mpop documentation

Release v1.0.0

SMHI

December 13, 2013
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, MTSAT1R, MTSAT2, GOES 11, GOES 12, GOES 13 through the use of mipp, and Noaa 15, 16, 17, 18, 19, and Metop A through the use of aapp and ahamap.

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 retrieve a tarball from there if you prefer, then run:

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

Then you need to install mpop on your 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):
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.
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:

```python
>>> 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", "09", "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 retrieve 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.

```python
>>> 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 \( \mu m \)) were loaded. If we try to generate a day natural color composite, which requires also the 1.6um channel, it will result in an error:

```python
>>> 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:

```python
>>> global_data.load(global_data.image.natural.prerequisites, area_extent=europe.area_extent)
>>> img = global_data.image.natural()
```

Now you can save the image:

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

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

```python
>>> global_data.load(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:

```python
>>> print global_data[0.6]
‘VIS006: (0.560,0.635,0.710)\( \mu m \), shape (1200, 3000), resolution 3000.40316582m’
```
or from the channel name:

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

or from the resolution:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> global_data.area = "EuropeCanary"
>>> global_data.load(["CTTH"])
```
and they can be retrieved as simply as before:

```python
>>> 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:

```python
>>> 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=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:

```python
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")

    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,
```
```python
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]

Note the `seviri` variable in the end. This means that the composites it contains will be available to all scenes using the Seviri instrument. If we replace this by:

```python
meteosat09seviri = [overview,
                    hr_visual]
```

then the composites will only be available for the Meteosat 9 satellite scenes.

### 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:

```python
>>> 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:

```python
>>> 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()`.
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:

- **satellite name** will designate the name of the platform, e.g. “noaa” in satellite “noaa 19”
- **satellite number** will refer to the number of the satellite in the series, e.g. “19” for satellite “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 02 from direct readout, regional coverage or global coverage.

### 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 referred 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 identifier 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 corresponding 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((satname, number, variant)=(None, None, None))

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

    @classmethod
    def add_method(func)
        Add a method to the class.

    def add_method_to_instance(func)
        Add a method to the instance.

    def fullname
        Full name of the satellite, that is platform name and number (eg “metop02”).

    @classmethod
    def remove_attribute(name)
        Remove an attribute from the class.

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 use for slicing of big datasets, or can be set automatically when loading), and orbit which is a string giving the orbit number.

    def add_to_history(message)
        Adds a message to history info.

    @property
    def channel_list
        Return the set of loaded_channels.

    @property
    def check_channels(*channels)
        Check if the channels are loaded, raise an error otherwise.

    def 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.

    def loaded_channels()
        Return the set of loaded_channels.

    def project(dest_area, channels=None, precompute=False, mode=None, radius=None)
        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. Setting it to None, or omitting it will fallback to default values (5 times the channel resolution) or 10km if the resolution is not available.

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

```python
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.

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

```python
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`.

```python
def area()
    Getter for area.

def get_area ()
    Getter for area.

def set_area (area)
    Setter for area.
```

```python
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.

```python
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.

```python
def 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()`).
```

```python
def 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.
```

```python
def data
    Getter for channel data.
```
get_data()
    Getter for channel data.

is_loaded()
    Tells if the channel contains loaded data.

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.

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.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)

airmass()
    Make an airmass RGB image composite.

<table>
<thead>
<tr>
<th>Channels</th>
<th>Temp</th>
<th>Gamma</th>
</tr>
</thead>
<tbody>
<tr>
<td>WV6.2 - WV7.3</td>
<td>-25 to 0 K</td>
<td>gamma 1</td>
</tr>
<tr>
<td>IR9.7 - IR10.8</td>
<td>-40 to 5 K</td>
<td>gamma 1</td>
</tr>
<tr>
<td>WV6.2</td>
<td>243 to 208 K</td>
<td>gamma 1</td>
</tr>
</tbody>
</table>

ash()
    Make a Ash RGB image composite.

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

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

convection()
    Make a Severe Convection RGB image composite.

dust()
    Make a Dust RGB image composite.

fog()
    Make a Fog RGB image composite.
green_snow()
    Make a Green Snow RGB image composite.

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

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

night_fog()
    Make a Night Fog RGB image composite.

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

red_snow()
    Make a Red Snow RGB image composite.

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

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

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

3.9.4 Projection facility

This module handles coverage objects. Such objects are used to transform area projected data by changing either the
area or the projection or both. A typical usage is to transform one large area in satellite projection to an area of interest
in polar projection for example.

class mpop.projector.Projector(in_area, out_area=None, in_latlons=None, mode=None, radius=10000)
    This class define projector objects. They contain the mapping information necessary for projection purposes.
    For efficiency reasons, generated projectors can be saved to disk for later reuse. Use the save() method for
    this.

    To define a projector object, on has to specify in_area and out_area, and can also input the in_latlons or the
    mode (‘quick’ which works only if both in- and out-areas are AreaDefinitions, or ‘nearest’). radius defines the
    radius of influence for nearest neighbour search in ‘nearest’ mode.

    in_area = None
    mode = ‘quick’
    out_area = None

project_array(data)
    Project an array data along the given Projector object.

save(resave=False)
    Save the precomputation to disk, and overwrite existing file in case resave is true.

mpop.projector.get_area_def(area_name)
    Get the definition of area_name from file. The file is defined to use is to be placed in the $PPP_CONFIG_DIR
directory, and its name is defined in mpop’s configuration file.
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((satellite, number, variant), instrument)
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((satellite, number, variant), instrument)
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 corre-
spanding configuration file, see build_sat_instr_compositer().
Input plugins: the mpop.satin package

4.1 Available plugins and their requirements

4.1.1 mipp_xrit

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

4.1.2 aapp1b

Reader for aapp level 1b format. Requires AHAMAP, recommends pyresample. Reader for aapp level 1b data.

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

```python
class mpop.satin.aapp1b.AAPP1b (fname)
    AAPP-level 1b data reader
    
    calibrate (chns=('1', '2', '3A', '3B', '4', '5'), calibrate=1)
        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.

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

mpop.satin.aapp1b.show (data, negate=False)
    Show the stetched data.
```
4.1.3 eps_format

Reader for eps level 1b format. Recommends pyresample.

4.1.4 thin_modis

Reader for thinned modis format (as send via Eumetcast). Require pyhdf.

4.1.5 msg_hdf

Reader for msg cloud products. Requires pytable and acpg.

4.1.6 pps_hdf

Reader for pps cloud products. Requires acpg. Plugin for reading PPS’s cloud products hdf files.

```python
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.
```

4.1.7 hrpt

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

4.1.8 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.

```python
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 advantage of using masked arrays as pixel arrays, so that data arrays containing invalid values may be properly handled.

```python
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.

    channels = None

    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 ()).

fill_value = None

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.

height = 0

invert (invert=True)
Inverts all the channels of a image according to invert. If invert is a tuple or a list, elementwise inversion 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.

mode = None

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

palette = None

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.

shape = None
Shape (dimensions) of the image.

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.

```
width = 0
```

`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 automatically 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   | +--------+
```

```
geotiff_save(filename, compression=6, tags=None, gdal_options=None, blocksize=0, geotransform=None, spatialref=None, floating_point=False)
```
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.
save(filename, compression=6, tags=None, gdal_options=None, fformat=None, blocksize=256, **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.
Tools for batch production

6.1 Running stuff

Batch run stuff.

```python
class mpop.saturn.runner.SequentialRunner(satellite, instrument, tasklist, precompute=False)
    Runs scenes in a sequential order, as opposed to parallelized running.

    run_from_cmd()
    Batch run mpop.

    run_from_data(tasklist=None, radius=None)
    Run on given data.

    run_from_local_data(tasklist=None, extra_tags=None)
    Run on given local data (already projected).

    stop()
    Stops the runner.
```

```python
mpop.saturn.runner.parse_options()
    Parse command line options.

mpop.saturn.runner.usage(scriptname)
    Print useful information for running the script.
```

6.2 Handling tasks

Tasklist class and helper function.

```python
class mpop.saturn.tasklist.TaskList(product_file=None)
    Defines a tasklist.

    get_prerequisites(klass, area_id=None)
    Get the channels we need to load to fulfill the tasklist according to methods defined in klass. If area is provided, account only for tasks on this area.

    shape(klass, mode=set([]), original_areas=set([]), specific_composites=set([]))
    Shape the given the tasklist according to the options.
```
split (*keys)

Split the tasklist along the keys parameter: keys in the first part, non keys in the second.

mpop.saturn.tasklist.get_product_list (satscene)

Returns the tasklist corresponding to the satellite described in satscene, which can be a scene object, a list or a tuple. If the corresponding file could not be found, the function returns more generic tasklists (variant and name based, then only variant based), or None if no file can be found.

NB: the product files are looked for in the CONFIG_PATH directory.

6.3 Monitoring and processing incoming files

Watch files coming in a given directory.

class mpop.saturn.filewatcher.FileProcessor (file_queue, fun, refresh=None)

Execute fun on filenames provided by from file_queue. If refresh is a positive number, run fun every given number of seconds with None as argument.

run ()

Execute the given function on files from the file queue.

stop ()

Stops a running process.

terminate ()

Terminate thread.

class mpop.saturn.filewatcher.FileWatcher (filename_template, file_queue, frequency)

Looks for new files, and queues them.

run ()

Run the file watcher.

terminate ()

Terminate thread.

wait (secs)

6.4 Earth geometry

Spherical geometry module.

class mpop.saturn.assemble_segments.Arc (start, end)

An arc of the great circle between two points.

angle (other_arc)

Oriented angle between two arcs.

center_angle ()

Angle of an arc at the center of the sphere.

end = None

intersection (other_arc)

Says where, if two arcs defined by the current arc and the other_arc intersect. An arc is defined as the shortest tracks between two points.

intersections (other_arc)

Gives the two intersections of the greats circles defined by the current arc and other_arc.
intersects (other_arc)
Says if two arcs defined by the current arc and the other_arc intersect. An arc is defined as the shortest tracks between two points.

start = None
class mpop.saturn.assemble_segments.Coordinate (lat=None, lon=None, x__=None, y__=None, z__=None, R=1)
Point on earth in terms of lat and lon.
cross (point)
cross product with another vector.
cross2cart (point)
Compute the cross product, and convert to cartesian coordinates (assuming radius 1).
distance (point)
Vincenty formula.
dot (point)
dot product with another vector.
lat = None
lon = None
norm ()
Return the norm of the vector.
normalize ()
normalize the vector.
x__ = None
y__ = None
z__ = None
class mpop.saturn.assemble_segments.TestSphereGeometry (methodName='runTest')
Testing sphere geometry from this module.
test_angle ()
Testing the angle value between two arcs.
test_inside ()
Testing if a point is inside for other points.
test_intersects ()
Test if two arcs intersect.
test_overlap_rate ()
Test how much two areas overlap.
test_overlaps ()
Test if two areas overlap.
mpop.saturn.assemble_segments.get_area (corners)
Get the area of the convex area defined by corners.
mpop.saturn.assemble_segments.get_first_intersection (b__, boundaries)
Get the first intersection on b__ with boundaries.
mpop.saturn.assemble_segments.get_intersections (b__, boundaries)
Get the intersections of b__ with boundaries. Returns both the intersection coordinates and the concerned boundaries.
mpop.saturn.assemble_segments.get_next_intersection(p__, b__, boundaries)
 Get the next intersection from the intersection of arcs p__ and b__ along segment b__ with boundaries.

mpop.saturn.assemble_segments.min_distances(area_corners, segment_corners)
 Min distances between each corner of area_corners and segment_corners.

mpop.saturn.assemble_segments.modpi(val)
 Puts val between -pi and pi.

mpop.saturn.assemble_segments.modpi2(val)
 Puts val between 0 and 2pi.

mpop.saturn.assemble_segments.overlap_rate(swath_corners, area_corners)
 Get how much a swath overlaps an area.

mpop.saturn.assemble_segments.overlaps(area_corners, segment_corners)
 Are two areas overlapping? This uses great circle arcs as area boundaries.

mpop.saturn.assemble_segments.point_inside(point, corners)
 Is a point inside the 4 corners? This uses great circle arcs as area boundaries.

mpop.saturn.assemble_segments.polygon(area_corners, segment_corners)
 Get the intersection polygon between two areas.

mpop.saturn.assemble_segments.should_wait(area_corners, segment_corners, previous_segment_corners)
 Are the newest corner still inside the area? is the last segment boundary overlapping any boundary of the area?
 In this case we should wait for the next segment to arrive.
Indices and tables

- genindex
- modindex
- search
m
mpop.channel,??
mpop.imageo.geo_image,??
mpop.imageo.image,??
mpop.instruments.visir,??
mpop.plugin_base,??
mpop.projector,??
mpop.satellites,??
mpop.satin.aappib,??
mpop.satin.pps_hdf,??
mpop.saturn.assemble_segments,??
mpop.saturn.filewatcher,??
mpop.saturn.runner,??
mpop.saturn.tasklist,??
mpop.scene,??