhrr data from file and load it into *satscene*.

mpop.satin.aapp1b.**main**()

mpop.satin.aapp1b.**show**(*data*, *negate=False*)
Show the stetched data.

4.1.3 eps_l1b

Reader for EPS level 1b format. Recommends pyresample.

Reader for eps level 1b data. Uses xml files as a format description. See: http://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=PDF_TEN_97231-EPS-AVHRR&RevisionSelectionMethod=LatestReleased&Rendition=Web and http://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=PDF_TEN_990004-EPS-AVHRR1-PGS&RevisionSelectionMethod=LatestReleased&Rendition=Web

class mpop.satin.eps_l1b.**EpsAvhrrL1bReader**(*filename*)
Eps level 1b reader for AVHRR data.

get_channels(*channels*, *calib_type*)
Get calibrated channel data. *calib_type* = 0: Counts *calib_type* = 1: Reflectances and brightness temperatures *calib_type* = 2: Radiances

get_full_lonlats()
Get the interpolated lons/lats.

get_lonlat(*row*, *col*)
Get lons/lats for given indices. WARNING: if the lon/lats were not expanded, this will refer to the tiepoint data.

keys()
List of reader's keys.

mpop.satin.eps_l1b.**get_corners**(*filename*)
Get the corner lon/lats of the file.

mpop.satin.eps_l1b.**get_filename**(*satscene*, *level*)
Get the filename.

mpop.satin.eps_l1b.**get_lonlat**(*scene*, *row*, *col*)
Get the longitutes and latitudes for the give *rows* and *cols*.

mpop.satin.eps_l1b.**load**(*scene*, **args*, ***kwargs*)
Loads the *channels* into the satellite *scene*. A possible *calibrate* keyword argument is passed to the AAPP reader Should be 0 for off, 1 for default, and 2 for radiances only. However, as the AAPP-lvl1b file contains radiances this reader cannot return counts, so calibrate=0 is not allowed/supported. The radiance to counts conversion is not possible.

mpop.satin.eps_l1b.**norm255**(*a__*)
normalize array to uint8.

mpop.satin.eps_l1b.**read_raw**(*filename*)
Read *filename* without scaling it afterwards.

mpop.satin.eps_l1b.**show**(*a__*)
show array.

mpop.satin.eps_l1b.**to_bt**(*arr*, *wc_*, *a__*, *b__*)
Convert to BT.

mpop.satin.eps_l1b.**to_refl**(*arr*, *solar_flux*)
Convert to reflectances.

4.1.4 viirs_sdr

Reader for the VIIRS SDR format. Requires h5py.

Interface to VIIRS SDR format

Format documentation: http://npp.gsfc.nasa.gov/science/sciencedocuments/082012/474-00001-03_CDFCBVolIII_RevC.pdf

class mpop.satin.viirs_sdr.**GeolocationFlyweight**(*cls*)

clear_cache()

class mpop.satin.viirs_sdr.**HDF5MetaData**(*filename*)
Small class for inspecting a HDF5 file and retrieve its metadata/header data. It is developed for JPSS/NPP data but is really generic and should work on most other hdf5 files.

Supports

collect_metadata(*name*, *obj*)

get_data_keys()

keys()

read()

class mpop.satin.viirs_sdr.**NPPMetaData**(*filename*)

get_band_description()

get_begin_orbit_number()

get_begin_time()

get_brightness_temperature_keys()

get_end_orbit_number()

get_end_time()

get_geofilname()

get_radiance_keys()

get_reflectance_keys()

get_ring_lonlats()

get_shape()

get_unit(*calibrate=1*)

class mpop.satin.viirs_sdr.**ViirsBandData**(*filenames*, *calibrate=1*)
Placeholder for the VIIRS M&I-band data. Reads the SDR data - one hdf5 file for each band. Not yet considering the Day-Night Band

read()

read_lonlat(*geofilepaths=None*, *geodir=None*)

class mpop.satin.viirs_sdr.**ViirsSDRReader**(**args*, ***kwargs*)

get_elevation(***kwargs*)
Get elevation/topography for a given band type (M, I, or DNB) Optional arguments:

bandtype = 'M', 'I', or 'DNB'

Return elevation

get_sunsat_angles(***kwargs*)
Get sun-satellite viewing geometry for a given band type (M, I, or DNB) Optional arguments:

bandtype = 'M', 'I', or 'DNB'

Return sun-zenith, sun-azimuth, sat-zenith, sat-azimuth

load(*satscene*, *calibrate=1*, *time_interval=None*, *area=None*, *filename=None*, ***kwargs*)
Read viirs SDR reflectances and Tbs from file and load it into *satscene*.

pformat = 'viirs_sdr'

mpop.satin.viirs_sdr.**get_elevation_into**(*filename*, *out_height*, *out_mask*)
Read elevation/height from hdf5 file

mpop.satin.viirs_sdr.**get_lonlat_into**(*filename*, *out_lons*, *out_lats*, *out_height*, *out_mask*)
Read lon,lat from hdf5 file

mpop.satin.viirs_sdr.**get_viewing_angle_into**(*filename*, *out_val*, *out_mask*, *param*)
Read a sun-sat viewing angle from hdf5 file

mpop.satin.viirs_sdr.**globify**(*filename*)

4.1.5 viirs_compact

Reader for the VIIRS compact format from EUMETSAT. Requires h5py.

4.1.6 hdfeos_l1b

Reader for Modis data format. Requires pyhdf.

4.1.7 msg_hdf

Reader for MSG cloud products. Requires h5py, recommends acpg.

4.1.8 pps_hdf

Reader for PPS cloud products. Requires acpg.

Plugin for reading PPS's cloud products hdf files.

class mpop.satin.pps_hdf.**PpsCTTH**

copy(*other*)

read(*filename*)

class mpop.satin.pps_hdf.**PpsCloudType**

copy(*other*)

is_loaded()

read(*filename*)

mpop.satin.pps_hdf.**load**(*scene*, ***kwargs*)
Load data into the *channels*. *Channels* is a list or a tuple containing channels we will load data into. If None, all channels are loaded.

4.1.9 hrpt

Reader for level 0 hrpt format. Requires AAPP and pynav.

4.1.10 eps1a

Reader for level 1a Metop segments. Requires AAPP, kai and eugene.

4.2 Interaction with reader plugins

The reader plugin instance used for a specific scene is accessible through a scene attribute named after the plugin format. For example, the attribute for the *foo* format would be called *foo_reader*.

This way the other methods present in the plugins are available through the scene object.

4.3 The plugin API

Changed in version 0.13.0: New plugin API The *mpop.plugin_base* module defines the plugin API.

class mpop.plugin_base.**Plugin**
The base plugin class. It is not to be used as is, it has to be inherited by other classes.

class mpop.plugin_base.**Reader**(*scene*)
Reader plugins. They should have a *pformat* attribute, and implement the *load* method. This is an abstract class to be inherited.

load(*channels_to_load*)
Loads the *channels_to_load* into the scene object.

ptype = 'reader'

class mpop.plugin_base.**Writer**(*scene*)
Writer plugins. They must implement the *save* method. This is an abstract class to be inherited.

ptype = 'writer'

save(*filename*)
Saves the scene to a given *filename*.

4.4 Adding a new plugin

For now only reader and writer plugins base classes are defined.

To add one of those, just create a new class that subclasses the plugin.

The interface of any reader plugin must include the `load()` method.

Take a look at the existing readers for more insight.

CHAPTER 5

Geographic images

In order to build satellite composites, mpop has to handle images. We could have used PIL, but we felt the need to use numpy masked arrays as base for our image channels, and we had to handle geographically enriched images. Hence the two following modules: *mpop.imageo.image* to handle simple images, and *mpop.imageo.geo_image*.

5.1 Simple images

This module defines the image class. It overlaps largely the PIL library, but has the advandage of using masked arrays as pixel arrays, so that data arrays containing invalid values may be properly handled.

class mpop.imageo.image.**Image**(*channels=None*, *mode='L'*, *color_range=None*, *fill_value=None*, *palette=None*)

This class defines images. As such, it contains data of the different *channels* of the image (red, green, and blue for example). The *mode* tells if the channels define a black and white image ("L"), an rgb image ("RGB"), an YCbCr image ("YCbCr"), or an indexed image ("P"), in which case a *palette* is needed. Each mode has also a corresponding alpha mode, which is the mode with an "A" in the end: for example "RGBA" is rgb with an alpha channel. *fill_value* sets how the image is filled where data is missing, since channels are numpy masked arrays. Setting it to (0,0,0) in RGB mode for example will produce black where data is missing."None" (default) will produce transparency (thus adding an alpha channel) if the file format allows it, black otherwise.

The channels are considered to contain floating point values in the range [0.0,1.0]. In order to normalize the input data, the *color_range* parameter defines the original range of the data. The conversion to the classical [0,255] range and byte type is done automagically when saving the image to file.

clip(*channels=True*)
Limit the values of the array to the default [0,1] range. *channels* says which channels should be clipped.

convert(*mode*)
Convert the current image to the given *mode*. See *Image* for a list of available modes.

crude_stretch(*ch_nb*, *min_stretch=None*, *max_stretch=None*)
Perform simple linear stretching (without any cutoff) on the channel *ch_nb* of the current image and normalize to the [0,1] range.

enhance(*inverse=False*, *gamma=1.0*, *stretch='no'*)
Image enhancement function. It applies **in this order** inversion, gamma correction, and stretching to the current image, with parameters *inverse* (see `Image.invert()`), *gamma* (see `Image.gamma()`), and *stretch* (see `Image.stretch()`).

gamma(*gamma=1.0*)
Apply gamma correction to the channels of the image. If *gamma* is a tuple, then it should have as many elements as the channels of the image, and the gamma correction is applied elementwise. If *gamma* is a number, the same gamma correction is applied on every channel, if there are several channels in the image. The behaviour of `gamma()` is undefined outside the normal [0,1] range of the channels.

invert(*invert=True*)
Inverts all the channels of a image according to *invert*. If invert is a tuple or a list, elementwise invertion is performed, otherwise all channels are inverted if *invert* is true (default).

is_empty()
Checks for an empty image.

merge(*img*)
Use the provided image as background for the current *img* image, that is if the current image has missing data.

modes = ['L', 'LA', 'RGB', 'RGBA', 'YCbCr', 'YCbCrA', 'P', 'PA']

pil_image()
Return a PIL image from the current image.

pil_save(*filename*, *compression=6*, *fformat=None*)
Save the image to the given *filename* using PIL. For now, the compression level [0-9] is ignored, due to PIL's lack of support. See also `save()`.

putalpha(*alpha*)
Adds an *alpha* channel to the current image, or replaces it with *alpha* if it already exists.

replace_luminance(*luminance*)
Replace the Y channel of the image by the array *luminance*. If the image is not in YCbCr mode, it is converted automatically to and from that mode.

resize(*shape*)
Resize the image to the given *shape* tuple, in place. For zooming, nearest neighbour method is used, while for shrinking, decimation is used. Therefore, *shape* must be a multiple or a divisor of the image shape.

save(*filename*, *compression=6*, *fformat=None*)
Save the image to the given *filename*. For some formats like jpg and png, the work is delegated to `pil_save()`, which doesn't support the *compression* option.

show()
Display the image on screen.

stretch(*stretch='no'*, ***kwarg*)
Apply stretching to the current image. The value of *stretch* sets the type of stretching applied. The values "histogram", "linear", "crude" (or "crude-stretch") perform respectively histogram equalization, contrast stretching (with 5% cutoff on both sides), and contrast stretching without cutoff. The value "logarithmic" or "log" will do a logarithmic enhancement towards white. If a tuple or a list of two values is given as input, then a contrast stretching is performed with the values as cutoff. These values should be normalized in the range [0.0,1.0].

stretch_hist_equalize(*ch_nb*)
Stretch the current image's colors by performing histogram equalization on channel *ch_nb*.

stretch_linear(*ch_nb*, *cutoffs=(0.005, 0.005)*)
Stretch linearly the contrast of the current image on channel *ch_nb*, using *cutoffs* for left and right trimming.

stretch_logarithmic(*ch_nb*, *factor=100.0*)
Move data into range [1:factor] and do a normalized logarithmic enhancement.

exception mpop.imageo.image.**UnknownImageFormat**
Exception to be raised when image format is unknown to MPOP

mpop.imageo.image.**all**(*iterable*)

mpop.imageo.image.**check_image_format**(*fformat*)

mpop.imageo.image.**rgb2ycbcr**(*r__*, *g__*, *b__*)
Convert the three RGB channels to YCbCr.

mpop.imageo.image.**ycbcr2rgb**(*y__*, *cb_*, *cr_*)
Convert the three YCbCr channels to RGB channels.

5.2 Geographically enriched images

Module for geographic images.

class mpop.imageo.geo_image.**GeoImage**(*channels*, *area*, *time_slot*, *mode='L'*, *crange=None*, *fill_value=None*, *palette=None*)
This class defines geographic images. As such, it contains not only data of the different *channels* of the image, but also the area on which it is defined (*area* parameter) and *time_slot* of the snapshot.

The channels are considered to contain floating point values in the range [0.0,1.0]. In order to normalize the input data, the *crange* parameter defines the original range of the data. The conversion to the classical [0,255] range and byte type is done automagically when saving the image to file.

See also `image.Image` for more information.

add_overlay(*color=(0, 0, 0)*, *width=0.5*, *resolution=None*)
Add coastline and political borders to image, using *color* (tuple of integers between 0 and 255). Warning: Loses the masks !

resolution is chosen automatically if None (default), otherwise it should be one of: +—–+——————–+————+ | 'f' | Full resolution | 0.04 km | | 'h' | High resolution | 0.2 km | | 'i' | Intermediate resolution | 1.0 km | | 'l' | Low resolution | 5.0 km | | 'c' | Crude resolution | 25 km | +—–+——————–+————+

add_overlay_config(*config_file*)
Add overlay to image parsing a configuration file.

geotiff_save(*filename*, *compression=6*, *tags=None*, *gdal_options=None*, *blocksize=0*, *geotransform=None*, *spatialref=None*, *floating_point=False*, *writer_options=None*)
Save the image to the given *filename* in geotiff format, with the *compression* level in [0, 9]. 0 means not compressed. The *tags* argument is a dict of tags to include in the image (as metadata). By default it uses the 'area' instance to generate geotransform and spatialref information, this can be overwritten by the arguments *geotransform* and *spatialref*. *floating_point* allows the saving of 'L' mode images in floating point format if set to True. When argument *writer_options* is not none and entry 'fill_value_subst' is included, its numeric value will be used to substitute image data that would be equal to the fill_value (used to replace masked data).

save(*filename*, *compression=6*, *tags=None*, *gdal_options=None*, *fformat=None*, *blocksize=256*, *writer_options=None*, ***kwargs*)
Save the image to the given *filename*. If the extension is "tif", the image is saved to geotiff format, in which case the *compression* level can be given ([0, 9], 0 meaning off). See also `image.Image.save()`,

`image.Image.double_save()`, and `image.Image.secure_save()`. The *tags* argument is a dict of tags to include in the image (as metadata), and the *gdal_options* holds options for the gdal saving driver. A *blocksize* other than 0 will result in a tiled image (if possible), with tiles of size equal to *blocksize*. If the specified format *fformat* is not know to MPOP (and PIL), we will try to import module *fformat* and call the method *fformat.save*.

Use *writer_options* to define parameters that should be forwarded to custom writers. Dictionary keys listed in mpop.imageo.formats.writer_options will be interpreted by this function instead of *compression*, *blocksize* and nbits in *tags* dict.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Y