Satpy Documentation

Release 0.45.1.dev0+ge338294dd.d20231129

Satpy Developers

Nov 29, 2023

CONTENTS

Satpy is a python library for reading, manipulating, and writing data from remote-sensing earth-observing satellite instruments. Satpy provides users with readers that convert geophysical parameters from various file formats to the common Xarray `DataArray` and `Dataset` classes for easier interoperability with other scientific python libraries. Satpy also provides interfaces for creating RGB (Red/Green/Blue) images and other composite types by combining data from multiple instrument bands or products. Various atmospheric corrections and visual enhancements are provided for improving the usefulness and quality of output images. Output data can be written to multiple output file formats such as PNG, GeoTIFF, and CF standard NetCDF files. Satpy also allows users to resample data to geographic projected grids (areas). Satpy is maintained by the open source Pytroll group.

The Satpy library acts as a high-level abstraction layer on top of other libraries maintained by the Pytroll group including:

- pyresample
- pyspectral
- trollimage
- pycoast
- pydecorate
- python-geotiepoints
- pyninjotiff

Go to the Satpy project page for source code and downloads.

Satpy is designed to be easily extendable to support any earth observation satellite by the creation of plugins (readers, compositors, writers, etc). The table at the bottom of this page shows the input formats supported by the base Satpy installation.

Note: Satpy's interfaces are not guaranteed stable and may change until version 1.0 when backwards compatibility will be a main focus.

CHAPTER

ONE

GETTING HELP

Having trouble installing or using Satpy? Feel free to ask questions at any of the contact methods for the PyTroll group here or file an issue on Satpy's GitHub page.

CHAPTER

TWO

DOCUMENTATION

2.1 Overview

Satpy is designed to provide easy access to common operations for processing meteorological remote sensing data. Any details needed to perform these operations are configured internally to Satpy meaning users should not have to worry about *how* something is done, only ask for what they want. Most of the features provided by Satpy can be configured by keyword arguments (see the *API Documentation* or other specific section for more details). For more complex customizations or added features Satpy uses a set of configuration files that can be modified by the user. The various components and concepts of Satpy are described below. The *Quickstart* guide also provides simple example code for the available features of Satpy.

2.1.1 Scene

Satpy provides most of its functionality through the `Scene` class. This acts as a container for the datasets being operated on and provides methods for acting on those datasets. It attempts to reduce the amount of low-level knowledge needed by the user while still providing a pythonic interface to the functionality underneath.

A Scene object represents a single geographic region of data, typically at a single continuous time range. It is possible to combine Scenes to form a Scene with multiple regions or multiple time observations, but it is not guaranteed that all functionality works in these situations.

2.1.2 DataArrays

Satpy's lower-level container for data is the `xarray.DataArray`. For historical reasons DataArrays are often referred to as "Datasets" in Satpy. These objects act similar to normal numpy arrays, but add additional metadata and attributes for describing the data. Metadata is stored in a `.attrs` dictionary and named dimensions can be accessed in a `.dims` attribute, along with other attributes. In most use cases these objects can be operated on like normal NumPy arrays with special care taken to make sure the metadata dictionary contains expected values. See the XArray documentation for more info on handling `xarray.DataArray` objects.

Additionally, Satpy uses a special form of DataArrays where data is stored in `dask.array.Array` objects which allows Satpy to perform multi-threaded lazy operations vastly improving the performance of processing. For help on developing with dask and xarray see *Migrating to xarray and dask* or the documentation for the specific project.

To uniquely identify `DataArray` objects Satpy uses *DataID*. A `DataID` consists of various pieces of available metadata. This usually includes *name* and *wavelength* as identifying metadata, but can also include *resolution*, *calibration*, *polarization*, and additional *modifiers* to further distinguish one dataset from another. For more information on *DataID* objects, have a look a *Satpy internal workings: having a look under the hood*.

Warning: XArray includes other object types called "Datasets". These are different from the "Datasets" mentioned in Satpy.

Data chunks

The usage of dask as the foundation for Satpy's operation means that the underlying data is chunked, that is, cut in smaller pieces that can then be processed in parallel. Information on dask's chunking can be found in the dask documentation here: https://docs.dask.org/en/stable/array-chunks.html The size of these chunks can have a significant impact on the performance of satpy, so to achieve best performance it can be necessary to adjust it.

Default chunk size used by Satpy can be configured by using the following around your code:

```
with dask.config.set("array.chunk-size": "32MiB"):
  # your code here
```

Or by using:

```
dask.config.set("array.chunk-size": "32MiB")
```

at the top of your code.

There are other ways to set dask configuration items, including configuration files or environment variables, see here: https://docs.dask.org/en/stable/configuration.html

The value of the chunk-size can be given in different ways, see here: https://docs.dask.org/en/stable/api.html#dask.utils.parse_bytes

The default value for this parameter is 128MiB, which can translate to chunk sizes of 4096x4096 for 64-bit float arrays.

Note however that some reader might choose to use a liberal interpretation of the chunk size which will not necessarily result in a square chunk, or even to a chunk size of the exact requested size. The motivation behind this is that data stored as stripes may load much faster if the horizontal striping is kept as much as possible instead of cutting the data in square chunks. However, the Satpy readers should respect the overall chunk size when it makes sense.

Note: The legacy way of providing the chunks size in Satpy is the `PYTROLL_CHUNK_SIZE` environment variable. This is now pending deprecation, so an equivalent way to achieve the same result is by using the `DASK_ARRAY__CHUNK_SIZE` environment variable. The value to assign to the variable is the square of the legacy variable, multiplied by the size of array data type at hand, so for example, for 64-bits floats:

```
export DASK_ARRAY__CHUNK_SIZE=134217728
```

which is the same as:

```
export DASK_ARRAY__CHUNK_SIZE="128MiB"
```

is equivalent to the deprecated:

```
export PYTROLL_CHUNK_SIZE=4096
```

2.1.3 Reading

One of the biggest advantages of using Satpy is the large number of input file formats that it can read. It encapsulates this functionality into individual *Reading*. Satpy Readers handle all of the complexity of reading whatever format they represent. Meteorological Satellite file formats can be extremely complex and formats are rarely reused across satellites or instruments. No matter the format, Satpy's Reader interface is meant to provide a consistent data loading interface while still providing flexibility to add new complex file formats.

2.1.4 Compositing

Many users of satellite imagery combine multiple sensor channels to bring out certain features of the data. This includes using one dataset to enhance another, combining 3 or more datasets in to an RGB image, or any other combination of datasets. Satpy comes with a lot of common composite combinations built-in and allows the user to request them like any other dataset. Satpy also makes it possible to create your own custom composites and have Satpy treat them like any other dataset. See *Composites* for more information.

2.1.5 Resampling

Satellite imagery data comes in two forms when it comes to geolocation, native satellite swath coordinates and uniform gridded projection coordinates. It is also common to see the channels from a single sensor in multiple resolutions, making it complicated to combine or compare the datasets. Many use cases of satellite data require the data to be in a certain projection other than the native projection or to have output imagery cover a specific area of interest. Satpy makes it easy to resample datasets to allow for users to combine them or grid them to these projections or areas of interest. Satpy uses the PyTroll *pyresample* package to provide nearest neighbor, bilinear, or elliptical weighted averaging resampling methods. See *Resampling* for more information.

2.1.6 Enhancements

When making images from satellite data the data has to be manipulated to be compatible with the output image format and still look good to the human eye. Satpy calls this functionality "enhancing" the data, also commonly called scaling or stretching the data. This process can become complicated not just because of how subjective the quality of an image can be, but also because of historical expectations of forecasters and other users for how the data should look. Satpy tries to hide the complexity of all the possible enhancement methods from the user and just provide the best looking image by default. Satpy still makes it possible to customize these procedures, but in most cases it shouldn't be necessary. See the documentation on *Writing* for more information on what's possible for output formats and enhancing images.

2.1.7 Writing

Satpy is designed to make data loading, manipulating, and analysis easy. However, the best way to get satellite imagery data out to as many users as possible is to make it easy to save it in multiple formats. Satpy allows users to save data in image formats like PNG or GeoTIFF as well as data file formats like NetCDF. Each format's complexity is hidden behind the interface of individual Writer objects and includes keyword arguments for accessing specific format features like compression and output data type. See the *Writing* documentation for the available writers and how to use them.

2.2 Installation Instructions

Satpy is available from conda-forge (via conda), PyPI (via pip), or from source (via pip+git). The below instructions show how to install stable versions of Satpy. For a development/unstable version see *Development installation*.

2.2.1 Conda-based Installation

Satpy can be installed into a conda environment by installing the package from the conda-forge channel. If you do not already have access to a conda installation, we recommend installing miniconda for the smallest and easiest installation.

The commands below will use `-c conda-forge` to make sure packages are downloaded from the conda-forge channel. Alternatively, you can tell conda to always use conda-forge by running:

```
$ conda config --add channels conda-forge
```

In a new conda environment

We recommend creating a separate environment for your work with Satpy. To create a new environment and install Satpy all in one command you can run:

```
$ conda create -c conda-forge -n my_satpy_env python satpy
```

You must then activate the environment so any future python or conda commands will use this environment.

```
$ conda activate my_satpy_env
```

This method of creating an environment with Satpy (and optionally other packages) installed can generally be created faster than creating an environment and then later installing Satpy and other packages (see the section below).

In an existing environment

Note: It is recommended that when first exploring Satpy, you create a new environment specifically for this rather than modifying one used for other work.

If you already have a conda environment, it is activated, and would like to install Satpy into it, run the following:

```
$ conda install -c conda-forge satpy
```

Note: Satpy only automatically installs the dependencies needed to process the most common use cases. Additional dependencies may need to be installed with conda or pip if import errors are encountered. To check your installation use the `check_satpy` function discussed *here*.

2.2.2 Pip-based Installation

Satpy is available from the Python Packaging Index (PyPI). A sandbox environment for *satpy* can be created using Virtualenv.

To install the *satpy* package and the minimum amount of python dependencies:

```
$ pip install satpy
```

Additional dependencies can be installed as "extras" and are grouped by reader, writer, or feature added. Extras available can be found in the setup.py file. They can be installed individually:

```
$ pip install "satpy[viirs_sdr]"
```

Or all at once, although this isn't recommended due to the large number of dependencies:

```
$ pip install "satpy[all]"
```

2.2.3 Ubuntu System Python Installation

To install Satpy on an Ubuntu system we recommend using virtual environments to separate Satpy and its dependencies from the rest of the system. Note that these instructions require using "sudo" privileges which may not be available to all users and can be very dangerous. The following instructions attempt to install some Satpy dependencies using the Ubuntu *apt* package manager to ease installation. Replace */path/to/pytroll-env* with the environment to be created.

```
$ sudo apt-get install python-pip python-gdal
$ sudo pip install virtualenv
$ virtualenv /path/to/pytroll-env
$ source /path/to/pytroll-env/bin/activate
$ pip install satpy
```

2.3 Configuration

Satpy has two levels of configuration that allow to control how Satpy and its various components behave. There are a series of "settings" that change the global Satpy behavior. There are also a series of "component configuration" YAML files for controlling the complex functionality in readers, compositors, writers, and other Satpy components that can't be controlled with traditional keyword arguments.

2.3.1 Settings

There are configuration parameters in Satpy that are not specific to one component and control more global behavior of Satpy. These parameters can be set in one of three ways:

1. Environment variable
2. YAML file
3. At runtime with `satpy.config`

This functionality is provided by the donfig library. The currently available settings are described below. Each option is available from all three methods. If specified as an environment variable or specified in the YAML file on disk, it must be set **before** Satpy is imported.

YAML Configuration

YAML files that include these parameters can be in any of the following locations:

1. `<python environment prefix>/etc/satpy/satpy.yaml`
2. `<user_config_dir>/satpy.yaml` (see below)
3. `~/.satpy/satpy.yaml`
4. `<SATPY_CONFIG_PATH>/satpy.yaml` (see *Component Configuration Path* below)

The above `user_config_dir` is provided by the `appdirs` package and differs by operating system. Typical user config directories are:

- Mac OSX: `~/Library/Preferences/satpy`
- Unix/Linux: `~/.config/satpy`
- Windows: `C:\\Users\\<username>\\AppData\\Local\\pytroll\\satpy`

All YAML files found from the above paths will be merged into one configuration object (accessed via `satpy.config`). The YAML contents should be a simple mapping of configuration key to its value. For example:

```
cache_dir: "/tmp"
data_dir: "/tmp"
```

Lastly, it is possible to specify an additional config path to the above options by setting the environment variable `SATPY_CONFIG`. The file specified with this environment variable will be added last after all of the above paths have been merged together.

At runtime

After import, the values can be customized at runtime by doing:

```
import satpy
satpy.config.set(cache_dir="/my/new/cache/path")
# ... normal satpy code ...
```

Or for specific blocks of code:

```
import satpy
with satpy.config.set(cache_dir="/my/new/cache/path"):
    # ... some satpy code ...
# ... code using the original cache_dir
```

Similarly, if you need to access one of the values you can use the `satpy.config.get` method.

Cache Directory

- **Environment variable**: `SATPY_CACHE_DIR`
- **YAML/Config Key**: `cache_dir`
- **Default**: See below

Directory where any files cached by Satpy will be stored. This directory is not necessarily cleared out by Satpy, but is rarely used without explicitly being enabled by the user. This defaults to a different path depending on your operating system following the appdirs "user cache dir".

Cache Longitudes and Latitudes

- **Environment variable**: `SATPY_CACHE_LONLATS`
- **YAML/Config Key**: `cache_lonlats`
- **Default**: `False`

Whether or not generated longitude and latitude coordinates should be cached to on-disk zarr arrays. Currently this only works in very specific cases. Mainly the lon/lats that are generated when computing sensor and solar zenith and azimuth angles used in various modifiers and compositors. This caching is only done for `AreaDefinition`-based geolocation, not `SwathDefinition`. Arrays are stored in `cache_dir` (see above).

When setting this as an environment variable, this should be set with the string equivalent of the Python boolean values `="True"` or `="False"`.

See also `cache_sensor_angles` below.

> **Warning:** This caching does not limit the number of entries nor does it expire old entries. It is up to the user to manage the contents of the cache directory.

Cache Sensor Angles

- **Environment variable**: `SATPY_CACHE_SENSOR_ANGLES`
- **YAML/Config Key**: `cache_sensor_angles`
- **Default**: `False`

Whether or not generated sensor azimuth and sensor zenith angles should be cached to on-disk zarr arrays. These angles are primarily used in certain modifiers and compositors. This caching is only done for `AreaDefinition`-based geolocation, not `SwathDefinition`. Arrays are stored in `cache_dir` (see above).

This caching requires producing an estimate of the angles to avoid needing to generate new angles for every new data case. This happens because the angle generation depends on the observation time of the data and the position of the satellite (longitude, latitude, altitude). The angles are estimated by using a constant observation time for all cases (maximum ~1e-10 error) and by rounding satellite position coordinates to the nearest tenth of a degree for longitude and latitude and nearest tenth meter (maximum ~0.058 error). Note these estimations are only done if caching is enabled (this parameter is True).

When setting this as an environment variable, this should be set with the string equivalent of the Python boolean values `="True"` or `="False"`.

See also `cache_lonlats` above.

> **Warning:** This caching does not limit the number of entries nor does it expire old entries. It is up to the user to manage the contents of the cache directory.

Component Configuration Path

- **Environment variable**: `SATPY_CONFIG_PATH`
- **YAML/Config Key**: `config_path`
- **Default**: `[]`

Base directory, or directories, where Satpy component YAML configuration files are stored. Satpy expects configuration files for specific component types to be in appropriate subdirectories (ex. `readers`, `writers`, etc), but these subdirectories should not be included in the `config_path`. For example, if you have custom composites configured in `/my/config/dir/etc/composites/visir.yaml`, then `config_path` should include `/my/config/dir/etc` for Satpy to find this configuration file when searching for composites. This option replaces the legacy `PPP_CONFIG_DIR` environment variable.

Note that this value must be a list. In Python, this could be set by doing:

```
satpy.config.set(config_path=['/path/custom1', '/path/custom2'])
```

If setting an environment variable then it must be a colon-separated (`:`) string on Linux/OSX or semicolon-separate (`;`) separated string and must be set **before** calling/importing Satpy. If the environment variable is a single path it will be converted to a list when Satpy is imported.

```
export SATPY_CONFIG_PATH="/path/custom1:/path/custom2"
```

On Windows, with paths on the *C:* drive, these paths would be:

```
set SATPY_CONFIG_PATH="C:/path/custom1;C:/path/custom2"
```

Satpy will always include the builtin configuration files that it is distributed with regardless of this setting. When a component supports merging of configuration files, they are merged in reverse order. This means "base" configuration paths should be at the end of the list and custom/user paths should be at the beginning of the list.

Data Directory

- **Environment variable**: `SATPY_DATA_DIR`
- **YAML/Config Key**: `data_dir`
- **Default**: See below

Directory where any data Satpy needs to perform certain operations will be stored. This replaces the legacy `SATPY_ANCPATH` environment variable. This defaults to a different path depending on your operating system following the appdirs "user data dir".

Demo Data Directory

- **Environment variable**: `SATPY_DEMO_DATA_DIR`
- **YAML/Config Key**: `demo_data_dir`
- **Default**: <current working directory>

Directory where demo data functions will download data files to. Available demo data functions can be found in *satpy.demo* subpackage.

Download Auxiliary Data

- **Environment variable**: `SATPY_DOWNLOAD_AUX`
- **YAML/Config Key**: `download_aux`
- **Default**: True

Whether to allow downloading of auxiliary files for certain Satpy operations. See *Auxiliary Data Download* for more information. If `True` then Satpy will download and cache any necessary data files to *Data Directory* when needed. If `False` then pre-downloaded files will be used, but any other files will not be downloaded or checked for validity.

Sensor Angles Position Preference

- **Environment variable**: `SATPY_SENSOR_ANGLES_POSITION_PREFERENCE`
- **YAML/Config Key**: `sensor_angles_position_preference`
- **Default**: "actual"

Control which satellite position should be preferred when generating sensor azimuth and sensor zenith angles. This value is passed directly to the `get_satpos()` function. See the documentation for that function for more information on how the value will be used. This is used as part of the `get_angles()` and `get_satellite_zenith_angle()` functions which is used by multiple modifiers and composites including the default rayleigh correction.

Clipping Negative Infrared Radiances

- **Environment variable**: `SATPY_READERS__CLIP_NEGATIVE_RADIANCES`
- **YAML/Config Key**: `readers.clip_negative_radiances`
- **Default**: False

Whether to clip negative infrared radiances to the minimum allowable value before computing the brightness temperature. If `clip_negative_radiances=False`, pixels with negative radiances will have `np.nan` brightness temperatures.

Clipping of negative radiances is currently implemented for the following readers:

- `abi_l1b`

Temporary Directory

- **Environment variable**: `SATPY_TMP_DIR`
- **YAML/Config Key**: `tmp_dir`
- **Default**: tempfile.gettempdir()

Directory where Satpy creates temporary files, for example decompressed input files. Default depends on the operating system.

2.3.2 Component Configuration

Much of the functionality of Satpy comes from the various components it uses, like readers, writers, compositors, and enhancements. These components are configured for reuse from YAML files stored inside Satpy or in custom user configuration files. Custom directories can be provided by specifying the *config_path setting* mentioned above.

To create and use your own custom component configuration you should:

1. Create a directory to store your new custom YAML configuration files. The files for each component will go in a subdirectory specific to that component (ex. `composites`, `enhancements`, `readers`, `writers`).
2. Set the Satpy *config_path* to point to your new directory. This could be done by setting the environment variable `SATPY_CONFIG_PATH` to your custom directory (don't include the component sub-directory) or one of the other methods for setting this path.
3. Create YAML configuration files with your custom YAML files. In most cases there is no need to copy configuration from the builtin Satpy files as these will be merged with your custom files.
4. If your custom configuration uses custom Python code, this code must be importable by Python. This means your code must either be installed in your Python environment or you must set your `PYTHONPATH` to the location of the modules.
5. Run your Satpy code and access your custom components like any of the builtin components.

2.4 Downloading Data

One of the main features of Satpy is its ability to read various satellite data formats. However, it currently only provides limited methods for downloading data from remote sources and these methods are limited to demo data for Pytroll examples. See the examples and the `demo` API documentation for details. Otherwise, Satpy assumes all data is available through the local system, either as a local directory or network mounted file systems. Certain readers that use `xarray` to open data files may be able to load files from remote systems by using OpenDAP or similar protocols.

As a user there are two options for getting access to data:

1. Download data to your local machine.
2. Connect to a remote system that already has access to data.

The most common case of a remote system having access to data is with a cloud computing service like Google Cloud Platform (GCP) or Amazon Web Services (AWS). Another possible case is an organization having direct broadcast antennas where they receive data directly from the satellite or satellite mission organization (NOAA, NASA, EUMETSAT, etc). In these cases data is usually available as a mounted network file system and can be accessed like a normal local path (with the added latency of network communications).

Below are some data sources that provide data that can be read by Satpy. If you know of others please let us know by either creating a GitHub issue or pull request.

2.4.1 NOAA GOES on Amazon Web Services

- Resource Description
- Data Browser
- Associated Readers: `abi_l1b`

In addition to the pages above, Brian Blaylock's GOES-2-Go python package is useful for downloading GOES data to your local machine. Brian also prepared some instructions for using the `rclone` tool for downloading AWS data to a local machine. The instructions can be found here.

2.4.2 NOAA GOES on Google Cloud Platform

GOES-16

- Resource Description
- Data Browser
- Associated Readers: `abi_l1b`

GOES-17

- Resource Description
- Data Browser
- Associated Readers: `abi_l1b`

2.4.3 NOAA CLASS

- Data Ordering
- Associated Readers: `viirs_sdr`

2.4.4 NASA VIIRS Atmosphere SIPS

- Resource Description
- Associated Readers: `viirs_l1b`

2.4.5 EUMETSAT Data Center

- Data Ordering

2.5 Examples

Satpy examples are available as Jupyter Notebooks on the pytroll-examples git repository. Some examples are described in further detail as separate pages in this documentation. They include python code, PNG images, and descriptions of what the example is doing. Below is a list of some of the examples and a brief summary. Additional examples can be found at the repository mentioned above or as explanations in the various sections of this documentation.

2.5.1 MTG FCI - Natural Color Example

Satpy includes a reader for the Meteosat Third Generation (MTG) FCI Level 1c data. The following Python code snippet shows an example on how to use Satpy to generate a Natural Color RGB composite over the European area.

> **Warning:** This example is currently a work in progress. Some of the below code may not work with the currently released version of Satpy. Additional updates to this example will be coming soon.

Note: For reading compressed data, a decompression library is needed. Either install the FCIDECOMP library (see the FCI L1 Product User Guide, or the `hdf5plugin` package with:

```
pip install hdf5plugin
```

or:

```
conda install hdf5plugin -c conda-forge
```

If you use `hdf5plugin`, make sure to add the line `import hdf5plugin` at the top of your script.

```
from satpy.scene import Scene
from satpy import find_files_and_readers

# define path to FCI test data folder
path_to_data = 'your/path/to/FCI/data/folder/'

# find files and assign the FCI reader
files = find_files_and_readers(base_dir=path_to_data, reader='fci_l1c_nc')

# create an FCI scene from the selected files
scn = Scene(filenames=files)

# print available dataset names for this scene (e.g. 'vis_04', 'vis_05','ir_38',...)
print(scn.available_dataset_names())

# print available composite names for this scene (e.g. 'natural_color', 'airmass',
↪'convection',...)
print(scn.available_composite_names())

# load the datasets/composites of interest
scn.load(['natural_color','vis_04'], upper_right_corner='NE')
# note: the data inside the FCI files is stored upside down. The upper_right_corner='NE'␣
↪argument
# flips it automatically in upright position.

# you can access the values of a dataset as a Numpy array with
vis_04_values = scn['vis_04'].values

# resample the scene to a specified area (e.g. "eurol1" for Europe in 1km resolution)
scn_resampled = scn.resample("eurol", resampler='nearest', radius_of_influence=5000)

# save the resampled dataset/composite to disk
scn_resampled.save_dataset("natural_color", filename='./fci_natural_color_resampled.png')
```

2.5.2 EPS-SG VII netCDF Example

Satpy includes a reader for the EPS-SG Visible and Infrared Imager (VII) Level 1b data. The following Python code snippet shows an example on how to use Satpy to read a channel and resample and save the image over the European area.

> **Warning:** This example is currently a work in progress. Some of the below code may not work with the currently released version of Satpy. Additional updates to this example will be coming soon.

```
import glob
from satpy.scene import Scene

# find the file/files to be read
filenames = glob.glob('/path/to/VII/data/W_xx-eumetsat-darmstadt,SAT,SGA1-VII-1B-RAD_C_
↪EUMT_20191007055100*')

# create a VII scene from the selected granule(s)
scn = Scene(filenames=filenames, reader='vii_l1b_nc')

# print available dataset names for this scene
print(scn.available_dataset_names())

# load the datasets of interest
# NOTE: only radiances are supported for test data
scn.load(["vii_668"], calibration="radiance")

# resample the scene to a specified area (e.g. "eurol1" for Europe in 1km resolution)
eur = scn.resample("eurol", resampler='nearest', radius_of_influence=5000)

# save the resampled data to disk
eur.save_dataset("vii_668", filename='./vii_668_eur.png')
```

Name	Description
Quickstart with MSG data	Satpy quickstart for loading and processing satellite data, with MSG data in this examples
Cartopy Plot	Plot a single VIIRS SDR granule using Cartopy and matplotlib
Himawari-8 AHI True Color	Generate and resample a rayleigh corrected true color RGB from Himawari-8 AHI data
Sentinel-3 OLCI True Color	Reading OLCI data from Sentinel 3 with Pytroll/Satpy
Sentinel 2 MSI true color	Reading MSI data from Sentinel 2 with Pytroll/Satpy
Suomi-NPP VIIRS SDR True Color	Generate a rayleigh corrected true color RGB from VIIRS I- and M-bands
Aqua/Terra MODIS True Color	Generate and resample a rayleigh corrected true color RGB from MODIS
Sentinel 1 SAR-C False Color	Generate a false color composite RGB from SAR-C polarized datasets
Level 2 EARS-NWC cloud products	Reading Level 2 EARS-NWC cloud products
Level 2 MAIA cloud products	Reading Level 2 MAIA cloud products
Meteosat Third Generation FCI Natural Color RGB	Generate Natural Color RGB from Meteosat Third Generation (MTG) FCI Level 1c data
Reading EPS-SG Visible and Infrared Imager (VII) with Pytroll	Read and visualize EPS-SG VII L1B test data and save it to an image

2.6 Quickstart

2.6.1 Loading and accessing data

To work with weather satellite data you must create a `Scene` object. Satpy does not currently provide an interface to download satellite data, it assumes that the data is on a local hard disk already. In order for Satpy to get access to the data the Scene must be told what files to read and what *Satpy Reader* should read them:

```
>>> from satpy import Scene
>>> from glob import glob
>>> filenames = glob("/home/a001673/data/satellite/Meteosat-10/seviri/lvl1.5/2015/04/20/
↪HRIT/*201504201000*")
>>> global_scene = Scene(reader="seviri_l1b_hrit", filenames=filenames)
```

To load data from the files use the `Scene.load` method. Printing the Scene object will list each of the `xarray.DataArray` objects currently loaded:

```
>>> global_scene.load(['0.8', '1.6', '10.8'])
>>> print(global_scene)
<xarray.DataArray 'reshape-d66223a8e05819b890c4535bc7e74356' (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
  * x        (x) float64 5.567e+06 5.564e+06 5.561e+06 5.558e+06 5.555e+06 ...
  * y        (y) float64 -5.567e+06 -5.564e+06 -5.561e+06 -5.558e+06 ...
Attributes:
    orbital_parameters:   {'projection_longitude': 0.0, 'pr...
    sensor:               seviri
    platform_name:        Meteosat-11
    standard_name:        brightness_temperature
```

(continues on next page)

(continued from previous page)

```
    units:                K
    wavelength:           (9.8, 10.8, 11.8)
    start_time:           2018-02-28 15:00:10.814000
    end_time:             2018-02-28 15:12:43.956000
    area:                 Area ID: some_area_name\nDescription: On-the-fly ar...
    name:                 IR_108
    resolution:           3000.40316582
    calibration:          brightness_temperature
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
<xarray.DataArray 'reshape-1982d32298aca15acb42c481fd74a629' (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
  * x        (x) float64 5.567e+06 5.564e+06 5.561e+06 5.558e+06 5.555e+06 ...
  * y        (y) float64 -5.567e+06 -5.564e+06 -5.561e+06 -5.558e+06 ...
Attributes:
    orbital_parameters:   {'projection_longitude': 0.0, 'pr...
    sensor:               seviri
    platform_name:        Meteosat-11
    standard_name:        toa_bidirectional_reflectance
    units:                %
    wavelength:           (0.74, 0.81, 0.88)
    start_time:           2018-02-28 15:00:10.814000
    end_time:             2018-02-28 15:12:43.956000
    area:                 Area ID: some_area_name\nDescription: On-the-fly ar...
    name:                 VIS008
    resolution:           3000.40316582
    calibration:          reflectance
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
<xarray.DataArray 'reshape-e86d03c30ce754995ff9da484c0dc338' (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
  * x        (x) float64 5.567e+06 5.564e+06 5.561e+06 5.558e+06 5.555e+06 ...
  * y        (y) float64 -5.567e+06 -5.564e+06 -5.561e+06 -5.558e+06 ...
Attributes:
    orbital_parameters:   {'projection_longitude': 0.0, 'pr...
    sensor:               seviri
    platform_name:        Meteosat-11
    standard_name:        toa_bidirectional_reflectance
    units:                %
    wavelength:           (1.5, 1.64, 1.78)
    start_time:           2018-02-28 15:00:10.814000
    end_time:             2018-02-28 15:12:43.956000
    area:                 Area ID: some_area_name\nDescription: On-the-fly ar...
    name:                 VIS006
    resolution:           3000.40316582
    calibration:          reflectance
```

(continues on next page)

(continued from previous page)

```
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
```

Satpy allows loading file data by wavelengths in micrometers (shown above) or by channel name:

```
>>> global_scene.load(["VIS008", "IR_016", "IR_108"])
```

To have a look at the available channels for loading from your *Scene* object use the *available_dataset_names()* method:

```
>>> global_scene.available_dataset_names()
['HRV',
 'IR_108',
 'IR_120',
 'VIS006',
 'WV_062',
 'IR_039',
 'IR_134',
 'IR_097',
 'IR_087',
 'VIS008',
 'IR_016',
 'WV_073']
```

To access the loaded data use the wavelength or name:

```
>>> print(global_scene[0.8])
```

For more information on loading datasets by resolution, calibration, or other advanced loading methods see the *Reading* documentation.

2.6.2 Calculating measurement values and navigation coordinates

Once loaded, measurement values can be calculated from a DataArray within a scene, using .values to get a fully calculated numpy array:

```
>>> vis008 = global_scene["VIS008"]
>>> vis008_meas = vis008.values
```

Note that for very large images, such as half-kilometer geostationary imagery, calculated measurement arrays may require multiple gigabytes of memory; using deferred computation and/or subsetting of datasets may be preferred in such cases.

The 'area' attribute of the DataArray, if present, can be converted to latitude and longitude arrays. For some instruments (typically polar-orbiters), the get_lonlats() may result in arrays needing an additional .compute() or .values extraction.

```
>>> vis008_lon, vis008_lat = vis008.attrs['area'].get_lonlats()
```

2.6.3 Visualizing data

To visualize loaded data in a pop-up window:

```
>>> global_scene.show(0.8)
```

Alternatively if working in a Jupyter notebook the scene can be converted to a geoviews object using the *to_geoviews()* method. The geoviews package is not a requirement of the base satpy install so in order to use this feature the user needs to install the geoviews package himself.

```
>>> import holoviews as hv
>>> import geoviews as gv
>>> import geoviews.feature as gf
>>> gv.extension("bokeh", "matplotlib")
>>> %opts QuadMesh Image [width=600 height=400 colorbar=True] Feature [apply_
↪ranges=False]
>>> %opts Image QuadMesh (cmap='RdBu_r')
>>> gview = global_scene.to_geoviews(vdims=[0.6])
>>> gview[::5,::5] * gf.coastline * gf.borders
```

2.6.4 Creating new datasets

Calculations based on loaded datasets/channels can easily be assigned to a new dataset:

```
>>> global_scene.load(['VIS006', 'VIS008'])
>>> global_scene["ndvi"] = (global_scene['VIS008'] - global_scene['VIS006']) / (global_
↪scene['VIS008'] + global_scene['VIS006'])
>>> global_scene.show("ndvi")
```

When doing calculations Xarray, by default, will drop all attributes so attributes need to be copied over by hand. The `combine_metadata()` function can assist with this task. Assigning additional custom metadata is also possible.

```
>>> from satpy.dataset import combine_metadata
>>> scene['new_band'] = scene['VIS008'] / scene['VIS006']
>>> scene['new_band'].attrs = combine_metadata(scene['VIS008'], scene['VIS006'])
>>> scene['new_band'].attrs['some_other_key'] = 'whatever_value_you_want'
```

2.6.5 Generating composites

Satpy comes with many composite recipes built-in and makes them loadable like any other dataset:

```
>>> global_scene.load(['overview'])
```

To get a list of all available composites for the current scene:

```
>>> global_scene.available_composite_names()
['overview_sun',
 'airmass',
 'natural_color',
 'night_fog',
 'overview',
 'green_snow',
```

(continues on next page)

(continued from previous page)

```
 'dust',
 'fog',
 'natural_color_raw',
 'cloudtop',
 'convection',
 'ash']
```

Loading composites will load all necessary dependencies to make that composite and unload them after the composite has been generated.

Note: Some composite require datasets to be at the same resolution or shape. When this is the case the Scene object must be resampled before the composite can be generated (see below).

2.6.6 Resampling

In certain cases it may be necessary to resample datasets whether they come from a file or are generated composites. Resampling is useful for mapping data to a uniform grid, limiting input data to an area of interest, changing from one projection to another, or for preparing datasets to be combined in a composite (see above). For more details on resampling, different resampling algorithms, and creating your own area of interest see the *Resampling* documentation. To resample a Satpy Scene:

```
>>> local_scene = global_scene.resample("eurol")
```

This creates a copy of the original `global_scene` with all loaded datasets resampled to the built-in "eurol" area. Any composites that were requested, but could not be generated are automatically generated after resampling. The new `local_scene` can now be used like the original `global_scene` for working with datasets, saving them to disk or showing them on screen:

```
>>> local_scene.show('overview')
>>> local_scene.save_dataset('overview', './local_overview.tif')
```

2.6.7 Saving to disk

To save all loaded datasets to disk as geotiff images:

```
>>> global_scene.save_datasets()
```

To save all loaded datasets to disk as PNG images:

```
>>> global_scene.save_datasets(writer='simple_image')
```

Or to save an individual dataset:

```
>>> global_scene.save_dataset('VIS006', 'my_nice_image.png')
```

Datasets are automatically scaled or "enhanced" to be compatible with the output format and to provide the best looking image. For more information on saving datasets and customizing enhancements see the documentation on *Writing*.

2.6.8 Slicing and subsetting scenes

Array slicing can be done at the scene level in order to get subsets with consistent navigation throughout. Note that this does not take into account scenes that may include channels at multiple resolutions, i.e. index slicing does not account for dataset spatial resolution.

```
>>> scene_slice = global_scene[2000:2004, 2000:2004]
>>> vis006_slice = scene_slice['VIS006']
>>> vis006_slice_meas = vis006_slice.values
>>> vis006_slice_lon, vis006_slice_lat = vis006_slice.attrs['area'].get_lonlats()
```

To subset multi-resolution data consistently, use the *crop()* method.

```
>>> scene_llbox = global_scene.crop(ll_bbox=(-4.0, -3.9, 3.9, 4.0))
>>> vis006_llbox = scene_llbox['VIS006']
>>> vis006_llbox_meas = vis006_llbox.values
>>> vis006_llbox_lon, vis006_llbox_lat = vis006_llbox.attrs['area'].get_lonlats()
```

2.6.9 Troubleshooting

When something goes wrong, a first step to take is check that the latest Version of satpy and its dependencies are installed. Satpy drags in a few packages as dependencies per default, but each reader and writer has it's own dependencies which can be unfortunately easy to miss when just doing a regular *pip install*. To check the missing dependencies for the readers and writers, a utility function called *check_satpy()* can be used:

```
>>> from satpy.utils import check_satpy
>>> check_satpy()
```

Due to the way Satpy works, producing as many datasets as possible, there are times that behavior can be unexpected but with no exceptions raised. To help troubleshoot these situations log messages can be turned on. To do this run the following code before running any other Satpy code:

```
>>> from satpy.utils import debug_on
>>> debug_on()
```

2.7 Reading

Satpy supports reading and loading data from many input file formats and schemes through the concept of *readers*. Each reader supports a specific type of input data. The *Scene* object provides a simple interface around all the complexity of these various formats through its `load` method. The following sections describe the different way data can be loaded, requested, or added to a Scene object.

2.7.1 Available Readers

For readers currently available in Satpy see *Satpy Readers*. Additionally to get a list of available readers you can use the *available_readers* function. By default, it returns the names of available readers. To return additional reader information use *available_readers(as_dict=True)*:

```
>>> from satpy import available_readers
>>> available_readers()
```

2.7.2 Filter loaded files

Coming soon…

2.7.3 Load data

Datasets in Satpy are identified by certain pieces of metadata set during data loading. These include *name*, *wavelength*, *calibration*, *resolution*, *polarization*, and *modifiers*. Normally, once a `Scene` is created requesting datasets by *name* or *wavelength* is all that is needed:

```
>>> from satpy import Scene
>>> scn = Scene(reader="seviri_l1b_hrit", filenames=filenames)
>>> scn.load([0.6, 0.8, 10.8])
>>> scn.load(['IR_120', 'IR_134'])
```

However, in many cases datasets are available in multiple spatial resolutions, multiple calibrations (`brightness_temperature`, `reflectance`, `radiance`, etc), multiple polarizations, or have corrections or other modifiers already applied to them. By default Satpy will provide the version of the dataset with the highest resolution and the highest level of calibration (brightness temperature or reflectance over radiance). It is also possible to request one of these exact versions of a dataset by using the `DataQuery` class:

```
>>> from satpy import DataQuery
>>> my_channel_id = DataQuery(name='IR_016', calibration='radiance')
>>> scn.load([my_channel_id])
>>> print(scn['IR_016'])
```

Or request multiple datasets at a specific calibration, resolution, or polarization:

```
>>> scn.load([0.6, 0.8], resolution=1000)
```

Or multiple calibrations:

```
>>> scn.load([0.6, 10.8], calibration=['brightness_temperature', 'radiance'])
```

In the above case Satpy will load whatever dataset is available and matches the specified parameters. So the above `load` call would load the `0.6` (a visible/reflectance band) radiance data and `10.8` (an IR band) brightness temperature data.

For geostationary satellites that have the individual channel data separated to several files (segments) the missing segments are padded by default to full disk area. This is made to simplify caching of resampling look-up tables (see *Resampling* for more information). To disable this, the user can pass `pad_data` keyword argument when loading datasets:

```
>>> scn.load([0.6, 10.8], pad_data=False)
```

For geostationary products, where the imagery is stored in the files in an unconventional orientation (e.g. MSG SEVIRI L1.5 data are stored with the southwest corner in the upper right), the keyword argument `upper_right_corner` can be passed into the load call to automatically flip the datasets to the wished orientation. Accepted argument values are `'NE'`, `'NW'`, `'SE'`, `'SW'`, and `'native'`. By default, no flipping is applied (corresponding to `upper_right_corner='native'`) and the data are delivered in the original format. To get the data in the common upright orientation, load the datasets using e.g.:

```
>>> scn.load(['VIS008'], upper_right_corner='NE')
```

Note: If a dataset could not be loaded there is no exception raised. You must check the *scn.missing_datasets* property for any `DataID` that could not be loaded.

To find out what datasets are available from a reader from the files that were provided to the `Scene` use *available_dataset_ids()*:

```
>>> scn.available_dataset_ids()
```

Or *available_dataset_names()* for just the string names of Datasets:

```
>>> scn.available_dataset_names()
```

2.7.4 Load remote data

Starting with Satpy version 0.25.1 with supported readers it is possible to load data from remote file systems like `s3fs` or `fsspec`. For example:

```
>>> from satpy import Scene
>>> from satpy.readers import FSFile
>>> import fsspec

>>> filename = 'noaa-goes16/ABI-L1b-RadC/2019/001/17/*_G16_s20190011702186*'

>>> the_files = fsspec.open_files("simplecache::s3://" + filename, s3={'anon': True})

>>> fs_files = [FSFile(open_file) for open_file in the_files]

>>> scn = Scene(filenames=fs_files, reader='abi_l1b')
>>> scn.load(['true_color_raw'])
```

Check the list of *Satpy Readers* to see which reader supports remote files. For the usage of `fsspec` and advanced features like caching files locally see the fsspec Documentation .

2.7.5 Search for local/remote files

Satpy provides a utility *find_files_and_readers()* for searching for files in a base directory matching various search parameters. This function discovers files based on filename patterns. It returns a dictionary mapping reader name to a list of filenames supported. This dictionary can be passed directly to the *Scene* initialization.

```
>>> from satpy import find_files_and_readers, Scene
>>> from datetime import datetime
>>> my_files = find_files_and_readers(base_dir='/data/viirs_sdrs',
...                                   reader='viirs_sdr',
...                                   start_time=datetime(2017, 5, 1, 18, 1, 0),
...                                   end_time=datetime(2017, 5, 1, 18, 30, 0))
>>> scn = Scene(filenames=my_files)
```

See the *find_files_and_readers()* documentation for more information on the possible parameters as well as for searching on remote file systems.

2.7.6 Metadata

The datasets held by a scene also provide vital metadata such as dataset name, units, observation time etc. The following attributes are standardized across all readers:

- `name`, and other identifying metadata keys: See *Satpy internal workings: having a look under the hood*.
- `start_time`: Left boundary of the time interval covered by the dataset. For more information see the *Time Metadata* section below.
- `end_time`: Right boundary of the time interval covered by the dataset. For more information see the *Time Metadata* section below.
- `area`: `AreaDefinition` or `SwathDefinition` if data is geolocated. Areas are used for gridded projected data and Swaths when data must be described by individual longitude/latitude coordinates. See the Coordinates section below.
- `reader`: The name of the Satpy reader that produced the dataset.
- `orbital_parameters`: Dictionary of orbital parameters describing the satellite's position. See the *Orbital Parameters* section below for more information.
- `time_parameters`: Dictionary of additional time parameters describing the time ranges related to the requests or schedules for when observations should happen and when they actually do. See *Time Metadata* below for details.
- `raw_metadata`: Raw, unprocessed metadata from the reader.

Note that the above attributes are not necessarily available for each dataset.

Time Metadata

In addition to the generic `start_time` and `end_time` pieces of metadata there are other time fields that may be provided if the reader supports them. These items are stored in a `time_parameters` sub-dictionary and they include values like:

- `observation_start_time`: The point in time when a sensor began recording for the current data.
- `observation_end_time`: Same as `observation_start_time`, but when data has stopped being recorded.

- `nominal_start_time`: The "human friendly" time describing the start of the data observation interval or repeat cycle. This time is often on a round minute (seconds=0). Along with the nominal end time, these times define the regular interval of the data collection. For example, GOES-16 ABI full disk images are collected every 10 minutes (in the common configuration) so `nominal_start_time` and `nominal_end_time` would be 10 minutes apart regardless of when the instrument recorded data inside that interval. This time may also be referred to as the repeat cycle, repeat slot, or time slot.
- `nominal_end_time`: Same as `nominal_start_time`, but the end of the interval.

In general, `start_time` and `end_time` will be set to the "nominal" time by the reader. This ensures that other Satpy components get a consistent time for calculations (ex. generation of solar zenith angles) and can be reused between bands.

See the *Coordinates* section below for more information on time information that may show up as a per-element/row "coordinate" on the DataArray (ex. acquisition time) instead of as metadata.

Orbital Parameters

Orbital parameters describe the position of the satellite. As such they typically come in a few "flavors" for the common types of orbits a satellite may have.

For *geostationary* satellites it is described using the following scalar attributes:

- `satellite_actual_longitude/latitude/altitude`: Current position of the satellite at the time of observation in geodetic coordinates (i.e. altitude is relative and normal to the surface of the ellipsoid). The longitude and latitude are given in degrees, the altitude in meters.
- `satellite_nominal_longitude/latitude/altitude`: Center of the station keeping box (a confined area in which the satellite is actively maintained in using maneuvers). Inbetween major maneuvers, when the satellite is permanently moved, the nominal position is constant. The longitude and latitude are given in degrees, the altitude in meters.
- `nadir_longitude/latitude`: Intersection of the instrument's Nadir with the surface of the earth. May differ from the actual satellite position, if the instrument is pointing slightly off the axis (satellite, earth-center). If available, this should be used to compute viewing angles etc. Otherwise, use the actual satellite position. The values are given in degrees.
- `projection_longitude/latitude/altitude`: Projection center of the re-projected data. This should be used to compute lat/lon coordinates. Note that the projection center can differ considerably from the actual satellite position. For example MSG-1 was at times positioned at 3.4 degrees west, while the image data was re-projected to 0 degrees. The longitude and latitude are given in degrees, the altitude in meters.

> **Note:** For use in pyorbital, the altitude has to be converted to kilometers, see for example `pyorbital.orbital.get_observer_look()`.

For *polar orbiting* satellites the readers usually provide coordinates and viewing angles of the swath as ancillary datasets. Additional metadata related to the satellite position includes:

- `tle`: Two-Line Element (TLE) set used to compute the satellite's orbit

2.7.7 Coordinates

Each `DataArray` produced by Satpy has several Xarray coordinate variables added to them.

- `x` and `y`: Projection coordinates for gridded and projected data. By default *y* and *x* are the preferred **dimensions** for all 2D data, but these **coordinates** are only added for gridded (non-swath) data. For 1D data only the `y` dimension may be specified.
- `crs`: A `CRS` object defined the Coordinate Reference System for the data. Requires pyproj 2.0 or later to be installed. This is stored as a scalar array by Xarray so it must be accessed by doing `crs = my_data_arr.attrs['crs'].item()`. For swath data this defaults to a `longlat` CRS using the WGS84 datum.
- `longitude`: Array of longitude coordinates for swath data.
- `latitude`: Array of latitude coordinates for swath data.

Readers are free to define any coordinates in addition to the ones above that are automatically added. Other possible coordinates you may see:

- `acq_time`: Instrument data acquisition time per scan or row of data.

2.7.8 Adding a Reader to Satpy

This is described in the developer guide, see *Adding a Custom Reader to Satpy*.

2.7.9 Implemented readers

SEVIRI L1.5 data readers

Common functionality for SEVIRI L1.5 data readers.

Introduction

The Spinning Enhanced Visible and InfraRed Imager (SEVIRI) is the primary instrument on Meteosat Second Generation (MSG) and has the capacity to observe the Earth in 12 spectral channels.

Level 1.5 corresponds to image data that has been corrected for all unwanted radiometric and geometric effects, has been geolocated using a standardised projection, and has been calibrated and radiance-linearised. (From the EUMETSAT documentation)

Satpy provides the following readers for SEVIRI L1.5 data in different formats:

- Native: `satpy.readers.seviri_l1b_native`
- HRIT: `satpy.readers.seviri_l1b_hrit`
- netCDF: `satpy.readers.seviri_l1b_nc`

Calibration

This section describes how to control the calibration of SEVIRI L1.5 data.

Calibration to radiance

The SEVIRI L1.5 data readers allow for choosing between two file-internal calibration coefficients to convert counts to radiances:

- Nominal for all channels (default)
- GSICS where available (IR currently) and nominal for the remaining channels (VIS & HRV currently)

In order to change the default behaviour, use the `reader_kwargs` keyword argument upon Scene creation:

```
import satpy
scene = satpy.Scene(filenames=filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'calib_mode': 'GSICS'})
scene.load(['VIS006', 'IR_108'])
```

In addition, two other calibration methods are available:

1. It is possible to specify external calibration coefficients for the conversion from counts to radiances. External coefficients take precedence over internal coefficients and over the Meirink coefficients, but you can also mix internal and external coefficients: If external calibration coefficients are specified for only a subset of channels, the remaining channels will be calibrated using the chosen file-internal coefficients (nominal or GSICS). Calibration coefficients must be specified in [mW m-2 sr-1 (cm-1)-1].
2. The calibration mode `meirink-2023` uses coefficients based on an intercalibration with Aqua-MODIS for the visible channels, as found in Inter-calibration of polar imager solar channels using SEVIRI (2013) by J. F. Meirink, R. A. Roebeling, and P. Stammes.

In the following example we use external calibration coefficients for the `VIS006` & `IR_108` channels, and nominal coefficients for the remaining channels:

```
coefs = {'VIS006': {'gain': 0.0236, 'offset': -1.20},
         'IR_108': {'gain': 0.2156, 'offset': -10.4}}
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'ext_calib_coefs': coefs})
scene.load(['VIS006', 'VIS008', 'IR_108', 'IR_120'])
```

In the next example we use external calibration coefficients for the `VIS006` & `IR_108` channels, GSICS coefficients where available (other IR channels) and nominal coefficients for the rest:

```
coefs = {'VIS006': {'gain': 0.0236, 'offset': -1.20},
         'IR_108': {'gain': 0.2156, 'offset': -10.4}}
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'calib_mode': 'GSICS',
                                   'ext_calib_coefs': coefs})
scene.load(['VIS006', 'VIS008', 'IR_108', 'IR_120'])
```

In the next example we use the mode `meirink-2023` calibration coefficients for all visible channels and nominal coefficients for the rest:

```
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'calib_mode': 'meirink-2023'})
scene.load(['VIS006', 'VIS008', 'IR_016'])
```

Calibration to reflectance

When loading solar channels, the SEVIRI L1.5 data readers apply a correction for the Sun-Earth distance variation throughout the year - as recommended by the EUMETSAT document Conversion from radiances to reflectances for SEVIRI warm channels. In the unlikely situation that this correction is not required, it can be removed on a per-channel basis using *satpy.readers.utils.remove_earthsun_distance_correction()*.

Masking of bad quality scan lines

By default bad quality scan lines are masked and replaced with `np.nan` for radiance, reflectance and brightness temperature calibrations based on the quality flags provided by the data (for details on quality flags see MSG Level 1.5 Image Data Format Description page 109). To disable masking `reader_kwargs={'mask_bad_quality_scan_lines': False}` can be passed to the Scene.

Metadata

The SEVIRI L1.5 readers provide the following metadata:

- The `orbital_parameters` attribute provides the nominal and actual satellite position, as well as the projection centre. See the *Metadata* section in the *Reading* chapter for more information.
- The `acq_time` coordinate provides the mean acquisition time for each scanline. Use a `MultiIndex` to enable selection by acquisition time:

```
import pandas as pd
mi = pd.MultiIndex.from_arrays([scn['IR_108']['y'].data, scn['IR_108']['acq_time'].
↪data],
                               names=('y_coord', 'time'))
scn['IR_108']['y'] = mi
scn['IR_108'].sel(time=np.datetime64('2019-03-01T12:06:13.052000000'))
```

- Raw metadata from the file header can be included by setting the reader argument `include_raw_metadata=True` (HRIT and Native format only). Note that this comes with a performance penalty of up to 10% if raw metadata from multiple segments or scans need to be combined. By default, arrays with more than 100 elements are excluded to limit the performance penalty. This threshold can be adjusted using the `mda_max_array_size` reader keyword argument:

```
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_hrit/native',
                    reader_kwargs={'include_raw_metadata': True,
                                   'mda_max_array_size': 1000})
```

References

- MSG Level 1.5 Image Data Format Description
- Radiometric Calibration of MSG SEVIRI Level 1.5 Image Data in Equivalent Spectral Blackbody Radiance

SEVIRI HRIT format reader

SEVIRI Level 1.5 HRIT format reader.

Introduction

The `seviri_l1b_hrit` reader reads and calibrates MSG-SEVIRI L1.5 image data in HRIT format. The format is explained in the MSG Level 1.5 Image Data Format Description. The files are usually named as follows:

```
H-000-MSG4__-MSG4________-_________-PRO______-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000001___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000002___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000003___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000004___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000005___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000006___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000007___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000008___-201903011200-__
H-000-MSG4__-MSG4________-_________-EPI______-201903011200-__
```

Each image is decomposed into 24 segments (files) for the high-resolution-visible (HRV) channel and 8 segments for other visible (VIS) and infrared (IR) channels. Additionally, there is one prologue and one epilogue file for the entire scan which contain global metadata valid for all channels.

Reader Arguments

Some arguments can be provided to the reader to change its behaviour. These are provided through the *Scene* instantiation, eg:

```
scn = Scene(filenames=filenames, reader="seviri_l1b_hrit", reader_kwargs={'fill_hrv':␣
↪False})
```

To see the full list of arguments that can be provided, look into the documentation of *HRITMSGFileHandler*.

Compression

This reader accepts compressed HRIT files, ending in `C_` as other HRIT readers, see *satpy.readers.hrit_base.HRITFileHandler*.

This reader also accepts bzipped file with the extension `.bz2` for the prologue, epilogue, and segment files.

Nominal start/end time

Warning: attribute access change

`nominal_start_time` and `nominal_end_time` should be accessed using the `time_parameters` attribute.

`nominal_start_time` and `nominal_end_time` are also available directly via `start_time` and `end_time` respectively.

Here is an exmaple of the content of the start/end time and `time_parameters` attibutes

```
Start time: 2019-08-29 12:00:00
End time:   2019-08-29 12:15:00
time_parameters:
                {'nominal_start_time': datetime.datetime(2019, 8, 29, 12, 0),
                 'nominal_end_time': datetime.datetime(2019, 8, 29, 12, 15),
                 'observation_start_time': datetime.datetime(2019, 8, 29, 12, 0, 9,␣
↪338000),
                 'observation_end_time': datetime.datetime(2019, 8, 29, 12, 15, 9,␣
↪203000)
                 }
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene
import glob

filenames = glob.glob('data/H-000-MSG4__-MSG4________-*201903011200*')
scn = Scene(filenames=filenames, reader='seviri_l1b_hrit')
scn.load(['VIS006', 'IR_108'])
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
    acq_time  (y) datetime64[ns] NaT NaT NaT NaT NaT NaT ... NaT NaT NaT NaT NaT
  * x         (x) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
  * y         (y) float64 -5.566e+06 -5.563e+06 ... 5.566e+06 5.569e+06
Attributes:
    orbital_parameters:       {'projection_longitude': 0.0, 'projection_latit...
    platform_name:            Meteosat-11
    georef_offset_corrected:  True
    standard_name:            brightness_temperature
    raw_metadata:             {'file_type': 0, 'total_header_length': 6198, '...
    wavelength:               (9.8, 10.8, 11.8)
    units:                    K
    sensor:                   seviri
```

(continues on next page)

(continued from previous page)

```
    platform_name:            Meteosat-11
    start_time:               2019-03-01 12:00:09.716000
    end_time:                 2019-03-01 12:12:42.946000
    area:                     Area ID: some_area_name\\nDescription: On-the-fl...
    name:                     IR_108
    resolution:               3000.403165817
    calibration:              brightness_temperature
    polarization:             None
    level:                    None
    modifiers:                ()
    ancillary_variables:      []
```

The *filenames* argument can either be a list of strings, see the example above, or a list of `satpy.readers.FSFile` objects. FSFiles can be used in conjunction with fsspec, e.g. to handle in-memory data:

```
import glob

from fsspec.implementations.memory import MemoryFile, MemoryFileSystem
from satpy import Scene
from satpy.readers import FSFile

# In this example, we will make use of `MemoryFile`s in a `MemoryFileSystem`.
memory_fs = MemoryFileSystem()

# Usually, the data already resides in memory.
# For explanatory reasons, we will load the files found with glob in memory,
#  and load the scene with FSFiles.
filenames = glob.glob('data/H-000-MSG4__-MSG4_______-*201903011200*')
fs_files = []
for fn in filenames:
    with open(fn, 'rb') as fh:
        fs_files.append(MemoryFile(
            fs=memory_fs,
            path="{}{}".format(memory_fs.root_marker, fn),
            data=fh.read()
        ))
        fs_files[-1].commit()  # commit the file to the filesystem
fs_files = [FSFile(open_file) for open_file in filenames]  # wrap MemoryFiles as FSFiles
# similar to the example above, we pass a list of FSFiles to the `Scene`
scn = Scene(filenames=fs_files, reader='seviri_l1b_hrit')
scn.load(['VIS006', 'IR_108'])
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
    acq_time  (y) datetime64[ns] NaT NaT NaT NaT NaT NaT ... NaT NaT NaT NaT NaT
  * x         (x) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
  * y         (y) float64 -5.566e+06 -5.563e+06 ... 5.566e+06 5.569e+06
Attributes:
```

(continues on next page)

(continued from previous page)

```
    orbital_parameters:       {'projection_longitude': 0.0, 'projection_latit...
    platform_name:            Meteosat-11
    georef_offset_corrected:  True
    standard_name:            brightness_temperature
    raw_metadata:             {'file_type': 0, 'total_header_length': 6198, '...
    wavelength:               (9.8, 10.8, 11.8)
    units:                    K
    sensor:                   seviri
    platform_name:            Meteosat-11
    start_time:               2019-03-01 12:00:09.716000
    end_time:                 2019-03-01 12:12:42.946000
    area:                     Area ID: some_area_name\\nDescription: On-the-fl...
    name:                     IR_108
    resolution:               3000.403165817
    calibration:              brightness_temperature
    polarization:             None
    level:                    None
    modifiers:                ()
    ancillary_variables:      []
```

References

- EUMETSAT Product Navigator
- MSG Level 1.5 Image Data Format Description
- fsspec

SEVIRI Native format reader

SEVIRI Level 1.5 native format reader.

Introduction

The `seviri_l1b_native` reader reads and calibrates MSG-SEVIRI L1.5 image data in binary format. The format is explained in the MSG Level 1.5 Native Format File Definition. The files are usually named as follows:

```
MSG4-SEVI-MSG15-0100-NA-20210302124244.185000000Z-NA.nat
```

Reader Arguments

Some arguments can be provided to the reader to change its behaviour. These are provided through the *Scene* instantiation, eg:

```
scn = Scene(filenames=filenames, reader="seviri_l1b_native", reader_kwargs={'fill_disk':␣
↪True})
```

To see the full list of arguments that can be provided, look into the documentation of *NativeMSGFileHandler*.

Example:

Here is an example how to read the data in satpy.

NOTE: When loading the data, the orientation of the image can be set with `upper_right_corner`-keyword. Possible options are `NW`, `NE`, `SW`, `SE`, or `native`.

```
from satpy import Scene

filenames = ['MSG4-SEVI-MSG15-0100-NA-20210302124244.185000000Z-NA.nat']
scn = Scene(filenames=filenames, reader='seviri_l1b_native')
scn.load(['VIS006', 'IR_108'], upper_right_corner='NE')
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray 'reshape-969ef97d34b7b0c70ca19f53c6abcb68' (y: 3712, x: 3712)>
dask.array<truediv, shape=(3712, 3712), dtype=float32, chunksize=(928, 3712),␣
↪chunktype=numpy.ndarray>
Coordinates:
    acq_time  (y) datetime64[ns] NaT NaT NaT NaT NaT NaT ... NaT NaT NaT NaT NaT
    crs       object PROJCRS["unknown",BASEGEOGCRS["unknown",DATUM["unknown",...
  * y         (y) float64 -5.566e+06 -5.563e+06 ... 5.566e+06 5.569e+06
  * x         (x) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
Attributes:
    orbital_parameters:       {'projection_longitude': 0.0, 'projection_latit...
    time_parameters:          {'nominal_start_time': datetime.datetime(2021, ...
    units:                    K
    wavelength:               10.8 µm (9.8-11.8 µm)
    standard_name:            toa_brightness_temperature
    platform_name:            Meteosat-11
    sensor:                   seviri
    georef_offset_corrected:  True
    start_time:               2021-03-02 12:30:11.584603
    end_time:                 2021-03-02 12:45:09.949762
    reader:                   seviri_l1b_native
    area:                     Area ID: msg_seviri_fes_3km\\nDescription: MSG S...
    name:                     IR_108
    resolution:               3000.403165817
    calibration:              brightness_temperature
    modifiers:                ()
    _satpy_id:                DataID(name='IR_108', wavelength=WavelengthRang...
    ancillary_variables:      []
```

References

- EUMETSAT Product Navigator
- MSG Level 1.5 Native Format File Definition

SEVIRI netCDF format reader

SEVIRI netcdf format reader.

Other xRIT-based readers

HRIT/LRIT format reader.

This module is the base module for all HRIT-based formats. Here, you will find the common building blocks for hrit reading.

One of the features here is the on-the-fly decompression of hrit files. It needs a path to the xRITDecompress binary to be provided through the environment variable called XRIT_DECOMPRESS_PATH. When compressed hrit files are then encountered (files finishing with *.C_*), they are decompressed to the system's temporary directory for reading.

JMA HRIT format reader

HRIT format reader for JMA data.

Introduction

The JMA HRIT format is described in the JMA HRIT - Mission Specific Implementation. There are three readers for this format in Satpy:

- `jami_hrit`: For data from the *JAMI* instrument on MTSAT-1R
- `mtsat2-imager_hrit`: For data from the *Imager* instrument on MTSAT-2
- `ahi_hrit`: For data from the *AHI* instrument on Himawari-8/9

Although the data format is identical, the instruments have different characteristics, which is why there is a dedicated reader for each of them. Sample data is available here:

- JAMI/Imager sample data
- AHI sample data

Example:

Here is an example how to read Himwari-8 HRIT data with Satpy:

```
from satpy import Scene
import glob

filenames = glob.glob('data/IMG_DK01B14_2018011109*')
scn = Scene(filenames=filenames, reader='ahi_hrit')
scn.load(['B14'])
print(scn['B14'])
```

Output:

```
<xarray.DataArray (y: 5500, x: 5500)>
dask.array<concatenate, shape=(5500, 5500), dtype=float64, chunksize=(550, 4096), ...
Coordinates:
    acq_time  (y) datetime64[ns] 2018-01-11T09:00:20.995200 ... 2018-01-11T09:09:40.
↪348800
    crs       object +proj=geos +lon_0=140.7 +h=35785831 +x_0=0 +y_0=0 +a=6378169 ...
  * y         (y) float64 5.5e+06 5.498e+06 5.496e+06 ... -5.496e+06 -5.498e+06
  * x         (x) float64 -5.498e+06 -5.496e+06 -5.494e+06 ... 5.498e+06 5.5e+06
Attributes:
    orbital_parameters:   {'projection_longitude': 140.7, 'projection_latitud...
    standard_name:        toa_brightness_temperature
    level:                None
    wavelength:           (11.0, 11.2, 11.4)
    units:                K
    calibration:          brightness_temperature
    file_type:            ['hrit_b14_seg', 'hrit_b14_fd']
    modifiers:            ()
    polarization:         None
    sensor:               ahi
    name:                 B14
    platform_name:        Himawari-8
    resolution:           4000
    start_time:           2018-01-11 09:00:20.995200
    end_time:             2018-01-11 09:09:40.348800
    area:                 Area ID: FLDK, Description: Full Disk, Projection I...
    ancillary_variables:  []
```

JMA HRIT data contain the scanline acquisition time for only a subset of scanlines. Timestamps of the remaining scanlines are computed using linear interpolation. This is what you'll find in the `acq_time` coordinate of the dataset.

Compression

Gzip-compressed MTSAT files can be decompressed on the fly using *FSFile*:

```
import fsspec
from satpy import Scene
from satpy.readers import FSFile

filename = "/data/HRIT_MTSAT1_20090101_0630_DK01IR1.gz"
open_file = fsspec.open(filename, compression="gzip")
fs_file = FSFile(open_file)
scn = Scene([fs_file], reader="jami_hrit")
scn.load(["IR1"])
```

GOES HRIT format reader

GOES HRIT format reader.

References

LRIT/HRIT Mission Specific Implementation, February 2012 GVARRDL98.pdf 05057_SPE_MSG_LRIT_HRI

Electro-L HRIT format reader

HRIT format reader.

References

ELECTRO-L GROUND SEGMENT MSU-GS INSTRUMENT,
LRIT/HRIT Mission Specific Implementation, February 2012

hdf-eos based readers

Modis level 1b hdf-eos format reader.

Introduction

The `modis_l1b` reader reads and calibrates Modis L1 image data in hdf-eos format. Files often have a pattern similar to the following one:

```
M[O/Y]D02[1/H/Q]KM.A[date].[time].[collection].[processing_time].hdf
```

Other patterns where "collection" and/or "proccessing_time" are missing might also work (see the readers yaml file for details). Geolocation files (MOD03) are also supported. The IMAPP direct broadcast naming format is also supported with names like: `a1.12226.1846.1000m.hdf`.

Saturation Handling

Band 2 of the MODIS sensor is available in 250m, 500m, and 1km resolutions. The band data may include a special fill value to indicate when the detector was saturated in the 250m version of the data. When the data is aggregated to coarser resolutions this saturation fill value is converted to a "can't aggregate" fill value. By default, Satpy will replace these fill values with NaN to indicate they are invalid. This is typically undesired when generating images for the data as they appear as "holes" in bright clouds. To control this the keyword argument `mask_saturated` can be passed and set to `False` to set these two fill values to the maximum valid value.

```
scene = satpy.Scene(filenames=filenames,
                    reader='modis_l1b',
                    reader_kwargs={'mask_saturated': False})
scene.load(['2'])
```

Note that the saturation fill value can appear in other bands (ex. bands 7-19) in addition to band 2. Also, the "can't aggregate" fill value is a generic "catch all" for any problems encountered when aggregating high resolution bands to lower resolutions. Filling this with the max valid value could replace non-saturated invalid pixels with valid values.

Geolocation files

For the 1km data (mod021km) geolocation files (mod03) are optional. If not given to the reader 1km geolocations will be interpolated from the 5km geolocation contained within the file.

For the 500m and 250m data geolocation files are needed.

References

- Modis gelocation description: http://www.icare.univ-lille1.fr/wiki/index.php/MODIS_geolocation

Modis level 2 hdf-eos format reader.

Introduction

The `modis_l2` reader reads and calibrates Modis L2 image data in hdf-eos format. Since there are a multitude of different level 2 datasets not all of theses are implemented (yet).

Currently the reader supports:

- m[o/y]d35_l2: cloud_mask dataset
- some datasets in m[o/y]d06 files

To get a list of the available datasets for a given file refer to the "Load data" section in *Reading*.

Geolocation files

Similar to the `modis_l1b` reader the geolocation files (mod03) for the 1km data are optional and if not given 1km geolocations will be interpolated from the 5km geolocation contained within the file.

For the 500m and 250m data geolocation files are needed.

References

- Documentation about the format: https://modis-atmos.gsfc.nasa.gov/products

satpy cf nc readers

Reader for files produced with the cf netcdf writer in satpy.

Introduction

The `satpy_cf_nc` reader reads data written by the satpy cf_writer. Filenames for cf_writer are optional. There are several readers using the same satpy_cf_nc.py reader.

- Generic reader `satpy_cf_nc`
- EUMETSAT GAC FDR reader `avhrr_l1c_eum_gac_fdr_nc`

Generic reader

The generic `satpy_cf_nc` reader reads files of type:

```
'{platform_name}-{sensor}-{start_time:%Y%m%d%H%M%S}-{end_time:%Y%m%d%H%M%S}.nc'
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene

filenames = ['data/npp-viirs-mband-20201007075915-20201007080744.nc']
scn = Scene(reader='satpy_cf_nc', filenames=filenames)
scn.load(['M05'])
scn['M05']
```

Output:

```
<xarray.DataArray 'M05' (y: 4592, x: 3200)>
dask.array<open_dataset-d91cfbf1bf4f14710d27446d91cdc6e4M05, shape=(4592, 3200),
    dtype=float32, chunksize=(4096, 3200), chunktype=numpy.ndarray>
Coordinates:
    longitude  (y, x) float32 dask.array<chunksize=(4096, 3200), meta=np.ndarray>
    latitude   (y, x) float32 dask.array<chunksize=(4096, 3200), meta=np.ndarray>
Dimensions without coordinates: y, x
Attributes:
    start_time:                  2020-10-07 07:59:15
    start_orbit:                 46350
    end_time:                    2020-10-07 08:07:44
    end_orbit:                   46350
    calibration:                 reflectance
    long_name:                   M05
    modifiers:                   ('sunz_corrected',)
    platform_name:               Suomi-NPP
    resolution:                  742
    sensor:                      viirs
    standard_name:               toa_bidirectional_reflectance
    units:                       %
    wavelength:                  0.672µm (0.662-0.682µm)
    date_created:                2020-10-07T08:20:02Z
    instrument:                  VIIRS
```

Notes

Available datasets and attributes will depend on the data saved with the cf_writer.

EUMETSAT AVHRR GAC FDR L1C reader

The `avhrr_l1c_eum_gac_fdr_nc` reader reads files of type:

```
''AVHRR-GAC_FDR_1C_{platform}_{start_time:%Y%m%dT%H%M%SZ}_{end_time:%Y%m%dT%H%M%SZ}_
↪{processing_mode}_{disposition_mode}_{creation_time}_{version_int:04d}.nc'
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene

filenames = ['data/AVHRR-GAC_FDR_1C_N06_19810330T042358Z_19810330T060903Z_R_O_
↪20200101T000000Z_0100.nc']
scn = Scene(reader='avhrr_l1c_eum_gac_fdr_nc', filenames=filenames)
scn.load(['brightness_temperature_channel_4'])
scn['brightness_temperature_channel_4']
```

Output:

```
<xarray.DataArray 'brightness_temperature_channel_4' (y: 11, x: 409)>
dask.array<open_dataset-55ffbf3623b32077c67897f4283640a5brightness_temperature_channel_4,
↪ shape=(11, 409),
    dtype=float32, chunksize=(11, 409), chunktype=numpy.ndarray>
Coordinates:
  * x          (x) int16 0 1 2 3 4 5 6 7 8 ... 401 402 403 404 405 406 407 408
  * y          (y) int64 0 1 2 3 4 5 6 7 8 9 10
    acq_time   (y) datetime64[ns] dask.array<chunksize=(11,), meta=np.ndarray>
    longitude  (y, x) float64 dask.array<chunksize=(11, 409), meta=np.ndarray>
    latitude   (y, x) float64 dask.array<chunksize=(11, 409), meta=np.ndarray>
Attributes:
    start_time:                           1981-03-30 04:23:58
    end_time:                             1981-03-30 06:09:03
    calibration:                          brightness_temperature
    modifiers:                            ()
    resolution:                           1050
    standard_name:                        toa_brightness_temperature
    units:                                K
    wavelength:                           10.8µm (10.3-11.3µm)
    Conventions:                          CF-1.8 ACDD-1.3
    comment:                              Developed in cooperation with EUME...
    creator_email:                        ops@eumetsat.int
    creator_name:                         EUMETSAT
    creator_url:                          https://www.eumetsat.int/
    date_created:                         2020-09-14T10:50:51.073707
    disposition_mode:                     O
```

(continues on next page)

(continued from previous page)

```
    gac_filename:                          NSS.GHRR.NA.D81089.S0423.E0609.B09...
    geospatial_lat_max:                    89.95386902434623
    geospatial_lat_min:                    -89.97581969005503
    geospatial_lat_resolution:             1050 meters
    geospatial_lat_units:                  degrees_north
    geospatial_lon_max:                    179.99952992568998
    geospatial_lon_min:                    -180.0
    geospatial_lon_resolution:             1050 meters
    geospatial_lon_units:                  degrees_east
    ground_station:                        GC
    id:                                    DOI:10.5676/EUM/AVHRR_GAC_L1C_FDR/...
    institution:                           EUMETSAT
    instrument:                            Earth Remote Sensing Instruments >...
    keywords:                              ATMOSPHERE > ATMOSPHERIC RADIATION...
    keywords_vocabulary:                   GCMD Science Keywords, Version 9.1
    licence:                               EUMETSAT data policy https://www.e...
    naming_authority:                      int.eumetsat
    orbit_number_end:                      9123
    orbit_number_start:                    9122
    orbital_parameters_tle:                ['1 11416U 79057A   81090.16350942...
    platform:                              Earth Observation Satellites > NOA...
    processing_level:                      1C
    processing_mode:                       R
    product_version:                       1.0.0
    references:                            Devasthale, A., M. Raspaud, C. Sch...
    source:                                AVHRR GAC Level 1 Data
    standard_name_vocabulary:              CF Standard Name Table v73
    summary:                               Fundamental Data Record (FDR) of m...
    sun_earth_distance_correction_factor:  0.9975244779999585
    time_coverage_end:                     19820803T003900Z
    time_coverage_start:                   19800101T000000Z
    title:                                 AVHRR GAC L1C FDR
    version_calib_coeffs:                  PATMOS-x, v2017r1
    version_pygac:                         1.4.0
    version_pygac_fdr:                     0.1.dev107+gceb7b26.d20200910
    version_satpy:                         0.21.1.dev894+g5cf76e6
    history:                               Created by pytroll/satpy on 2020-0...
    name:                                  brightness_temperature_channel_4
    _satpy_id:                             DataID(name='brightness_temperatur...
    ancillary_variables:                   []
```

hdf5 based readers

Advanced Geostationary Radiation Imager reader for the Level_1 HDF format.

The files read by this reader are described in the official Real Time Data Service:

http://fy4.nsmc.org.cn/data/en/data/realtime.html

Geostationary High-speed Imager reader for the Level_1 HDF format.

This instrument is aboard the Fengyun-4B satellite. No document is available to describe this format is available, but it's broadly similar to the co-flying AGRI instrument.

Arctica-M N1 HDF5 format reader

Reader for the Arctica-M1 MSU-GS/A data.

The files for this reader are HDF5 and contain channel data at 1km resolution for the VIS channels and 4km resolution for the IR channels. Geolocation data is available at both resolutions, as is sun and satellite geometry.

This reader was tested on sample data provided by EUMETSAT.

2.8 Reading remote files

2.8.1 Using a single reader

Some of the readers in Satpy can read data directly over various transfer protocols. This is done using fsspec and various packages it is using underneath.

As an example, reading ABI data from public AWS S3 storage can be done in the following way:

```
from satpy import Scene

storage_options = {'anon': True}
filenames = ['s3://noaa-goes16/ABI-L1b-RadC/2019/001/17/*_G16_s20190011702186*']
scn = Scene(reader='abi_l1b', filenames=filenames, reader_kwargs={'storage_options':␣
↪storage_options})
scn.load(['true_color_raw'])
```

Reading from S3 as above requires the *s3fs* library to be installed in addition to *fsspec*.

As an alternative, the storage options can be given using fsspec configuration. For the above example, the configuration could be saved to *s3.json* in the *fsspec* configuration directory (by default placed in *~/.config/fsspec/* directory in Linux):

```
{
    "s3": {
        "anon": "true"
    }
}
```

Note: Options given in *reader_kwargs* override only the matching options given in configuration file and everythin else is left as-is. In case of problems in data access, remove the configuration file to see if that solves the issue.

For reference, reading SEVIRI HRIT data from a local S3 storage works the same way:

```
filenames = [
    's3://satellite-data-eumetcast-seviri-rss/H-000-MSG3*202204260855*',
]
storage_options = {
    "client_kwargs": {"endpoint_url": "https://PLACE-YOUR-SERVER-URL-HERE"},
    "secret": "VERYBIGSECRET",
    "key": "ACCESSKEY"
}
scn = Scene(reader='seviri_l1b_hrit', filenames=filenames, reader_kwargs={'storage_
```

(continues on next page)

(continued from previous page)

```
↪options': storage_options})
scn.load(['WV_073'])
```

Using the *fsspec* configuration in *s3.json* the configuration would look like this:

```
{
    "s3": {
        "client_kwargs": {"endpoint_url": "https://PLACE-YOUR-SERVER-URL-HERE"},
        "secret": "VERYBIGSECRET",
        "key": "ACCESSKEY"
    }
}
```

2.8.2 Using multiple readers

If multiple readers are used and the required credentials differ, the storage options are passed per reader like this:

```
reader1_filenames = [...]
reader2_filenames = [...]
filenames = {
    'reader1': reader1_filenames,
    'reader2': reader2_filenames,
}
reader1_storage_options = {...}
reader2_storage_options = {...}
reader_kwargs = {
    'reader1': {
        'option1': 'foo',
        'storage_options': reader1_storage_options,
    },
    'reader2': {
        'option1': 'foo',
        'storage_options': reader1_storage_options,
    }
}
scn = Scene(filenames=filenames, reader_kwargs=reader_kwargs)
```

2.8.3 Caching the remote files

Caching the remote file locally can speedup the overall processing time significantly, especially if the data are re-used for example when testing. The caching can be done by taking advantage of the fsspec caching mechanism:

```
reader_kwargs = {
    'storage_options': {
        's3': {'anon': True},
        'simple': {
            'cache_storage': '/tmp/s3_cache',
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
filenames = ['simplecache::s3://noaa-goes16/ABI-L1b-RadC/2019/001/17/*_G16_
↪s20190011702186*']
scn = Scene(reader='abi_l1b', filenames=filenames, reader_kwargs=reader_kwargs)
scn.load(['true_color_raw'])
scn2 = scn.resample(scn.coarsest_area(), resampler='native')
scn2.save_datasets(base_dir='/tmp/', tiled=True, blockxsize=512, blockysize=512, driver=
↪'COG', overviews=[])
```

The following table shows the timings for running the above code with different cache statuses:

```
.. _cache_timing_table:
```

Table 1: Processing times without and with caching

Caching	Elapsed time	Notes
No caching	650 s	remove *reader_kwargs* and *simplecache::* from the code
File cache	66 s	Initial run
File cache	13 s	Second run

Note: The cache is not cleaned by Satpy nor fsspec so the user should handle cleaning excess files from *cache_storage*.

Note: Only *simplecache* is considered thread-safe, so using the other caching mechanisms may or may not work depending on the reader, Dask scheduler or the phase of the moon.

2.8.4 Resources

See `FSFile` for direct usage of *fsspec* with Satpy, and fsspec documentation for more details on connection options and detailes.

2.9 Composites

Composites are defined as arrays of data that are created by processing and/or combining one or multiple data arrays (prerequisites) together.

Composites are generated in satpy using Compositor classes. The attributes of the resulting composites are usually a combination of the prerequisites' attributes and the key/values of the DataID used to identify it.

2.9.1 Built-in Compositors

There are many built-in compositors available in Satpy. The majority use the *GenericCompositor* base class which handles various image modes (*L*, *LA*, *RGB*, and *RGBA* at the moment) and updates attributes.

The below sections summarize the composites that come with Satpy and show basic examples of creating and using them with an existing *Scene* object. It is recommended that any composites that are used repeatedly be configured in YAML configuration files. General-use compositor code dealing with visible or infrared satellite data can be put in a configuration file called `visir.yaml`. Composites that are specific to an instrument can be placed in YAML config files named accordingly (e.g., `seviri.yaml` or `viirs.yaml`). See the satpy repository for more examples.

GenericCompositor

GenericCompositor class can be used to create basic single channel and RGB composites. For example, building an overview composite can be done manually within Python code with:

```
>>> from satpy.composites import GenericCompositor
>>> compositor = GenericCompositor("overview")
>>> composite = compositor([local_scene[0.6],
...                         local_scene[0.8],
...                         local_scene[10.8]])
```

One important thing to notice is that there is an internal difference between a composite and an image. A composite is defined as a special dataset which may have several bands (like *R*, *G* and *B* bands). However, the data isn't stretched, or clipped or gamma filtered until an image is generated. To get an image out of the above composite:

```
>>> from satpy.writers import to_image
>>> img = to_image(composite)
>>> img.invert([False, False, True])
>>> img.stretch("linear")
>>> img.gamma(1.7)
>>> img.show()
```

This part is called *enhancement*, and is covered in more detail in *Enhancements*.

Single channel composites can also be generated with the *GenericCompositor*, but in some cases, the *SingleBandCompositor* may be more appropriate. For example, the *GenericCompositor* removes attributes such as `units` because they are typically not meaningful for an RGB image. Such attributes are retained in the *SingleBandCompositor*.

DifferenceCompositor

DifferenceCompositor calculates a difference of two datasets:

```
>>> from satpy.composites import DifferenceCompositor
>>> compositor = DifferenceCompositor("diffcomp")
>>> composite = compositor([local_scene[10.8], local_scene[12.0]])
```

FillingCompositor

FillingCompositor:: fills the missing values in three datasets with the values of another dataset::

```
>>> from satpy.composites import FillingCompositor
>>> compositor = FillingCompositor("fillcomp")
>>> filler = local_scene[0.6]
>>> data_with_holes_1 = local_scene['ch_a']
>>> data_with_holes_2 = local_scene['ch_b']
>>> data_with_holes_3 = local_scene['ch_c']
>>> composite = compositor([filler, data_with_holes_1, data_with_holes_2,
...                         data_with_holes_3])
```

PaletteCompositor

PaletteCompositor creates a color version of a single channel categorical dataset using a colormap:

```
>>> from satpy.composites import PaletteCompositor
>>> compositor = PaletteCompositor("palcomp")
>>> composite = compositor([local_scene['cma'], local_scene['cma_pal']])
```

The palette should have a single entry for all the (possible) values in the dataset mapping the value to an RGB triplet. Typically the palette comes with the categorical (e.g. cloud mask) product that is being visualized.

Deprecated since version 0.40: Composites produced with *PaletteCompositor* will result in an image with mode RGB when enhanced. To produce an image with mode P, use the *SingleBandCompositor* with an associated *palettize()* enhancement and pass `keep_palette=True` to `save_datasets()`. If the colormap is sourced from the same dataset as the dataset to be palettized, it must be contained in the auxiliary datasets.

Since Satpy 0.40, all built-in composites that used *PaletteCompositor* have been migrated to use *SingleBandCompositor* instead. This has no impact on resulting images unless `keep_palette=True` is passed to `save_datasets()`, but the loaded composite now has only one band (previously three).

DayNightCompositor

DayNightCompositor merges two different composites. The first composite will be placed on the day-side of the scene, and the second one on the night side. The transition from day to night is done by calculating solar zenith angle (SZA) weighed average of the two composites. The SZA can optionally be given as third dataset, and if not given, the angles will be calculated. Four arguments are used to generate the image (default values shown in the example below). They can be defined when initializing the compositor:

```
- lim_low (float): lower limit of Sun zenith angle for the
                   blending of the given channels
- lim_high (float): upper limit of Sun zenith angle for the
                    blending of the given channels
                    Together with `lim_low` they define the width
                    of the blending zone
- day_night (string): "day_night" means both day and night portions will be kept
                      "day_only" means only day portion will be kept
                      "night_only" means only night portion will be kept
- include_alpha (bool): This only affects the "day only" or "night only" result.
                        True means an alpha band will be added to the output image for␣
↪transparency.
```

(continues on next page)

(continued from previous page)

```
                        False means the output is a single-band image with undesired␣
↪pixels being masked out
                        (replaced with NaNs).
```

Usage (with default values):

```
>>> from satpy.composites import DayNightCompositor
>>> compositor = DayNightCompositor("dnc", lim_low=85., lim_high=88., day_night="day_
↪night")
>>> composite = compositor([local_scene['true_color'],
...                         local_scene['night_fog']])
```

As above, with *day_night* flag it is also available to use only a day product or only a night product and mask out (make transparent) the opposite portion of the image (night or day). The example below provides only a day product with night portion masked-out:

```
>>> from satpy.composites import DayNightCompositor
>>> compositor = DayNightCompositor("dnc", lim_low=85., lim_high=88., day_night="day_only
↪")
>>> composite = compositor([local_scene['true_color']])
```

By default, the image under *day_only* or *night_only* flag will come out with an alpha band to display its transparency. It could be changed by setting *include_alpha* to False if there's no need for that alpha band. In such cases, it is recommended to use it together with *fill_value=0* when saving to geotiff to get a single-band image with black background. In the case below, the image shows its day portion and day/night transition with night portion blacked-out instead of transparent:

```
>>> from satpy.composites import DayNightCompositor
>>> compositor = DayNightCompositor("dnc", lim_low=85., lim_high=88., day_night="day_only
↪", include_alpha=False)
>>> composite = compositor([local_scene['true_color']])
```

RealisticColors

`RealisticColors` compositor is a special compositor that is used to create realistic near-true-color composite from MSG/SEVIRI data:

```
>>> from satpy.composites import RealisticColors
>>> compositor = RealisticColors("realcols", lim_low=85., lim_high=95.)
>>> composite = compositor([local_scene['VIS006'],
...                         local_scene['VIS008'],
...                         local_scene['HRV']])
```

CloudCompositor

CloudCompositor can be used to threshold the data so that "only" clouds are visible. These composites can be used as an overlay on top of e.g. static terrain images to show a rough idea where there are clouds. The data are thresholded using three variables:

```
- `transition_min`: values below or equal to this are clouds -> opaque white
- `transition_max`: values above this are cloud free -> transparent
- `transition_gamma`: gamma correction applied to clarify the clouds
```

Usage (with default values):

```
>>> from satpy.composites import CloudCompositor
>>> compositor = CloudCompositor("clouds", transition_min=258.15,
...                              transition_max=298.15,
...                              transition_gamma=3.0)
>>> composite = compositor([local_scene[10.8]])
```

Support for using this compositor for VIS data, where the values for high/thick clouds tend to be in reverse order to brightness temperatures, is to be added.

RatioSharpenedRGB

RatioSharpenedRGB

SelfSharpenedRGB

SelfSharpenedRGB sharpens the RGB with ratio of a band with a strided version of itself.

LuminanceSharpeningCompositor

LuminanceSharpeningCompositor replaces the luminance from an RGB composite with luminance created from reflectance data. If the resolutions of the reflectance data _and_ of the target area definition are higher than the base RGB, more details can be retrieved. This compositor can be useful also with matching resolutions, e.g. to highlight shadowing at cloudtops in colorized infrared composite.

```
>>> from satpy.composites import LuminanceSharpeningCompositor
>>> compositor = LuminanceSharpeningCompositor("vis_sharpened_ir")
>>> vis_data = local_scene['HRV']
>>> colorized_ir_clouds = local_scene['colorized_ir_clouds']
>>> composite = compositor([vis_data, colorized_ir_clouds])
```

SandwichCompositor

Similar to *LuminanceSharpeningCompositor*, *SandwichCompositor* uses reflectance data to bring out more details out of infrared or low-resolution composites. *SandwichCompositor* multiplies the RGB channels with (scaled) reflectance.

```
>>> from satpy.composites import SandwichCompositor
>>> compositor = SandwichCompositor("ir_sandwich")
>>> vis_data = local_scene['HRV']
>>> colorized_ir_clouds = local_scene['colorized_ir_clouds']
>>> composite = compositor([vis_data, colorized_ir_clouds])
```

StaticImageCompositor

StaticImageCompositor can be used to read an image from disk and used just like satellite data, including resampling and using as a part of other composites.

```
>>> from satpy.composites import StaticImageCompositor
>>> compositor = StaticImageCompositor("static_image", filename="image.tif")
>>> composite = compositor()
```

BackgroundCompositor

BackgroundCompositor can be used to stack two composites together. If the composites don't have *alpha* channels, the *background* is used where *foreground* has no data. If *foreground* has alpha channel, the *alpha* values are used to weight when blending the two composites.

```
>>> from satpy import Scene
>>> from satpy.composites import BackgroundCompositor
>>> compositor = BackgroundCompositor()
>>> clouds = local_scene['ir_cloud_day']
>>> background = local_scene['overview']
>>> composite = compositor([clouds, background])
```

CategoricalDataCompositor

CategoricalDataCompositor can be used to recategorize categorical data. This is for example useful to combine comparable categories into a common category. The category remapping from *data* to *composite* is done using a look-up-table (*lut*):

```
composite = [[lut[data[0,0]], lut[data[0,1]], lut[data[0,Nj]]],
             [[lut[data[1,0]], lut[data[1,1]], lut[data[1,Nj]]],
             [[lut[data[Ni,0]], lut[data[Ni,1]], lut[data[Ni,Nj]]]
```

Hence, *lut* must have a length that is greater than the maximum value in *data* in orer to avoid an *IndexError*. Below is an example on how to create a binary clear-sky/cloud mask from a pseodu cloud type product with six categories representing clear sky (cat1/cat5), cloudy features (cat2-cat4) and missing/undefined data (cat0):

```
>>> cloud_type = local_scene['cloud_type']  # 0 - cat0, 1 - cat1, 2 - cat2, 3 - cat3, 4 -
↪ cat4, 5 - cat5,
# categories: 0    1  2  3  4  5
```

(continues on next page)

(continued from previous page)

```
>>> lut = [np.nan, 0, 1, 1, 1, 0]
>>> compositor = CategoricalDataCompositor('binary_cloud_mask', lut=lut)
>>> composite = compositor([cloud_type])  # 0 - cat1/cat5, 1 - cat2/cat3/cat4, nan - cat0
```

2.9.2 Creating composite configuration files

To save the custom composite, follow the *Component Configuration* documentation. Once your component configuration directory is created you can create your custom composite YAML configuration files. Compositors that can be used for multiple instruments can be placed in the generic `$SATPY_CONFIG_PATH/composites/visir.yaml` file. Composites that are specific to one sensor should be placed in `$SATPY_CONFIG_PATH/composites/<sensor>.yaml`. Custom enhancements for your new composites can be stored in `$SATPY_CONFIG_PATH/enhancements/generic.yaml` or `$SATPY_CONFIG_PATH/enhancements/<sensor>.yaml`.

With that, you should be able to load your new composite directly. Example configuration files can be found in the satpy repository as well as a few simple examples below.

Simple RGB composite

This is the overview composite shown in the first code example above using *GenericCompositor*:

```
sensor_name: visir

composites:
  overview:
    compositor: !!python/name:satpy.composites.GenericCompositor
    prerequisites:
    - 0.6
    - 0.8
    - 10.8
    standard_name: overview
```

For an instrument specific version (here MSG/SEVIRI), we should use the channel _names_ instead of wavelengths. Note also that the sensor_name is now combination of visir and seviri, which means that it extends the generic visir composites:

```
sensor_name: visir/seviri

composites:

  overview:
    compositor: !!python/name:satpy.composites.GenericCompositor
    prerequisites:
    - VIS006
    - VIS008
    - IR_108
    standard_name: overview
```

In the following examples only the composite receipes are shown, and the header information (sensor_name, composites) and intendation needs to be added.

Using modifiers

In many cases the basic datasets that go into the composite need to be adjusted, e.g. for Solar zenith angle normalization. These modifiers can be applied in the following way:

```
overview:
  compositor: !!python/name:satpy.composites.GenericCompositor
  prerequisites:
  - name: VIS006
    modifiers: [sunz_corrected]
  - name: VIS008
    modifiers: [sunz_corrected]
  - IR_108
  standard_name: overview
```

Here we see two changes:

1. channels with modifiers need to have either *name* or *wavelength* added in front of the channel name or wavelength, respectively
2. a list of modifiers attached to the dictionary defining the channel

The modifier above is a built-in that normalizes the Solar zenith angle to Sun being directly at the zenith.

More examples can be found in Satpy source code directory satpy/etc/composites.

See the modifiers documentation for more information on available built-in modifiers.

Using other composites

Often it is handy to use other composites as a part of the composite. In this example we have one composite that relies on solar channels on the day side, and another for the night side:

```
natural_with_night_fog:
  compositor: !!python/name:satpy.composites.DayNightCompositor
  prerequisites:
    - natural_color
    - night_fog
  standard_name: natural_with_night_fog
```

This compositor has three additional keyword arguments that can be defined (shown with the default values, thus identical result as above):

```
natural_with_night_fog:
  compositor: !!python/name:satpy.composites.DayNightCompositor
  prerequisites:
    - natural_color
    - night_fog
  lim_low: 85.0
  lim_high: 88.0
  day_night: "day_night"
  standard_name: natural_with_night_fog
```

Defining other composites in-line

It is also possible to define sub-composites in-line. This example is the built-in airmass composite:

```
airmass:
  compositor: !!python/name:satpy.composites.GenericCompositor
  prerequisites:
  - compositor: !!python/name:satpy.composites.DifferenceCompositor
    prerequisites:
    - wavelength: 6.2
    - wavelength: 7.3
  - compositor: !!python/name:satpy.composites.DifferenceCompositor
    prerequisites:
      - wavelength: 9.7
      - wavelength: 10.8
  - wavelength: 6.2
  standard_name: airmass
```

Using a pre-made image as a background

Below is an example composite config using `StaticImageCompositor`, `DayNightCompositor`, `CloudCompositor` and `BackgroundCompositor` to show how to create a composite with a blended day/night imagery as background for clouds. As the images are in PNG format, and thus not georeferenced, the name of the area definition for the background images are given. When using GeoTIFF images the *area* parameter can be left out.

Note: The background blending uses the current time if there is no timestamps in the image filenames.

```
clouds_with_background:
  compositor: !!python/name:satpy.composites.BackgroundCompositor
  standard_name: clouds_with_background
  prerequisites:
    - ir_cloud_day
    - compositor: !!python/name:satpy.composites.DayNightCompositor
      prerequisites:
        - static_day
        - static_night

static_day:
  compositor: !!python/name:satpy.composites.StaticImageCompositor
  standard_name: static_day
  filename: /path/to/day_image.png
  area: euro4

static_night:
  compositor: !!python/name:satpy.composites.StaticImageCompositor
  standard_name: static_night
  filename: /path/to/night_image.png
  area: euro4
```

To ensure that the images aren't auto-stretched and possibly altered, the following should be added to enhancement config (assuming 8-bit image) for both of the static images:

```
static_day:
  standard_name: static_day
  operations:
  - name: stretch
    method: !!python/name:satpy.enhancements.stretch
    kwargs:
      stretch: crude
      min_stretch: [0, 0, 0]
      max_stretch: [255, 255, 255]
```

2.9.3 Enhancing the images

After the composite is defined and created, it needs to be converted to an image. To do this, it is necessary to describe how the data values are mapped to values stored in the image format. This procedure is called `stretching`, and in Satpy it is implemented by `enhancements`.

The first step is to convert the composite to an `XRImage` object:

```
>>> from satpy.writers import to_image
>>> img = to_image(composite)
```

Now it is possible to apply enhancements available in the class:

```
>>> img.invert([False, False, True])
>>> img.stretch("linear")
>>> img.gamma(1.7)
```

And finally either show or save the image:

```
>>> img.show()
>>> img.save('image.tif')
```

As pointed out in the composite section, it is better to define frequently used enhancements in configuration files under `$SATPY_CONFIG_PATH/enhancements/`. The enhancements can either be in `generic.yaml` or instrument-specific file (e.g., `seviri.yaml`).

The above enhancement can be written (with the headers necessary for the file) as:

```
enhancements:
  overview:
    standard_name: overview
    operations:
      - name: inverse
        method: !!python/name:satpy.enhancements.invert
        args: [False, False, True]
      - name: stretch
        method: !!python/name:satpy.enhancements.stretch
        kwargs:
          stretch: linear
      - name: gamma
        method: !!python/name:satpy.enhancements.gamma
        kwargs:
          gamma: [1.7, 1.7, 1.7]
```

Warning: If you define a composite with no matching enhancement, Satpy will by default apply the `stretch_linear()` enhancement with cutoffs of 0.5% and 99.5%. If you want no enhancement at all (maybe you are enhancing a composite based on *DayNightCompositor* where the components have their own enhancements defined), you need to define an enhancement that does nothing:

```
enhancements:
  day_x:
    standard_name: day_x
    operations: []
```

It is recommended to define an enhancement even if you intend to use the default, in case the default should change in future versions of Satpy.

More examples can be found in Satpy source code directory `satpy/etc/enhancements/generic.yaml`.

See the *Enhancements* documentation for more information on available built-in enhancements.

2.9.4 Modifiers

Modifiers are filters applied to datasets prior to computing composites. They take at least one input (a dataset) and have exactly one output (the same dataset, modified). They can take additional input datasets or parameters.

Modifiers are defined in composites files in `etc/composites` within `$SATPY_CONFIG_PATH`.

The instruction to use a certain modifier can be contained in a composite definition or in a reader definition. If it is defined in a composite definition, it is applied upon constructing the composite.

When using built-in composites, Satpy users do not need to understand the mechanics of modifiers, as they are applied automatically. The *Composites* documentation contains information on how to apply modifiers when creating new composites.

Some readers read data where certain modifiers are already applied. Here, the reader definition will refer to the Satpy modifier. This marking adds the modifier to the metadata to prevent it from being applied again upon composite calculation.

Commonly used modifiers are listed in the table below. Further details on those modifiers can be found in the linked API documentation.

Table 2: Commonly used modifiers

Label	Class	Description
`sunz_corrected`	*SunZenithCorrector*	Modifies solar channels for the solar zenith angle to provide smoother images.
`effective_solar_pathle`	*EffectiveSolarPathLe*	Modifies solar channels for atmospheric path length of solar radiation.
`nir_reflectance`	*NIRReflectance*	Calculates reflective part of channels at the edge of solar and terrestrial radiation (3.7 µm or 3.9 µm).
`nir_emissive`	*NIREmissivePartFromR*	Calculates emissive part of channels at the edge of solar and terrestrial radiation (3.7 µm or 3.9 µm)
`rayleigh_corrected`	*PSPRayleighReflectan*	Modifies solar channels to filter out the visual impact of rayleigh scattering.

A complete list can be found in the etc/composites source code and in the *modifiers* module documentation.

Parallax correction

Warning: The Satpy parallax correction is experimental and subject to change.

Since version 0.37 (mid 2022), Satpy has included a modifier for parallax correction, implemented in the `ParallaxCorrectionModifier` class. This modifier is important for some applications, but not applied by default to any Satpy datasets or composites, because it can be applied to any input dataset and used with any source of (cloud top) height. Therefore, users wishing to apply the parallax correction semi-automagically have to define their own modifier and then apply that modifier for their datasets. An example is included with the `ParallaxCorrectionModifier` API documentation. Note that Satpy cannot apply modifiers to composites, so users wishing to apply parallax correction to a composite will have to use a lower level API or duplicate an existing composite recipe to use modified inputs.

The parallax correction is directly calculated from the cloud top height. Information on satellite position is obtained from cloud top height metadata. If no orbital parameters are present in the cloud top height metadata, Satpy will attempt to calculate orbital parameters from the platform name and start time. The backup calculation requires skyfield and astropy to be installed. If the metadata include neither orbital parameters nor platform name and start time, parallax calculation will fail. Because the cloud top height metadata are used, it is essential that the cloud top height data are derived from the same platform as the measurements to be corrected are taken by.

The parallax error moves clouds away from the observer. Therefore, the parallax correction shifts clouds in the direction of the observer. The space left behind by the cloud will be filled with fill values. As the cloud is shifted toward the observer, it may occupy less pixels than before, because pixels closer to the observer have a smaller surface area. It can also be deformed (a "rectangular" cloud may get the shape of a parallelogram).

The utility function `get_surface_parallax_displacement()` allows to calculate the magnitude of the parallax error. For a cloud with a cloud top height of 10 km:

The parallax correction is currently experimental and subject to change. Although it is covered by tests, there may be cases that yield unexpected or incorrect results. It does not yet perform any checks that the provided (cloud top) height covers the area of the dataset for which the parallax correction shall be applied.

For more general background information and web routines related to the parallax effect, see also *this collection at the CIMSS website <https://cimss.ssec.wisc.edu/goes/webapps/parallax/>_*.

New in version 0.37.

2.10 Resampling

Resampling in Satpy.

Satpy provides multiple resampling algorithms for resampling geolocated data to uniform projected grids. The easiest way to perform resampling in Satpy is through the `Scene` object's `resample()` method. Additional utility functions are also available to assist in resampling data. Below is more information on resampling with Satpy as well as links to the relevant API documentation for available keyword arguments.

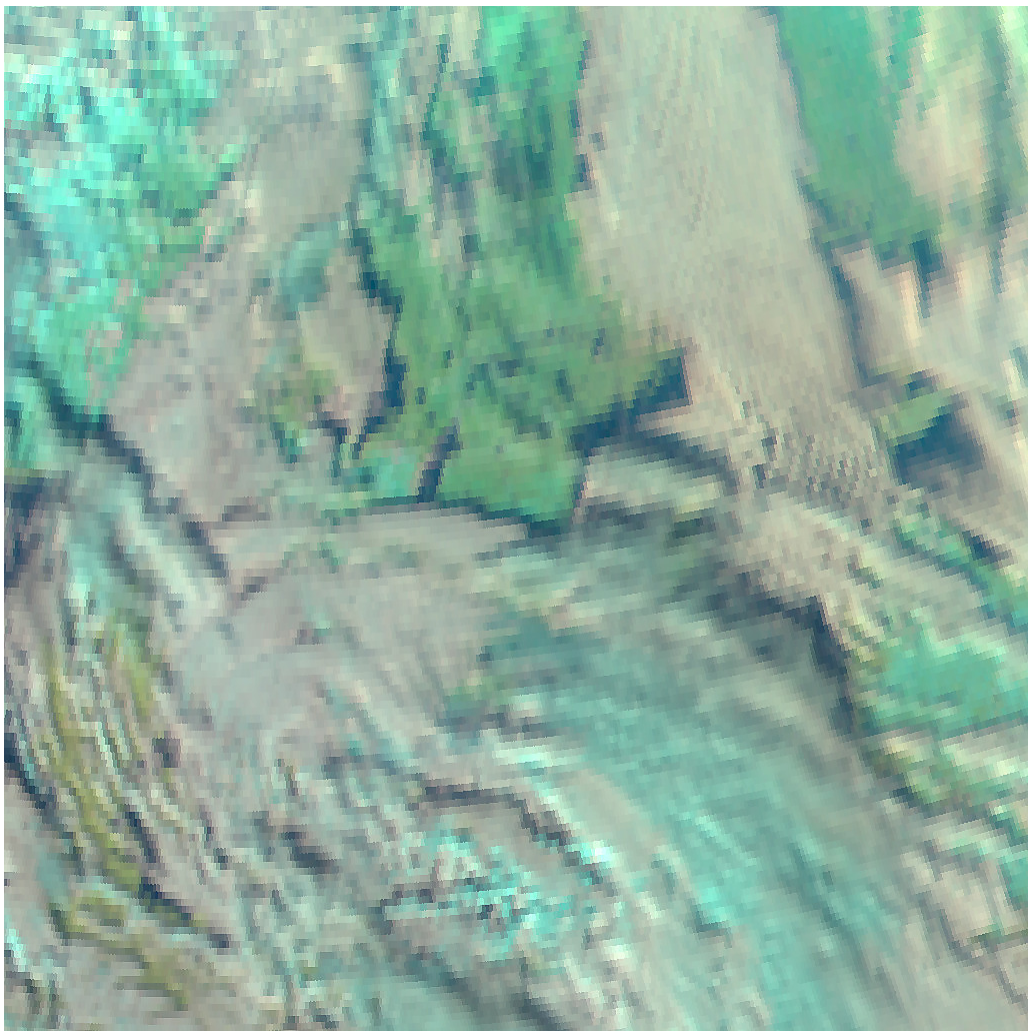

Fig. 1: SEVIRI view of southern Sweden, 2021-11-30 12:15Z, without parallax correction. This is the `natural_color` composite as built into Satpy.

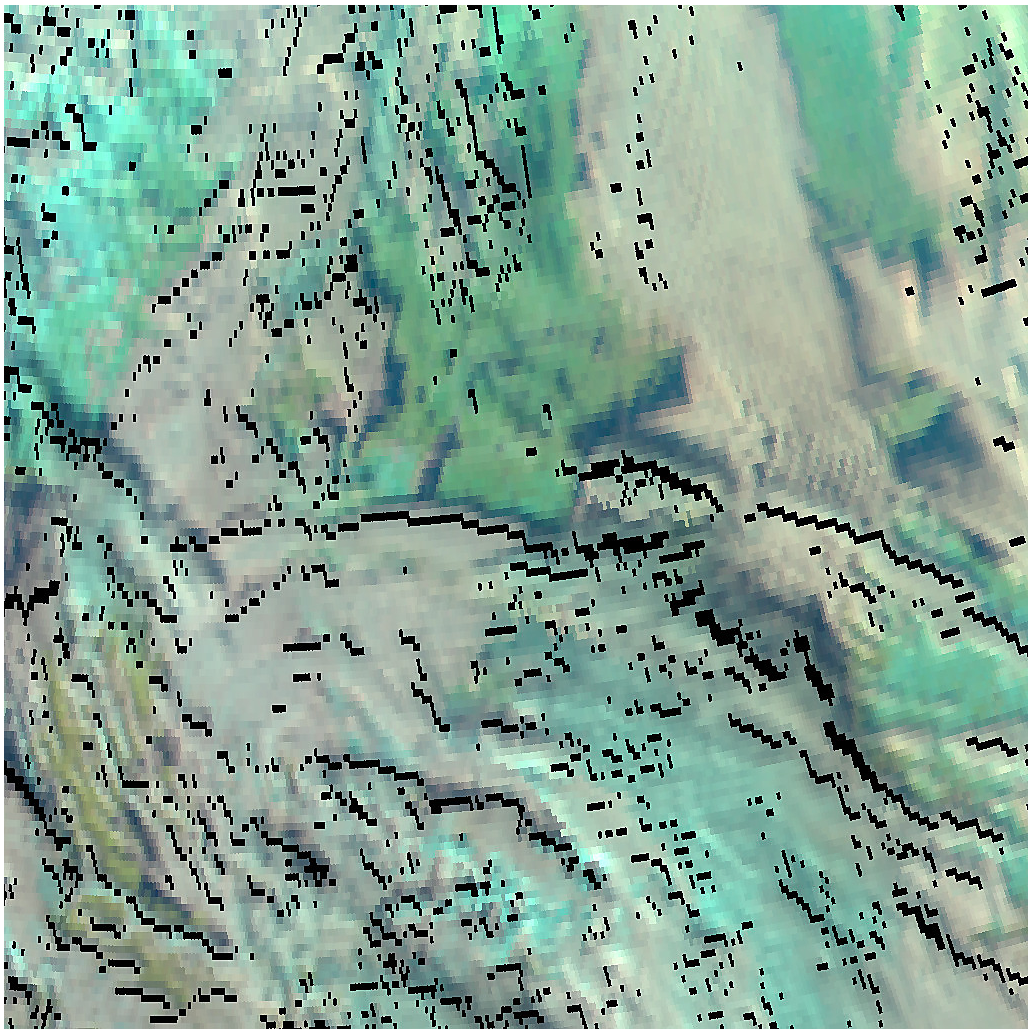

Fig. 2: The same satellite view with parallax correction. The most obvious change are the gaps left behind by the parallax correction, shown as black pixels. Otherwise it shows that clouds have "moved" south-south-west in the direction of the satellite. To view the images side-by-side or alternating, look at the figshare page

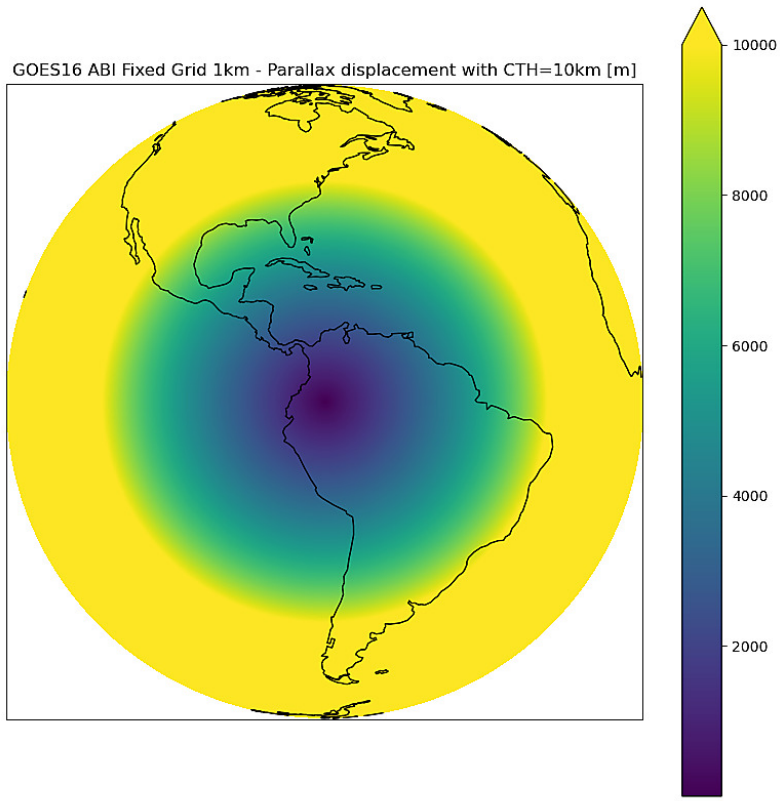

Fig. 3: Magnitude of the parallax error for a fictitious cloud with a cloud top height of 10 km for the GOES-East (GOES-16) full disc.

2.10.1 Resampling algorithms

Table 3: Available Resampling Algorithms

Resampler	Description	Related
nearest	Nearest Neighbor	*KDTreeResampler*
ewa	Elliptical Weighted Averaging	DaskEWAResampler
ewa_legacy	Elliptical Weighted Averaging (Legacy)	LegacyDaskEWAResampler
native	Native	*NativeResampler*
bilinear	Bilinear	*BilinearResampler*
bucket_avg	Average Bucket Resampling	*BucketAvg*
bucket_sum	Sum Bucket Resampling	*BucketSum*
bucket_count	Count Bucket Resampling	*BucketCount*
bucket_fraction	Fraction Bucket Resampling	*BucketFraction*
gradient_search	Gradient Search Resampling	GradientSearchResampler

The resampling algorithm used can be specified with the `resampler` keyword argument and defaults to `nearest`:

```
>>> scn = Scene(...)
>>> euro_scn = scn.resample('euro4', resampler='nearest')
```

Warning: Some resampling algorithms expect certain forms of data. For example, the EWA resampling expects polar-orbiting swath data and prefers if the data can be broken in to "scan lines". See the API documentation for a specific algorithm for more information.

2.10.2 Resampling for comparison and composites

While all the resamplers can be used to put datasets of different resolutions on to a common area, the 'native' resampler is designed to match datasets to one resolution in the dataset's original projection. This is extremely useful when generating composites between bands of different resolutions.

```
>>> new_scn = scn.resample(resampler='native')
```

By default this resamples to the *highest resolution area* (smallest footprint per pixel) shared between the loaded datasets. You can easily specify the lowest resolution area:

```
>>> new_scn = scn.resample(scn.coarsest_area(), resampler='native')
```

Providing an area that is neither the minimum or maximum resolution area may work, but behavior is currently undefined.

2.10.3 Caching for geostationary data

Satpy will do its best to reuse calculations performed to resample datasets, but it can only do this for the current processing and will lose this information when the process/script ends. Some resampling algorithms, like `nearest` and `bilinear`, can benefit by caching intermediate data on disk in the directory specified by *cache_dir* and using it next time. This is most beneficial with geostationary satellite data where the locations of the source data and the target pixels don't change over time.

```
>>> new_scn = scn.resample('euro4', cache_dir='/path/to/cache_dir')
```

See the documentation for specific algorithms to see availability and limitations of caching for that algorithm.

2.10.4 Create custom area definition

See `pyresample.geometry.AreaDefinition` for information on creating areas that can be passed to the resample method:

```
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> local_scene = scn.resample(my_area)
```

2.10.5 Create dynamic area definition

See `pyresample.geometry.DynamicAreaDefinition` for more information.

Examples coming soon…

2.10.6 Store area definitions

Area definitions can be saved to a custom YAML file (see pyresample's writing to disk) and loaded using pyresample's utility methods (pyresample's loading from disk):

```
>>> from pyresample import load_area
>>> my_area = load_area('my_areas.yaml', 'my_area')
```

Or using *satpy.resample.get_area_def()*, which will search through all `areas.yaml` files in your `SATPY_CONFIG_PATH`:

```
>>> from satpy.resample import get_area_def
>>> area_eurol = get_area_def("eurol")
```

For examples of area definitions, see the file `etc/areas.yaml` that is included with Satpy and where all the area definitions shipped with Satpy are defined.

2.11 Enhancements

2.11.1 Built-in enhancement methods

stretch

The most basic operation is to stretch the image so that the data fits to the output format. There are many different ways to stretch the data, which are configured by giving them in *kwargs* dictionary, like in the example above. The default, if nothing else is defined, is to apply a linear stretch. For more details, see *enhancing the images*.

linear

As the name suggests, linear stretch converts the input values to output values in a linear fashion. By default, 5% of the data is cut on both ends of the scale, but these can be overridden with `cutoffs=(0.005, 0.005)` argument:

```
- name: stretch
  method: !!python/name:satpy.enhancements.stretch
  kwargs:
    stretch: linear
    cutoffs: [0.003, 0.005]
```

Note: This enhancement is currently not optimized for dask because it requires getting minimum/maximum information for the entire data array.

crude

The crude stretching is used to limit the input values to a certain range by clipping the data. This is followed by a linear stretch with no cutoffs specified (see above). Example:

```
- name: stretch
  method: !!python/name:satpy.enhancements.stretch
  kwargs:
    stretch: crude
    min_stretch: [0, 0, 0]
    max_stretch: [100, 100, 100]
```

It is worth noting that this stretch can also be used to _invert_ the data by giving larger values to the min_stretch than to max_stretch.

histogram

gamma

invert

piecewise_linear_stretch

Use `numpy.interp()` to linearly interpolate data to a new range. See *satpy.enhancements.piecewise_linear_stretch()* for more information and examples.

cira_stretch

Logarithmic stretch based on a cira recipe.

reinhard_to_srgb

Stretch method based on the Reinhard algorithm, using luminance.

The function includes conversion to sRGB colorspace.

> Reinhard, Erik & Stark, Michael & Shirley, Peter & Ferwerda, James. (2002). Photographic Tone Reproduction For Digital Images. ACM Transactions on Graphics. :doi: *21. 10.1145/566654.566575*

lookup

colorize

The colorize enhancement can be used to map scaled/calibrated physical values to colors. One or several standard Trollimage color maps may be used as in the example here:

```
- name: colorize
  method: !!python/name:satpy.enhancements.colorize
  kwargs:
      palettes:
        - {colors: spectral, min_value: 193.15, max_value: 253.149999}
        - {colors: greys, min_value: 253.15, max_value: 303.15}
```

It is also possible to provide your own custom defined color mapping by specifying a list of RGB values and the corresponding min and max values between which to apply the colors. This is for instance a common use case for Sea Surface Temperature (SST) imagery, as in this example with the EUMETSAT Ocean and Sea Ice SAF (OSISAF) GHRSST product:

```
- name: osisaf_sst
  method: !!python/name:satpy.enhancements.colorize
  kwargs:
      palettes:
        - colors: [
          [255, 0, 255],
          [195, 0, 129],
          [129, 0, 47],
          [195, 0, 0],
```

(continues on next page)

(continued from previous page)

```
          [255, 0, 0],
          [236, 43, 0],
          [217, 86, 0],
          [200, 128, 0],
          [211, 154, 13],
          [222, 180, 26],
          [233, 206, 39],
          [244, 232, 52],
          [255.99609375, 255.99609375, 63.22265625],
          [203.125, 255.99609375, 52.734375],
          [136.71875, 255.99609375, 27.34375],
          [0, 255.99609375, 0],
          [0, 207.47265625, 0],
          [0, 158.94921875, 0],
          [0, 110.42578125, 0],
          [0, 82.8203125, 63.99609375],
          [0, 55.21484375, 127.9921875],
          [0, 27.609375, 191.98828125],
          [0, 0, 255.99609375],
          [100.390625, 100.390625, 255.99609375],
          [150.5859375, 150.5859375, 255.99609375]]
          min_value: 296.55
          max_value: 273.55
```

The RGB color values will be interpolated to give a smooth result. This is contrary to using the palettize enhancement.

If the source dataset already defines a palette, this can be applied directly. This requires that the palette is listed as an auxiliary variable and loaded as such by the reader. To apply such a palette directly, pass the `dataset` keyword. For example:

```
- name: colorize
  method: !!python/name:satpy.enhancements.colorize
  kwargs:
    palettes:
      - dataset: ctth_alti_pal
        color_scale: 255
```

Warning: If the source data have a valid range defined, one should **not** define `min_value` and `max_value` in the enhancement configuration! If those are defined and differ from the values in the valid range, the colors will be wrong.

The above examples are just three different ways to apply colors to images with Satpy. There is a wealth of other options for how to declare a colormap, please see *create_colormap()* for more inspiration.

palettize

three_d_effect

The *three_d_effect* enhancement adds an 3D look to an image by convolving with a 3x3 kernel. User can adjust the strength of the effect by determining the weight (default: 1.0). Example:

```
- name: 3d_effect
  method: !!python/name:satpy.enhancements.three_d_effect
  kwargs:
    weight: 1.0
```

btemp_threshold

2.12 Writing

Satpy makes it possible to save datasets in multiple formats, with *writers* designed to save in a given format. For details on additional arguments and features available for a specific Writer see the table below. Most use cases will want to save datasets using the `save_datasets()` method:

```
>>> scn.save_datasets(writer="simple_image")
```

The `writer` parameter defaults to using the `geotiff` writer. One common parameter across almost all Writers is `filename` and `base_dir` to help automate saving files with custom filenames:

```
>>> scn.save_datasets(
...     filename="{name}_{start_time:%Y%m%d_%H%M%S}.tif",
...     base_dir="/tmp/my_ouput_dir")
```

Changed in version 0.10: The *file_pattern* keyword argument was renamed to *filename* to match the *save_dataset* method"s keyword argument.

Table 4: Satpy Writers

Description	Writer name	Status	Examples
GeoTIFF	`geotiff`	Nominal	
Simple Image (PNG, JPEG, etc)	`simple_image`	Nominal	
NinJo TIFF (using `pyninjotiff` package)	`ninjotiff`	Deprecated from NinJo 7 (use ninjo-geotiff)	
NetCDF (Standard CF)	`cf`	Beta	`Usage example`
AWIPS II Tiled NetCDF4	`awips_tiled`	Beta	
GeoTIFF with NinJo tags (from NinJo 7)	`ninjogeotiff`	Beta	

2.12.1 Available Writers

To get a list of available writers use the *available_writers* function:

```
>>> from satpy import available_writers
>>> available_writers()
```

2.12.2 Colorizing and Palettizing using user-supplied colormaps

Note: In the future this functionality will be added to the `Scene` object.

It is possible to create single channel "composites" that are then colorized using users' own colormaps. The colormaps are Numpy arrays with shape (num, 3), see the example below how to create the mapping file(s).

This example creates a 2-color colormap, and we interpolate the colors between the defined temperature ranges. Beyond those limits the image clipped to the specified colors.

```
>>> import numpy as np
>>> from satpy.composites import BWCompositor
>>> from satpy.enhancements import colorize
>>> from satpy.writers import to_image
>>> arr = np.array([[0, 0, 0], [255, 255, 255]])
>>> np.save("/tmp/binary_colormap.npy", arr)
>>> compositor = BWCompositor("test", standard_name="colorized_ir_clouds")
>>> composite = compositor((local_scene[10.8], ))
>>> img = to_image(composite)
>>> kwargs = {"palettes": [{"filename": "/tmp/binary_colormap.npy",
...           "min_value": 223.15, "max_value": 303.15}]}
>>> colorize(img, **kwargs)
>>> img.show()
```

Similarly it is possible to use discrete values without color interpolation using *palettize()* instead of *colorize()*.

You can define several colormaps and ranges in the *palettes* list and they are merged together. See trollimage documentation for more information how colormaps and color ranges are merged.

The above example can be used in enhancements YAML config like this:

```
hot_or_cold:
  standard_name: hot_or_cold
  operations:
    - name: colorize
      method: &colorizefun !!python/name:satpy.enhancements.colorize ''
      kwargs:
        palettes:
          - {filename: /tmp/binary_colormap.npy, min_value: 223.15, max_value: 303.15}
```

2.12.3 Saving multiple Scenes in one go

As mentioned earlier, it is possible to save *Scene* datasets directly using `save_datasets()` method. However, sometimes it is beneficial to collect more *Scene*s together and process and save them all at once.

```
>>> from satpy.writers import compute_writer_results
>>> res1 = scn.save_datasets(filename="/tmp/{name}.png",
...                          writer="simple_image",
...                          compute=False)
>>> res2 = scn.save_datasets(filename="/tmp/{name}.tif",
...                          writer="geotiff",
...                          compute=False)
>>> results = [res1, res2]
>>> compute_writer_results(results)
```

2.12.4 Adding text to images

Satpy, via pydecorate, can add text to images when they're being saved. To use this functionality, you must create a dictionary describing the text to be added.

```
>>> decodict = {"decorate": [{"text": {"txt": "my_text",
...                                    "align": {"top_bottom": "top", "left_right": "left
↪"},
...                                    "font": <path_to_font>,
...                                    "font_size": 48,
...                                    "line": "white",
...                                    "bg_opacity": 255,
...                                    "bg": "black",
...                                    "height": 30,
...                                     }}]}
```

Where *my_text* is the text you wish to add and *<path_to_font>* is the location of the font file you wish to use, often in */usr/share/fonts/*

This dictionary can then be passed to the `save_dataset()` or `save_datasets()` command.

```
>>> scene.save_dataset(my_dataset, writer="simple_image", fill_value=False,
...                    decorate=decodict)
```

2.13 MultiScene (Experimental)

Scene objects in Satpy are meant to represent a single geographic region at a specific single instant in time or range of time. This means they are not suited for handling multiple orbits of polar-orbiting satellite data, multiple time steps of geostationary satellite data, or other special data cases. To handle these cases Satpy provides the *MultiScene* class. The below examples will walk through some basic use cases of the MultiScene.

> **Warning:** These features are still early in development and may change overtime as more user feedback is received and more features added.

2.13.1 MultiScene Creation

There are two ways to create a `MultiScene`. Either by manually creating and providing the scene objects,

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> scenes = [
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_1/*t180*.h5')),
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_2/*t180*.h5'))
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['I04'])
```

or by using the `MultiScene.from_files` class method to create a `MultiScene` from a series of files. This uses the *`group_files()`* utility function to group files by start time or other filenames parameters.

```
>>> from satpy import MultiScene
>>> from glob import glob
>>> mscn = MultiScene.from_files(glob('/data/abi/day_1/*C0[12]*.nc'), reader='abi_l1b')
>>> mscn.load(['C01', 'C02'])
```

New in version 0.12: The `from_files` and `group_files` functions were added in Satpy 0.12. See below for an alternative solution.

For older versions of Satpy we can manually create the *Scene* objects used. The `glob()` function and for loops are used to group files into Scene objects that, if used individually, could load the data we want. The code below is equivalent to the `from_files` code above:

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> scene_files = []
>>> for time_step in ['1800', '1810', '1820', '1830']:
...     scene_files.append(glob('/data/abi/day_1/*C0[12]*s???????{}*.nc'.format(time_
↪step)))
>>> scenes = [
...     Scene(reader='abi_l1b', filenames=files) for files in sorted(scene_files)
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['C01', 'C02'])
```

2.13.2 Blending Scenes in MultiScene

Scenes contained in a MultiScene can be combined in different ways.

Stacking scenes

The code below uses the `blend()` method of the `MultiScene` object to stack two separate orbits from a VIIRS sensor. By default the `blend` method will use the `stack()` function which uses the first dataset as the base of the image and then iteratively overlays the remaining datasets on top.

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> scenes = [
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_1/*t180*.h5')),
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_2/*t180*.h5'))
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['I04'])
>>> new_mscn = mscn.resample(my_area)
>>> blended_scene = new_mscn.blend()
>>> blended_scene.save_datasets()
```

Stacking scenes using weights

It is also possible to blend scenes together in a bit more sophisticated manner using pixel based weighting instead of just stacking the scenes on top of each other as described above. This can for instance be useful to make a cloud parameter (cover, height, etc) composite combining cloud parameters derived from both geostationary and polar orbiting satellite data close in time and over a given area. This is useful for instance at high latitudes where geostationary data degrade quickly with latitude and polar data are more frequent.

This weighted blending can be accomplished via the use of the builtin `partial()` function (see Partial) and the default `stack()` function. The `stack()` function can take the optional argument *weights* (*None* on default) which should be a sequence (of length equal to the number of scenes being blended) of arrays with pixel weights.

The code below gives an example of how two cloud scenes can be blended using the satellite zenith angles to weight which pixels to take from each of the two scenes. The idea being that the reliability of the cloud parameter is higher when the satellite zenith angle is small.

```
>>> from satpy import Scene, MultiScene,  DataQuery
>>> from functools import partial
>>> from satpy.resample import get_area_def
>>> areaid = get_area_def("myarea")
>>> geo_scene = Scene(filenames=glob('/data/to/nwcsaf/geo/files/*nc'), reader='nwcsaf-geo
↪')
>>> geo_scene.load(['ct'])
>>> polar_scene = Scene(filenames=glob('/data/to/nwcsaf/pps/noaa18/files/*nc'), reader=
↪'nwcsaf-pps_nc')
>>> polar_scene.load(['cma', 'ct'])
>>> mscn = MultiScene([geo_scene, polar_scene])
>>> groups = {DataQuery(name='CTY_group'): ['ct']}
```

(continues on next page)

(continued from previous page)

```
>>> mscn.group(groups)
>>> resampled = mscn.resample(areaid, reduce_data=False)
>>> weights = [1./geo_satz, 1./n18_satz]
>>> stack_with_weights = partial(stack, weights=weights)
>>> blended = resampled.blend(blend_function=stack_with_weights)
>>> blended_scene.save_dataset('CTY_group', filename='./blended_stack_weighted_geo_polar.
↪nc')
```

Grouping Similar Datasets

By default, `MultiScene` only operates on datasets shared by all scenes. Use the `group()` method to specify groups of datasets that shall be treated equally by `MultiScene`, even if their names or wavelengths are different.

Example: Stacking scenes from multiple geostationary satellites acquired at roughly the same time. First, create scenes and load datasets individually:

```
>>> from satpy import Scene
>>> from glob import glob
>>> h8_scene = satpy.Scene(filenames=glob('/data/HS_H08_20200101_1200*'),
...                        reader='ahi_hsd')
>>> h8_scene.load(['B13'])
>>> g16_scene = satpy.Scene(filenames=glob('/data/OR_ABI*s20200011200*.nc'),
...                         reader='abi_l1b')
>>> g16_scene.load(['C13'])
>>> met10_scene = satpy.Scene(filenames=glob('/data/H-000-MSG4*-202001011200-__'),
...                           reader='seviri_l1b_hrit')
>>> met10_scene.load(['IR_108'])
```

Now create a `MultiScene` and group the three similar IR channels together:

```
>>> from satpy import MultiScene, DataQuery
>>> mscn = MultiScene([h8_scene, g16_scene, met10_scene])
>>> groups = {DataQuery('IR_group', wavelength=(10, 11, 12)): ['B13', 'C13', 'IR_108']}
>>> mscn.group(groups)
```

Finally, resample the datasets to a common grid and blend them together:

```
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> resampled = mscn.resample(my_area, reduce_data=False)
>>> blended = resampled.blend()  # you can also use a custom blend function
```

You can access the results via `blended['IR_group']`.

Timeseries

Using the `blend()` method with the `timeseries()` function will combine multiple scenes from different time slots by time. A single *Scene* with each dataset/channel extended by the time dimension will be returned. If used together with the *`to_geoviews()`* method, creation of interactive timeseries Bokeh plots is possible.

```
>>> from satpy import Scene, MultiScene
>>> from satpy.multiscene import timeseries
>>> from glob import glob
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> scenes = [
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_1/*t180*.h5')),
...     Scene(reader='viirs_sdr', filenames=glob('/data/viirs/day_2/*t180*.h5'))
... ]
>>> mscn = MultiScene(scenes)
>>> mscn.load(['I04'])
>>> new_mscn = mscn.resample(my_area)
>>> blended_scene = new_mscn.blend(blend_function=timeseries)
>>> blended_scene['I04']
<xarray.DataArray (time: 2, y: 1536, x: 6400)>
dask.array<shape=(2, 1536, 6400), dtype=float64, chunksize=(1, 1536, 4096)>
Coordinates:
  * time     (time) datetime64[ns] 2012-02-25T18:01:24.570942 2012-02-25T18:02:49.975797
Dimensions without coordinates: y, x
```

2.13.3 Saving frames of an animation

The MultiScene can take "frames" of data and join them together in a single animation movie file. Saving animations requires the *imageio* python library and for most available formats the `ffmpeg` command line tool suite should also be installed. The below example saves a series of GOES-EAST ABI channel 1 and channel 2 frames to MP4 movie files.

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> mscn = MultiScene.from_files(glob('/data/abi/day_1/*C0[12]*.nc'), reader='abi_l1b')
>>> mscn.load(['C01', 'C02'])
>>> mscn.save_animation('{name}_{start_time:%Y%m%d_%H%M%S}.mp4', fps=2)
```

This will compute one video frame (image) at a time and write it to the MPEG-4 video file. For users with more powerful systems it is possible to use the `client` and `batch_size` keyword arguments to compute multiple frames in parallel using the dask `distributed` library (if installed). See the dask distributed documentation for information on creating a `Client` object. If working on a cluster you may want to use dask jobqueue to take advantage of multiple nodes at a time.

It is possible to add an overlay or decoration to each frame of an animation. For text added as a decoration, string substitution will be applied based on the attributes of the dataset, for example:

```
>>> mscn.save_animation(
...     "{name:s}_{start_time:%Y%m%d_%H%M}.mp4",
...     enh_args={
...     "decorate": {
...         "decorate": [
...             {"text": {
```

(continues on next page)

(continued from previous page)

```
...                     "txt": "time {start_time:%Y-%m-%d %H:%M}",
...                     "align": {
...                         "top_bottom": "bottom",
...                         "left_right": "right"},
...                     "font": '/usr/share/fonts/truetype/arial.ttf',
...                     "font_size": 20,
...                     "height": 30,
...                     "bg": "black",
...                     "bg_opacity": 255,
...                     "line": "white"}}]}})
```

If your file covers ABI MESO data for an hour for channel 2 lasting from 2020-04-12 01:00-01:59, then the output file will be called `C02_20200412_0100.mp4` (because the first dataset/frame corresponds to an image that started to be taken at 01:00), consist of sixty frames (one per minute for MESO data), and each frame will have the start time for that frame floored to the minute blended into the frame. Note that this text is "burned" into the video and cannot be switched on or off later.

> **Warning:** GIF images, although supported, are not recommended due to the large file sizes that can be produced from only a few frames.

2.13.4 Saving multiple scenes

The `MultiScene` object includes a `save_datasets()` method for saving the data from multiple Scenes to disk. By default this will operate on one Scene at a time, but similar to the `save_animation` method above this method can accept a dask distributed `Client` object via the `client` keyword argument to compute scenes in parallel (see documentation above). Note however that some writers, like the `geotiff` writer, do not support multi-process operations at this time and will fail when used with dask distributed. To save multiple Scenes use:

```
>>> from satpy import Scene, MultiScene
>>> from glob import glob
>>> mscn = MultiScene.from_files(glob('/data/abi/day_1/*C0[12]*.nc'), reader='abi_l1b')
>>> mscn.load(['C01', 'C02'])
>>> mscn.save_datasets(base_dir='/path/for/output')
```

2.13.5 Combining multiple readers

New in version 0.23.

The `from_files()` constructor allows to automatically combine multiple readers into a single MultiScene. It is no longer necessary for the user to create the *Scene* objects themselves. For example, you can combine Advanced Baseline Imager (ABI) and Global Lightning Mapper (GLM) measurements. Constructing a multi-reader MultiScene requires more parameters than a single-reader MultiScene, because Satpy can poorly guess how to group files belonging to different instruments. For an example creating a video with lightning superimposed on ABI channel 14 (11.2 µm) using the built-in composite `C14_flash_extent_density`, which superimposes flash extent density from GLM (read with the *NCGriddedGLML2* or `glm_l2` reader) on ABI channel 14 data (read with the *NC_ABI_L1B* or `abi_l1b` reader), and therefore needs Scene objects that combine both readers:

```
>>> glm_dir = "/path/to/GLMC/"
>>> abi_dir = "/path/to/ABI/"
```

(continues on next page)

(continued from previous page)

```
>>> ms = satpy.MultiScene.from_files(
...         glob.glob(glm_dir + "OR_GLM-L2-GLMC-M3_G16_s202010418*.nc") +
...         glob.glob(abi_dir + "C*/OR_ABI-L1b-RadC-M6C*_G16_s202010418*_e*_c*.nc"),
...         reader=["glm_l2", "abi_l1b"],
...         ensure_all_readers=True,
...         group_keys=["start_time"],
...         time_threshold=30)
>>> ms.load(["C14_flash_extent_density"])
>>> ms = ms.resample(ms.first_scene["C14"].attrs["area"])
>>> ms.save_animation("/path/for/output/{name:s}_{start_time:%Y%m%d_%H%M}.mp4")
```

In this example, we pass to `from_files()` the additional parameters `ensure_all_readers=True`, `group_keys=["start_time"]`, `time_threshold=30` so we only get scenes at times that both ABI and GLM have a file starting within 30 seconds from each other, and ignore all other differences for the purposes of grouping the two. For this example, the ABI files occur every 5 minutes but the GLM files (processed with glmtools) every minute. Scenes where there is a GLM file without an ABI file starting within at most ±30 seconds are skipped. The `group_keys` and `time_threshold` keyword arguments are processed by the *`group_files()`* function. The heavy work of blending the two instruments together is performed by the *`BackgroundCompositor`* class through the *"C14_flash_extent_density"* composite.

2.14 Developer's Guide

The below sections will walk through how to set up a development environment, make changes to the code, and test that they work. See the *How to contribute* section for more information on getting started and contributor expectations. Additional information for developer's can be found at the pages listed below.

2.14.1 How to contribute

Thank you for considering contributing to Satpy! Satpy's development team is made up of volunteers so any help we can get is very appreciated.

Contributions from users are what keep this community going. We welcome any contributions including bug reports, documentation fixes or updates, bug fixes, and feature requests. By contributing to Satpy you are providing code that everyone can use and benefit from.

The following guidelines will describe how the Satpy project structures its code contributions from discussion to code to package release.

For more information on contributing to open source projects see GitHub's Guide.

What can I do?

- Make sure you have a GitHub account.
- Submit a ticket for your issue, assuming one does not already exist.
- If you're uncomfortable using Git/GitHub, see Learn Git Branching or other online tutorials.
- If you are uncomfortable contributing to an open source project see:
  - How to Contribute to an Open Source Project on GitHub video series
  - Aaron Meurer's Git Workflow

  - How to Contribute to Open Source
- See what issues already exist. Issues marked good first issue or help wanted can be good issues to start with.
- Read the *Developer's Guide* for more details on contributing code.
- Fork the repository on GitHub and install the package in development mode.
- Update the Satpy documentation to make it clearer and more detailed.
- Contribute code to either fix a bug or add functionality and submit a Pull Request.
- Make an example Jupyter Notebook and add it to the available examples.

What if I break something?

Not possible. If something breaks because of your contribution it was our fault. When you submit your changes to be merged as a GitHub Pull Request they will be automatically tested and checked against coding style rules. Before they are merged they are reviewed by at least one maintainer of the Satpy project. If anything needs updating, we'll let you know.

What is expected?

You can expect the Satpy maintainers to help you. We are all volunteers, have jobs, and occasionally go on vacations. We will try our best to answer your questions as soon as possible. We will try our best to understand your use case and add the features you need. Although we strive to make Satpy useful for everyone there may be some feature requests that we can't allow if they would require breaking existing features. Other features may be best for a different package, PyTroll or otherwise. Regardless, we will help you find the best place for your feature and to make it possible to do what you want.

We, the Satpy maintainers, expect you to be patient, understanding, and respectful of both developers and users. Satpy can only be successful if everyone in the community feels welcome. We also expect you to put in as much work as you expect out of us. There is no dedicated PyTroll or Satpy support team, so there may be times when you need to do most of the work to solve your problem (trying different test cases, environments, etc).

Being respectful includes following the style of the existing code for any code submissions. Please follow PEP8 style guidelines and limit lines of code to 80 characters whenever possible and when it doesn't hurt readability. Satpy follows Google Style Docstrings for all code API documentation. When in doubt use the existing code as a guide for how coding should be done.

How do I get help?

The Satpy developers (and all other PyTroll package developers) monitor the:

- Mailing List
- Slack chat (get an invitation)
- GitHub issues

How do I submit my changes?

Any contributions should start with some form of communication (see above) to let the Satpy maintainers know how you plan to help. The larger the contribution the more important direct communication is so everyone can avoid duplicate code and wasted time. After talking to the Satpy developers any additional work like code or documentation changes can be provided as a GitHub Pull Request.

To make sure that your code complies with the pytroll python standard, you can run the flake8 linter on your changes before you submit them, or even better install a pre-commit hook that runs the style check for you. To this aim, we provide a configuration file for the pre-commit tool, that you can install with eg:

```
pip install pre-commit
pre-commit install
```

running from your base satpy directory. This will automatically check code style for every commit.

Code of Conduct

Satpy follows the same code of conduct as the PyTroll project. For reference it is copied to this repository in CODE_OF_CONDUCT.md.

As stated in the PyTroll home page, this code of conduct applies to the project space (GitHub) as well as the public space online and offline when an individual is representing the project or the community. Online examples of this include the PyTroll Slack team, mailing list, and the PyTroll twitter account. This code of conduct also applies to in-person situations like PyTroll Contributor Weeks (PCW), conference meet-ups, or any other time when the project is being represented.

Any violations of this code of conduct will be handled by the core maintainers of the project including David Hoese, Martin Raspaud, and Adam Dybbroe. If you wish to report one of the maintainers for a violation and are not comfortable with them seeing it, please contact one or more of the other maintainers to report the violation. Responses to violations will be determined by the maintainers and may include one or more of the following:

- Verbal warning
- Ask for public apology
- Temporary or permanent ban from in-person events
- Temporary or permanent ban from online communication (Slack, mailing list, etc)

For details see the official code of conduct document.

2.14.2 Migrating to xarray and dask

Many python developers dealing with meteorologic satellite data begin with using NumPy arrays directly. This work usually involves masked arrays, boolean masks, index arrays, and reshaping. Due to the libraries used by Satpy these operations can't always be done in the same way. This guide acts as a starting point for new Satpy developers in transitioning from NumPy's array operations to Satpy's operations, although they are very similar.

To provide the most functionality for users, Satpy uses the xarray library's `DataArray` object as the main representation for its data. DataArray objects can also benefit from the dask library. The combination of these libraries allow Satpy to easily distribute operations over multiple workers, lazy evaluate operations, and keep track additional metadata and coordinate information.

XArray

```
import xarray as xr
```

XArray's DataArray is now the standard data structure for arrays in satpy. They allow the array to define dimensions, coordinates, and attributes (that we use for metadata).

To create such an array, you can do for example

```
my_dataarray = xr.DataArray(my_data, dims=['y', 'x'],
                            coords={'x': np.arange(...)},
                            attrs={'sensor': 'olci'})
```

where `my_data` can be a regular numpy array, a numpy memmap, or, if you want to keep things lazy, a dask array (more on dask later). Satpy uses dask arrays with all of its DataArrays.

Dimensions

In satpy, the dimensions of the arrays should include:

- *x* for the x or column or pixel dimension
- *y* for the y or row or line dimension
- *bands* for composites
- *time* can also be provided, but we have limited support for it at the moment. Use metadata for common cases (*start_time*, *end_time*)

Dimensions are accessible through `my_dataarray.dims`. To get the size of a given dimension, use `sizes`:

```
my_dataarray.sizes['x']
```

Coordinates

Coordinates can be defined for those dimensions when it makes sense:

- *x* and *y*: Usually defined when the data's area is an `AreaDefinition`, and they contain the projection coordinates in x and y.
- *bands*: Contain the letter of the color they represent, eg `['R', 'G', 'B']` for an RGB composite.

This allows then to select for example a single band like this:

```
red = my_composite.sel(bands='R')
```

or even multiple bands:

```
red_and_blue = my_composite.sel(bands=['R', 'B'])
```

To access the coordinates of the data array, use the following syntax:

```
x_coords = my_dataarray['x']
my_dataarray['y'] = np.arange(...)
```

Most of the time, satpy will fill the coordinates for you, so you just need to provide the dimension names.

Attributes

To save metadata, we use the `attrs` dictionary.

```
my_dataarray.attrs['platform_name'] = 'Sentinel-3A'
```

Some metadata that should always be present in our dataarrays:

- `area` the area of the dataset. This should be handled in the reader.
- `start_time`, `end_time`
- `sensor`

Operations on DataArrays

DataArrays work with regular arithmetic operation as one would expect of eg numpy arrays, with the exception that using an operator on two DataArrays requires both arrays to share the same dimensions, and coordinates if those are defined.

For mathematical functions like cos or log, you can use numpy functions directly and they will return a DataArray object:

```
import numpy as np
cos_zen = np.cos(zen_xarray)
```

Masking data

In DataArrays, masked data is represented with NaN values. Hence the default type is `float64`, but `float32` works also in this case. XArray can't handle masked data for integer data, but in satpy we try to use the special `_FillValue` attribute (in `.attrs`) to handle this case. If you come across a case where this isn't handled properly, contact us.

Masking data from a condition can be done with:

```
result = my_dataarray.where(my_dataarray > 5)
```

Result is then analogous to my_dataarray, with values lower or equal to 5 replaced by NaNs.

Further reading

http://xarray.pydata.org/en/stable/generated/xarray.DataArray.html#xarray.DataArray

Dask

```
import dask.array as da
```

The data part of the DataArrays we use in satpy are mostly dask Arrays. That allows lazy and chunked operations for efficient processing.

Creation

From a numpy array

To create a dask array from a numpy array, one can call the `from_array()` function:

```
darr = da.from_array(my_numpy_array, chunks=4096)
```

The *chunks* keyword tells dask the size of a chunk of data. If the numpy array is 3-dimensional, the chunk size provide above means that one chunk will be 4096x4096x4096 elements. To prevent this, one can provide a tuple:

```
darr = da.from_array(my_numpy_array, chunks=(4096, 1024, 2))
```

meaning a chunk will be 4096x1024x2 elements in size.

Even more detailed sizes for the chunks can be provided if needed, see the dask documentation.

From memmaps or other lazy objects

To avoid loading the data into memory when creating a dask array, other kinds of arrays can be passed to `from_array()`. For example, a numpy memmap allows dask to know where the data is, and will only be loaded when the actual values need to be computed. Another example is a hdf5 variable read with h5py.

Procedural generation of data

Some procedural generation function are available in dask, eg `meshgrid()`, `arange()`, or `random.random`.

From XArray to Dask and back

Certain operations are easiest to perform on dask arrays by themselves, especially when certain functions are only available from the dask library. In these cases you can operate on the dask array beneath the DataArray and create a new DataArray when done. Note dask arrays do not support in-place operations. In-place operations on xarray DataArrays will reassign the dask array automatically.

```
dask_arr = my_dataarray.data
dask_arr = dask_arr + 1
# ... other non-xarray operations ...
new_dataarr = xr.DataArray(dask_arr, dims=my_dataarray.dims, attrs=my_dataarray.attrs.
↪copy())
```

Or if the operation should be assigned back to the original DataArray (if and only if the data is the same size):

```
my_dataarray.data = dask_arr
```

Operations and how to get actual results

Regular arithmetic operations are provided, and generate another dask array.

```
>>> arr1 = da.random.uniform(0, 1000, size=(1000, 1000), chunks=100)
>>> arr2 = da.random.uniform(0, 1000, size=(1000, 1000), chunks=100)
>>> arr1 + arr2
dask.array<add, shape=(1000, 1000), dtype=float64, chunksize=(100, 100)>
```

In order to compute the actual data during testing, use the `compute()` method. In normal Satpy operations you will want the data to be evaluated as late as possible to improve performance so *compute* should only be used when needed.

```
>>> (arr1 + arr2).compute()
array([[  898.08811639,  1236.96107629,  1154.40255292, ...,
         1537.50752674,  1563.89278664,   433.92598566],
       [ 1657.43843608,  1063.82390257,  1265.08687916, ...,
         1103.90421234,  1721.73564104,  1276.5424228 ],
       [ 1620.11393216,   212.45816261,   771.99348555, ...,
         1675.6561068 ,   585.89123159,   935.04366354],
       ...,
       [ 1533.93265862,  1103.33725432,   191.30794159, ...,
          520.00434673,   426.49238283,  1090.61323471],
       [  816.6108554 ,  1526.36292498,   412.91953023, ...,
          982.71285721,   699.087645  ,  1511.67447362],
       [ 1354.6127365 ,  1671.24591983,  1144.64848757, ...,
         1247.37586051,  1656.50487092,   978.28184726]])
```

Dask also provides *cos*, *log* and other mathematical function, that you can use with `da.cos` and `da.log`. However, since satpy uses xarrays as standard data structure, prefer the xarray functions when possible (they call in turn the dask counterparts when possible).

Wrapping non-dask friendly functions

Some operations are not supported by dask yet or are difficult to convert to take full advantage of dask's multithreaded operations. In these cases you can wrap a function to run on an entire dask array when it is being computed and pass on the result. Note that this requires fully computing all of the dask inputs to the function and are passed as a numpy array or in the case of an XArray DataArray they will be a DataArray with a numpy array underneath. You should *NOT* use dask functions inside the delayed function.

```
import dask
import dask.array as da

def _complex_operation(my_arr1, my_arr2):
    return my_arr1 + my_arr2

delayed_result = dask.delayed(_complex_operation)(my_dask_arr1, my_dask_arr2)
# to create a dask array to use in the future
my_new_arr = da.from_delayed(delayed_result, dtype=my_dask_arr1.dtype, shape=my_dask_
↪arr1.shape)
```

Dask Delayed objects can also be computed `delayed_result.compute()` if the array is not needed or if the function doesn't return an array.

http://dask.pydata.org/en/latest/array-api.html#dask.array.from_delayed

Map dask blocks to non-dask friendly functions

If the complicated operation you need to perform can be vectorized and does not need the entire data array to do its operations you can use `da.map_blocks` to get better performance than creating a delayed function. Similar to delayed functions the inputs to the function are fully computed DataArrays or numpy arrays, but only the individual chunks of the dask array at a time. Note that `map_blocks` must be provided dask arrays and won't function properly on XArray DataArrays. It is recommended that the function object passed to `map_blocks` **not** be an internal function (a function defined inside another function) or it may be unserializable and can cause issues in some environments.

```
my_new_arr = da.map_blocks(_complex_operation, my_dask_arr1, my_dask_arr2, dtype=my_dask_
↪arr1.dtype)
```

Helpful functions

- `map_blocks()`
- `map_overlap()`
- `atop()`
- `store()`
- `tokenize()`
- `compute()`
- Dask Delayed
- `rechunk()`
- `vindex`

2.14.3 Adding a Custom Reader to Satpy

In order to add a reader to satpy, you will need to create two files:

- a YAML file for describing the files to read and the datasets that are available
- a python file implementing the actual reading of the datasets and metadata

Satpy implements readers by defining a single "reader" object that pulls information from one or more file handler objects. The base reader class provided by Satpy is enough for most cases and does not need to be modified. The individual file handler classes do need to be created due to the small differences between file formats.

The below documentation will walk through each part of making a reader in detail. To do this we will implement a reader for the EUMETSAT NetCDF format for SEVIRI data.

Naming your reader

Satpy tries to follow a standard scheme for naming its readers. These names are used in filenames, but are also used by users so it is important that the name be recognizable and clear. Although some special cases exist, most fit in to the following naming scheme:

```
<sensor>[_<processing level>[_<level detail>]][_<file format>]
```

All components of the name should be lowercase and use underscores as the main separator between fields. Hyphens should be used as an intra-field separator if needed (ex. goes-imager).

sensor
The first component of the name represents the sensor or instrument that observed the data stored in the files being read. If the files are the output of a specific processing software or a certain algorithm implementation that supports multiple sensors then a lowercase version of that software's name should be used (e.g. clavrx for CLAVR-x, nucaps for NUCAPS). The `sensor` field is the only required field of the naming scheme. If it is actually an instrument name then the reader name should include one of the other optional fields. If sensor is a software package then that may be enough without any additional information to uniquely identify the reader.

processing level
This field marks the specific level of processing or calibration that has been performed to produce the data in the files being read. Common values of this field include: `sdr` for Sensor Data Record (SDR), `edr` for Environmental Data Record (EDR), `l1b` for Level 1B, and `l2` for Level 2.

level detail
In cases where the processing level is not enough to completely define the reader this field can be used to provide a little more context. For example, some VIIRS EDR products are specific to a particular field of study or type of scientific event, like a flood or cloud product. In these cases the detail field can be added to produce a name like `viirs_edr_flood`. This field shouldn't be used unless processing level is also specified.

file format
If the file format of the files is informative to the user or can distinguish one reader from another then this field should be specified. Common format names should be abbreviated following existing abbreviations like `nc` for NetCDF3 or NetCDF4, `hdf` for HDF4, `h5` for HDF5.

The existing *reader's table* can be used for reference. When in doubt, reader names can be discussed in the GitHub pull request when this reader is added to Satpy, or in a GitHub issue.

The YAML file

If your reader is going to be part of Satpy, the YAML file should be located in the `satpy/etc/readers` directory, along with the YAML files for all other readers. If you are developing a reader for internal purposes (such as for unpublished data), the YAML file should be located in any directory in `$SATPY_CONFIG_PATH` within the subdirectory `readers/` (see *Configuration*).

The YAML file is composed of three sections:

- the *reader* section, that provides basic parameters for the reader
- the *file_types* section, that gives the patterns of the files this reader can handle
- the *datasets* section, that describes the datasets available from this reader

The `reader` section

The `reader` section provides basic parameters for the overall reader.

The parameters to provide in this section are:

name
This is the name of the reader, it should be the same as the filename (without the .yaml extension). The naming convention for this is described above in the *Naming your reader* section above. short_name (optional): Human-readable version of the reader 'name'. If not provided, applications using this can default to taking the 'name', replacing _ with spaces and uppercasing every letter.

long_name
Human-readable title for the reader. This may be used as a section title on a website or in GUI applications using Satpy. Default naming scheme is `<space program> <sensor> Level <level> [<format>]`. For example, for the `abi_l1b` reader this is `"GOES-R ABI Level 1b"` where "GOES-R" is the name of the program and **not** the name of the platform/satellite. This scheme may not work for all readers, but in general should be followed. See existing readers for more examples.

description
General description of the reader. This may include any restructuredtext formatted text like links to PDFs or sites with more information on the file format. This can be multiline if formatted properly in YAML (see example below).

status
The status of the reader (one of: Nominal, Beta, Alpha)

supports_fsspec
If the reader supports reading data via fsspec (either true or false).

sensors
The list of sensors this reader will support. This must be all lowercase letters for full support throughout in Satpy.

reader
The main python reader class to use, in most cases the `FileYAMLReader` is a good choice.

```yaml
reader:
  name: seviri_l1b_nc
  short_name: SEVIRI L1b NetCDF4
  long_name: MSG SEVIRI Level 1b (NetCDF4)
  description: >
    NetCDF4 reader for EUMETSAT MSG SEVIRI Level 1b files.
  sensors: [seviri]
  reader: !!python/name:satpy.readers.yaml_reader.FileYAMLReader
```

Optionally, if you need to customize the `DataID` for this reader, you can provide the relevant keys with a `data_identification_keys` item here. See the *Satpy internal workings: having a look under the hood* section for more information.

The `file_types` section

Each file type needs to provide:

- `file_reader`, the class that will handle the files for this reader, that you will implement in the corresponding python file. See the *The python file* section below.
- `file_patterns`, the patterns to match to find files this reader can handle. The syntax to use is basically the same as `format` with the addition of time. See the trollsift package documentation for more details.
- Optionally, a file type can have a `requires` field: it is a list of file types that the current file types needs to function. For example, the HRIT MSG format segment files each need a prologue and epilogue file to be read properly, hence in this case we have added `requires: [HRIT_PRO, HRIT_EPI]` to the file type definition.

```yaml
file_types:
    nc_seviri_l1b:
        file_reader: !!python/name:satpy.readers.nc_seviri_l1b.NCSEVIRIFileHandler
        file_patterns: ['W_XX-EUMETSAT-Darmstadt,VIS+IR+IMAGERY,{satid:4s}+SEVIRI_C_EUMG_
↪{processing_time:%Y%m%d%H%M%S}.nc']
```

(continues on next page)

(continued from previous page)

```
    nc_seviri_l1b_hrv:
        file_reader: !!python/name:satpy.readers.nc_seviri_l1b.NCSEVIRIHRVFileHandler
        file_patterns: ['W_XX-EUMETSAT-Darmstadt,HRV+IMAGERY,{satid:4s}+SEVIRI_C_EUMG_
↪{processing_time:%Y%m%d%H%M%S}.nc']
```

The `datasets` section

The datasets section describes each dataset available in the files. The parameters provided are made available to the methods of the implemented python class.

If your input files contain all the necessary metadata or you have a lot of datasets to configure look at the *Dynamic Dataset Configuration* section below. Implementing this will save you from having to write a lot of configuration in the YAML files.

Parameters you can define for example are:

- name
- sensor
- resolution
- wavelength
- polarization
- standard_name: The CF standard name for the dataset that will be used to determine the type of data. See existing readers for common standard names in Satpy or the CF standard name documentation for other available names or how to define your own. Satpy does not currently have a hard requirement on these names being completely CF compliant, but consistency across readers is important.
- units: The units of the data when returned by the file handler. Although not technically a requirement, it is common for Satpy datasets to use "%" for reflectance fields and "K" for brightness temperature fields.
- modifiers: The modification(s) that have already been applied to the data when it is returned by the file handler. Only a few of these have been standardized across Satpy, but are based on the names of the modifiers configured in the "composites" YAML files. Examples include `sunz_corrected` or `rayleigh_corrected`. See the metadata wiki for more information.
- file_type: Name of file type (see above).
- coordinates: An optional two-element list with the names of the longitude and latitude datasets describing the location of this dataset. This is optional if the data being read is gridded already. Swath data, from example data from some polar-orbiting satellites, should have these defined or no geolocation information will be available when the data are loaded. For gridded datasets a `get_area_def` function will be implemented in python (see below) to define geolocation information.
- Any other field that is relevant for the reader or could be useful metadata provided to the user.

This section can be copied and adapted simply from existing seviri readers, like for example the `msg_native` reader.

```
datasets:
  HRV:
    name: HRV
    resolution: 1000.134348869
    wavelength: [0.5, 0.7, 0.9]
    calibration:
      reflectance:
```

(continues on next page)

(continued from previous page)

```
      standard_name: toa_bidirectional_reflectance
      units: "%"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b_hrv

IR_016:
  name: IR_016
  resolution: 3000.403165817
  wavelength: [1.5, 1.64, 1.78]
  calibration:
    reflectance:
      standard_name: toa_bidirectional_reflectance
      units: "%"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b
  nc_key: 'ch3'

IR_039:
  name: IR_039
  resolution: 3000.403165817
  wavelength: [3.48, 3.92, 4.36]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b
  nc_key: 'ch4'

IR_087:
  name: IR_087
  resolution: 3000.403165817
  wavelength: [8.3, 8.7, 9.1]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
```

(continues on next page)

(continued from previous page)

```
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

IR_097:
  name: IR_097
  resolution: 3000.403165817
  wavelength: [9.38, 9.66, 9.94]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

IR_108:
  name: IR_108
  resolution: 3000.403165817
  wavelength: [9.8, 10.8, 11.8]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

IR_120:
  name: IR_120
  resolution: 3000.403165817
  wavelength: [11.0, 12.0, 13.0]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
```

(continues on next page)

(continued from previous page)

```
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

IR_134:
  name: IR_134
  resolution: 3000.403165817
  wavelength: [12.4, 13.4, 14.4]
  calibration:
    brightness_temperature:
      standard_name: toa_brightness_temperature
      units: K
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

VIS006:
  name: VIS006
  resolution: 3000.403165817
  wavelength: [0.56, 0.635, 0.71]
  calibration:
    reflectance:
      standard_name: toa_bidirectional_reflectance
      units: "%"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b

VIS008:
  name: VIS008
  resolution: 3000.403165817
  wavelength: [0.74, 0.81, 0.88]
  calibration:
    reflectance:
      standard_name: toa_bidirectional_reflectance
      units: "%"
    radiance:
      standard_name: toa_outgoing_radiance_per_unit_wavelength
      units: W m-2 um-1 sr-1
    counts:
      standard_name: counts
      units: count
  file_type: nc_seviri_l1b
```

(continues on next page)

(continued from previous page)

```yaml
  WV_062:
    name: WV_062
    resolution: 3000.403165817
    wavelength: [5.35, 6.25, 7.15]
    calibration:
      brightness_temperature:
        standard_name: toa_brightness_temperature
        units: "K"
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b

  WV_073:
    name: WV_073
    resolution: 3000.403165817
    wavelength: [6.85, 7.35, 7.85]
    calibration:
      brightness_temperature:
        standard_name: toa_brightness_temperature
        units: "K"
      radiance:
        standard_name: toa_outgoing_radiance_per_unit_wavelength
        units: W m-2 um-1 sr-1
      counts:
        standard_name: counts
        units: count
    file_type: nc_seviri_l1b
```

The YAML file is now ready and you can move on to writing your python code.

Dynamic Dataset Configuration

The above "datasets" section for reader configuration is the most explicit method for specifying metadata about possible data that can be loaded from input files. It is also the easiest way for people with little python experience to customize or add new datasets to a reader. However, some file formats may have 10s or even 100s of datasets or variations of datasets. Writing the metadata and access information for every one of these datasets can easily become a problem. To help in these cases the `available_datasets()` file handler interface can be used.

This method, if needed, should be implemented in your reader's file handler classes. The best information for what this method does and how to use it is available in the `API documentation`. This method is good when you want to:

1. Define datasets dynamically without needing to define them in the YAML.
2. Supplement metadata from the YAML file with information from the file content (ex. `resolution`).
3. Determine if a dataset is available by the file contents. This differs from the default behavior of a dataset being considered loadable if its "file_type" is loaded.

Note that this is considered an advanced interface and involves more advanced Python concepts like generators. If you need help with anything feel free to ask questions in your pull request or on the *Pytroll Slack*.

The python file

The python files needs to implement a file handler class for each file type that we want to read. Such a class needs to implement a few methods:

- the `__init__` method, that takes as arguments
  - the filename (string)
  - the filename info (dict) that we get by parsing the filename using the pattern defined in the yaml file
  - the filetype info that we get from the filetype definition in the yaml file

  This method can also receive other file handler instances as parameter if the filetype at hand has requirements. (See the explanation in the YAML file filetype section above)
- the `get_dataset` method, which takes as arguments
  - the dataset ID of the dataset to load
  - the dataset info that is the description of the channel in the YAML file

  This method has to return an xarray.DataArray instance if the loading is successful, containing the data and *metadata* of the loaded dataset, or return None if the loading was unsuccessful.

  The DataArray should at least have a `y` dimension. For data covering a 2D region on the Earth, their should be at least a `y` and `x` dimension. This applies to non-gridded data like that of a polar-orbiting satellite instrument. The latitude dimension is typically named `y` and longitude named `x`. This may require renaming dimensions from the file, see for the `xarray.DataArray.rename()` method for more information and its use in the example below.

  If the reader should be compatible with opening remote files see *Adding remote file support to a reader*.
- the `get_area_def` method, that takes as single argument the `DataID` for which we want the area. It should return a `AreaDefinition` object. For data that cannot be geolocated with an area definition, the pixel coordinates will be loaded using the `get_dataset` method for the resulting scene to be navigated. The names of the datasets to be loaded should be specified as a special `coordinates` attribute in the YAML file. For example, by specifying `coordinates: [longitude_dataset, latitude_dataset]` in the YAML, Satpy will call `get_dataset` twice, once to load the dataset named `longitude_dataset` and once to load `latitude_dataset`. Satpy will then create a `SwathDefinition` with this coordinate information and assign it to the dataset's `.attrs['area']` attribute.
- Optionally, the `get_bounding_box` method can be implemented if filtering files by area is desirable for this data type

On top of that, two attributes need to be defined: `start_time` and `end_time`, that define the start and end times of the sensing. See the *Time Metadata* section for a description of the different times that Satpy readers typically use and what times should be used for the `start_time` and `end_time`. Note that these properties will be assigned to the `start_time` and `end_time` metadata of any DataArrays returned by `get_dataset`, any existing values will be overwritten.

If you are writing a file handler for more common formats like HDF4, HDF5, or NetCDF4 you may want to consider using the utility base classes for each: `satpy.readers.hdf4_utils.HDF4FileHandler`, `satpy.readers.hdf5_utils.HDF5FileHandler`, and `satpy.readers.netcdf_utils.NetCDF4FileHandler`. These were added as a convenience and are not required to read these formats. In many cases using the `xarray.open_dataset()` function in a custom file handler is a much better idea.

Note: Be careful about the data types of the DataArray attributes (*.attrs*) your reader is returning. Satpy or other tools may attempt to serialize these attributes (ex. hashing for cache keys). For example, Numpy types don't serialize into JSON and should therefore be cast to basic Python types (*float*, *int*, etc) before being assigned to the attributes.

Note: Be careful about the types of the data your reader is returning. It is easy to let the data be coerced into double precision floats (*np.float64*). At the moment, satellite instruments are rarely measuring in a resolution greater than what can be encoded in 16 bits. As such, to preserve processing power, please consider carefully what data type you should scale or calibrate your data to.

Single precision floats (*np.float32*) is a good compromise, as it has 23 significant bits (mantissa) and can thus represent 16 bit integers exactly, as well as keeping the memory footprint half of a double precision float.

One commonly used method in readers is `xarray.DataArray.where()` (to mask invalid data) which can be coercing the data to *np.float64*. To ensure for example that integer data is coerced to *np.float32* when `xarray.DataArray.where()` is used, you can do:

```
my_float_dataarray = my_int_dataarray.where(some_condition, np.float32(np.nan))
```

One way of implementing a file handler is shown below:

```
# this is seviri_l1b_nc.py
from satpy.readers.file_handlers import BaseFileHandler
from pyresample.geometry import AreaDefinition

class NCSEVIRIFileHandler(BaseFileHandler):
    def __init__(self, filename, filename_info, filetype_info):
        super(NCSEVIRIFileHandler, self).__init__(filename, filename_info, filetype_info)
        self.nc = None

    def get_dataset(self, dataset_id, dataset_info):
        if dataset_id['calibration'] != 'radiance':
            # TODO: implement calibration to reflectance or brightness temperature
            return
        if self.nc is None:
            self.nc = xr.open_dataset(self.filename,
                                      decode_cf=True,
                                      mask_and_scale=True,
                                      chunks={'num_columns_vis_ir': "auto",
                                              'num_rows_vis_ir': "auto"})
            self.nc = self.nc.rename({'num_columns_vir_ir': 'x', 'num_rows_vir_ir': 'y'})
        dataset = self.nc[dataset_info['nc_key']]
        dataset.attrs.update(dataset_info)
        return dataset

    def get_area_def(self, dataset_id):
        return pyresample.geometry.AreaDefinition(
            "some_area_name",
            "on-the-fly area",
            "geos",
            "+a=6378169.0 +h=35785831.0 +b=6356583.8 +lon_0=0 +proj=geos",
            3636,
            3636,
            [-5456233.41938636, -5453233.01608472, 5453233.01608472, 5456233.41938636])

class NCSEVIRIHRVFileHandler():
  # left as an exercise to the reader :)
```

If you have any questions, please contact the *Satpy developers*.

Auxiliary File Download

If your reader needs additional data files to do calibrations, corrections, or anything else see the *Auxiliary Data Download* document for more information on how to download and cache these files without including them in the Satpy python package.

2.14.4 Adding remote file support to a reader

Warning: This feature is currently very new and might improve and change in the future.

As of Satpy version 0.25.1 the possibility to search for files on remote file systems (see *Search for local/remote files*) as well as the possibility for supported readers to read from remote filesystems has been added.

To add this feature to a reader the call to `xarray.open_dataset()` has to be replaced by the function *`open_dataset()`* included in Satpy which handles passing on the filename to be opened regardless if it is a local file path or a *`FSFile`* object which can wrap `fsspec.open()` objects.

To be able to cache the `open_dataset` call which is favourable for remote files it should be separated from the `get_dataset` method which needs to be implemented in every reader. This could look like:

```
from satpy._compat importe cached_property
from satpy.readers.file_handlers import BaseFileHandler, open_dataset

class Reader(BaseFileHandler):

    def __init__(self, filename, filename_info, filetype_info):
        super(Reader).__init__(filename, filename_info, filetype_info):

    @cached_property
    def nc(self):
        return open_dataset(self.filename, chunks="auto")

    def get_dataset(self):
        # Access the opened dataset
        data = self.nc["key"]
```

Any parameters allowed for `xarray.open_dataset()` can be passed as keywords to *`open_dataset()`* if needed.

Note: It is important to know that for remote files xarray might use a different backend to open the file than for local files (e.g. h5netcdf instead of netcdf4), which might result in some attributes being returned as arrays instead of scalars. This has to be accounted for when accessing attributes in the reader.

2.14.5 Extending Satpy via plugins

Warning: This feature is experimental and being modified without warnings. For now, it should not be used for anything else than toy examples and should not be relied on.

Satpy is able to load additional functionality outside of the builtin features in the library. It does this by searching a series of configured paths for additional configuration files for:

- readers
- composites and modifiers
- enhancements
- writers

For basic testing and temporary configuration changes, you can follow the instructions in *Component Configuration*. This will tell Satpy where to look for your custom YAML configuration files and import any Python code you'd like it to use for these components. However, this requires telling Satpy of these paths on every execution (either as an environment variable or by using `satpy.config`).

Satpy also supports being told this information via setuptools "entry points". Once your custom Python package with entry points is installed Satpy will automatically discover it when searching for composites without the user needing to explicitly import your package. This has the added benefit of organizing your YAML configuration files and any custom python code into a single python package. How to structure a package in this way is described below.

An example project showing the usage of these entry points is available at this github repository where a custom compositor is created. This repository also includes common configuration files and tools for writing clean code and automatically testing your python code.

Plugin package structure

The below sections will use the example package name `satpy-myplugin`. This is only an example and naming a plugin package with a `satpy-` prefix is not required.

A plugin package should consist of three main parts:

1. `pyproject.toml` or `setup.py`: These files define the metadata and entry points for your package. Only one of them is needed. With only a few exceptions it is recommended to use a `pyproject.toml` as this is the new and future way Python package configuration will be supported by the `pip` package manager. See below for examples of the contents of this file.
2. `mypkg/etc/`: A directory of Satpy-compatible component YAML files. These YAML files should be in `readers/`, `composites/`, `enhancements/`, and `writers/` directories. These YAML files must follow the Satpy naming conventions for each component. For example, composites and enhancements allow for sensor-specific configuration files. Other directories can be added in this `etc` directory and will be ignored by Satpy. Satpy will collect all available YAML files from all installed plugins and merge them with those builtin to Satpy. The Satpy builtins will be used as a "base" configuration with all external YAML files applied after.
3. `mypkg/`: The python package with any custom python code. This code should be based on or at least compatible with Satpy's base classes for each component or use utilities available from Satpy whenever possible.
    - readers: `FileYAMLReader` for any reader subclasses and `BaseFileHandler` for any custom file handlers. See *Adding a Custom Reader to Satpy* for more information.
    - composites and modifiers: `CompositeBase` for any generic compositor and `GenericCompositor` for any composite that represents an image (RGB, L, etc). For modifiers, use `ModifierBase`.

- enhancements: Although not required, consider using `satpy.enhancements.apply_enhancement()`.
- writers: *Writer*

Lastly, this directory should be structured like a standard python package. This primarily means a `mypkg/__init__.py` file should exist.

pyproject.toml

We recommend using a pyproject.toml file can be used to define the metadata and configuration for a python package. With this file it is possible to use package building tools to make an installable package. By using a special feature called "entry points" we can configure our package to its satpy features are automatically discovered by Satpy.

A `pyproject.toml` file is typically placed in the root of a project repository and at the same level as the package (ex. `satpy_myplugin/` directory). An example for a package called `satpy-myplugin` with custom composites is shown below.

```
[project]
name = "satpy-myplugin"
description = "Example Satpy plugin package definition."
version = "1.0.0"
readme = "README.md"
license = {text = "GPL-3.0-or-later"}
requires-python = ">=3.8"
dependencies = [
    "satpy",
]

[tool.setuptools]
packages = ["satpy_myplugin"]

[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

[project.entry-points."satpy.composites"]
example_composites = "satpy_myplugin"
```

This definition uses setuptools to build the resulting package (under `build-system`). There are other alternative tools (like poetry) that can be used.

Other custom components like readers and writers can be defined in the same package by using additional entry points named `satpy.readers` for readers, `satpy.writers` for writers, and `satpy.enhancements` for enhancements.

Note the difference between the usage of the package name (`satpy-myplugin`) which includes a hyphen and the package directory (`satpy_myplugin`) which uses an underscore. Your package name does not need to have a separator (hyphen) in it, but is used here due to the common practice of naming plugins this way. Package directories can't use hyphens as this would be a syntax error when trying to import the package. Underscores can't be used in package names as this is not allowed by PyPI.

The first `project` section in this TOML file specifies metadata about the package. This is most important if you plan on distributing your package on PyPI or similar package repository. We specify that our package depends on `satpy` so if someone installs it Satpy will automatically be installed. The second `tools.setuptools` section tells the package building (via `setuptools`) what directory the Python code is in. The third section, `build-system`, says what tool(s) should be used for building the package and what extra requirements are needed during this build process.

The last section, `project.entry-points."satpy.composites"` is the only section specific to this package being a Satpy plugin. At the time of writing the `example_composites = "satpy_myplugin"` portion is not actually used by Satpy but is required to properly define the entry point in the plugin package. Instead Satpy will assume that a package that defines the `satpy.composites` (or any of the other component types) entry point will have a `etc/` directory in the root of the package structure. Even so, for future compatibility, it is best to use the name of the package directory on the right-hand side of the =.

Warning: Due to some limitations in setuptools you must also define a `setup.py` file in addition to `pyproject.toml` if you'd like to use "editable" installations (`pip install -e .`). Once this setuptools issue is resolved this won't be needed. For now this minimal `setup.py` will work:

```
from setuptools import setup
setup()
```

Alternative: setup.py

If you are more comfortable creating a `setup.py`-based python package you can use `setup.py` instead of `pyproject.toml`. When used for custom composites, in a package called `satpy-myplugin` it would look something like this:

```
from setuptools import setup
import os

setup(
    name='satpy-myplugin',
    entry_points={
        'satpy.composites': [
            'example_composites = satpy_myplugin',
        ],
    },
    package_data={'satpy_myplugin': [os.path.join('etc', 'composites/*.yaml')]},
    install_requires=["satpy"],
)
```

Note the difference between the usage of the package name (`satpy-plugin`) which includes a hyphen and the package directory (`satpy_plugin`) which uses an underscore. Your package name does not need to have a separator (hyphen) in it, but is used here due to the common practice of naming plugins this way. See the `pyproject.toml` information above for more information on what each of these values means.

Licenses

Disclaimer: We are not lawyers.

Satpy source code is under the GPLv3 license. This license requires any derivative works to also be GPLv3 or GPLv3 compatible. It is our understanding that importing a Python module could be considered "linking" that source code to your own (thus being a derivative work) and would therefore require your code to be licensed with a GPLv3-compatible license. It is currently only possible to make a Satpy-compatible plugin without importing Satpy if it contains only enhancements. Writers and compositors are possible without subclassing, but are likely difficult to implement. Readers are even more difficult to implement without using Satpy's base classes and utilities. It is also our understanding that if your custom Satpy plugin code is not publicly released then it does not need to be GPLv3.

2.14.6 Satpy internal workings: having a look under the hood

Querying and identifying data arrays

DataQuery

The loading of data in Satpy is usually done through giving the name or the wavelength of the data arrays we are interested in. This way, the highest, most calibrated data arrays is often returned.

However, in some cases, we need more control over the loading of the data arrays. The way to accomplish this is to load data arrays using queries, eg:

```
scn.load([DataQuery(name='channel1', resolution=400)]
```

Here a data array with name *channel1* and of resolution *400* will be loaded if available.

Note that None is not a valid value, and keys having a value set to None will simply be ignored.

If one wants to use wildcards to query data, just provide '*', eg:

```
scn.load([DataQuery(name='channel1', resolution=400, calibration='*')]
```

Alternatively, one can provide a list as parameter to query data, like this:

```
scn.load([DataQuery(name='channel1', resolution=[400, 800])]
```

DataID

Satpy stores loaded data arrays in a special dictionary (*DatasetDict*) inside scene objects. In order to identify each data array uniquely, Satpy is assigning an ID to each data array, which is then used as the key in the scene object. These IDs are of type *DataID* and are immutable. They are not supposed to be used by regular users and should only be created in special circumstances. Satpy should take care of creating and assigning these automatically. They are also stored in the *attrs* of each data array as *_satpy_id*.

Default and custom metadata keys

One thing however that the user has control over is which metadata keys are relevant to which datasets. Satpy provides two default sets of metadata key (or ID keys), one for regular imager bands, and the other for composites. The first one contains: name, wavelength, resolution, calibration, modifiers. The second one contains: name, resolution.

As an example here is the definition of the first one in yaml:

```yaml
data_identification_keys:
  name:
    required: true
  wavelength:
    type: !!python/name:satpy.dataset.WavelengthRange
  resolution:
  calibration:
    enum:
        - reflectance
        - brightness_temperature
        - radiance
```

(continues on next page)

(continued from previous page)

```
        - counts
    transitive: true
  modifiers:
    required: true
    default: []
    type: !!python/name:satpy.dataset.ModifierTuple
```

To create a new set, the user can provide indications in the relevant yaml file. It has to be provided in header of the reader configuration file, under the *reader* section, as *data_identification_keys*. Each key under this is the name of relevant metadata key that will used to find relevant information in the attributes of the data arrays. Under each of this, a few options are available:

- *required*: if the item is required, False by default
- *type*: the type to use. More on this further down.
- *enum*: if the item has to be limited to a finite number of options, an enum can be used. Be sure to place the options in the order of preference, with the most desirable option on top.
- *default*: the default value to assign to the item if nothing (or None) is provided. If this option isn't provided, the key will simply be omitted if it is not present in the attrs or if it is None. It will be passed to the type's *convert* method if available.
- *transitive*: whether the key is to be passed when looking for dependencies of composites/modifiers. Here for example, a composite that has in a given calibration type will pass this calibration type requirement to its dependencies.

If the definition of the metadata keys need to be done in python rather than in a yaml file, it will be a dictionary very similar to the yaml code. Here is the same example as above in python:

```
from satpy.dataset import WavelengthRange, ModifierTuple

id_keys_config = {'name': {
                      'required': True,
                  },
                  'wavelength': {
                      'type': WavelengthRange,
                  },
                  'resolution': None,
                  'calibration': {
                      'enum': [
                          'reflectance',
                          'brightness_temperature',
                          'radiance',
                          'counts'
                          ],
                      'transitive': True,
                  },
                  'modifiers': {
                      'required': True,
                      'default': ModifierTuple(),
                      'type': ModifierTuple,
                  },
                  }
```

Types

Types are classes that implement a type to be used as value for metadata in the *DataID*. They have to implement a few methods:

- a *convert* class method that returns it's argument as an instance of the class
- *__hash__*, *__eq__* and *__ne__* methods
- a *distance* method the tells how "far" an instance of this class is from it's argument.

An example of such a class is the `WavelengthRange` class. Through its implementation, it allows us to use the wavelength in a query to find out which of the DataID in a list which has its central wavelength closest to that query for example.

DataID and DataQuery interactions

Different DataIDs and DataQuerys can have different metadata items defined. As such we define equality between different instances of these classes, and across the classes as equality between the sorted key/value pairs shared between the instances. If a DataQuery has one or more values set to '*', the corresponding key/value pair will be omitted from the comparison. Instances sharing no keys will no be equal.

Breaking changes from DatasetIDs

- The way to access values from the DataID and DataQuery is through getitem: *my_dataid['resolution']*
- For checking if a dataset is loaded, use *'mydataset' in scene*, as *'mydataset' in scene.keys()* will always return *False*: the *DatasetDict* instance only supports *DataID* as key type.

Creating DataID for tests

Sometimes, it is useful to create *DataID* instances for testing purposes. For these cases, the *satpy.tests.utils* module now has a *make_dsid* function that can be used just for this:

```
from satpy.tests.utils import make_dataid
did = make_dataid(name='camembert', modifiers=('runny',))
```

2.14.7 Auxiliary Data Download

Sometimes Satpy components need some extra data files to get their work done properly. These include files like Look Up Tables (LUTs), coefficients, or Earth model data (ex. elevations). This includes any file that would be too large to be included in the Satpy python package; anything bigger than a small text file. To help with this, Satpy includes utilities for downloading and caching these files only when your component is used. This saves the user from wasting time and disk space downloading files they may never use. This functionality is made possible thanks to the Pooch library.

Downloaded files are stored in the directory configured by *Data Directory*.

Adding download functionality

The utility functions for data downloading include a two step process:

1. **Registering**: Tell Satpy what files might need to be downloaded and used later.
2. **Retrieving**: Ask Satpy to download and store the files locally.

Registering

Registering a file for downloading tells Satpy the remote URL for the file, and an optional hash. The hash is used to verify a successful download. Registering can also include a `filename` to tell Satpy what to name the file when it is downloaded. If not provided it will be determined from the URL. Once registered, Satpy can be told to retrieve the file (see below) by using a "cache key". Cache keys follow the general scheme of `<component_type>/<filename>` (ex. `readers/README.rst`).

Satpy includes a low-level function and a high-level Mixin class for registering files. The higher level class is recommended for any Satpy component like readers, writers, and compositors. The lower-level *register_file()* function can be used for any other use case.

The `DataMixIn` class is automatically included in the *FileYAMLReader* and *Writer* base classes. For any other component (like a compositor) you should include it as another parent class:

```
from satpy.aux_download import DataDownloadMixin
from satpy.composites import GenericCompositor

class MyCompositor(GenericCompositor, DataDownloadMixin):
    """Compositor that uses downloaded files."""

    def __init__(self, name, url=None, known_hash=None, **kwargs):
        super().__init__(name, **kwargs)
        data_files = [{'url': url, 'known_hash': known_hash}]
        self.register_data_files(data_files)
```

However your code registers files, to be consistent it must do it during initialization so that the *find_registerable_files()*. If your component isn't a reader, writer, or compositor then this function will need to be updated to find and load your registered files. See *Offline Downloads* below for more information.

As mentioned, the mixin class is included in the base reader and writer class. To register files in these cases, include a `data_files` section in your YAML configuration file. For readers this would go under the `reader` section and for writers the `writer` section. This parameter is a list of dictionaries including a `url`, `known_hash`, and optional `filename`. For example:

```
reader:
    name: abi_l1b
    short_name: ABI L1b
    long_name: GOES-R ABI Level 1b
    ... other metadata ...
    data_files:
      - url: "https://example.com/my_data_file.dat"
      - url: "https://raw.githubusercontent.com/pytroll/satpy/main/README.rst"
        known_hash:
↪"sha256:5891286b63e7745de08c4b0ac204ad44cfdb9ab770309debaba90308305fa759"
      - url: "https://raw.githubusercontent.com/pytroll/satpy/main/RELEASING.md"
```

(continues on next page)

(continued from previous page)

```
        filename: "satpy_releasing.md"
        known_hash: null
```

See the *DataDownloadMixin* for more information.

Retrieving

Files that have been registered (see above) can be retrieved by calling the *retrieve()* function. This function expects a single argument: the cache key. Cache keys are returned by registering functions, but can also be pre-determined by following the scheme `<component_type>/<filename>` (ex. `readers/README.rst`). Retrieving a file will download it to local disk if needed and then return the local pathname. Data is stored locally in the *Data Directory*. It is up to the caller to then open the file.

Offline Downloads

To assist with operational environments, Satpy includes a *retrieve_all()* function that will try to find all files that Satpy components may need to download in the future and download them to the current directory specified by *Data Directory*. This function allows you to specify a list of `readers`, `writers`, or `composite_sensors` to limit what components are checked for files to download.

The `retrieve_all` function is also available through a command line script called `satpy_retrieve_all_aux_data`. Run the following for usage information.

```
satpy_retrieve_all_aux_data --help
```

To make sure that no additional files are downloaded when running Satpy see *Demo Data Directory*.

2.14.8 Writing unit tests

Satpy tests are written using the third-party pytest package.

Fixtures

The usage of Pytest fixtures is encouraged for code re-usability.

As the builtin fixtures (and those defined in `conftest.py` file) are injected by Pytest without them being imported explicitly, their usage can be very confusing for new developers. To lessen the confusion, it is encouraged to add a note at the top of the test modules listing all the automatically injected external fixtures that are used in the module:

```
# NOTE:
# The following fixtures are not defined in this file, but are used and injected by␣
↪Pytest:
# - tmp_path
# - fixture_defined_in_conftest.py
```

2.14.9 Coding guidelines

Satpy is part of Pytroll, and all code should follow the Pytroll coding guidelines and best practices.

Satpy is now Python 3 only and it is no longer needed to support Python 2. Check `setup.py` for the current Python versions any new code needs to support.

2.14.10 Development installation

See the *Installation Instructions* section for basic installation instructions. When it comes time to install Satpy it should be installed from a clone of the git repository and in development mode so that local file changes are automatically reflected in the python environment. We highly recommend making a separate conda environment or virtualenv for development. For example, you can do this using conda:

```
conda create -n satpy-dev python=3.11
conda activate satpy-dev
```

This will create a new environment called "satpy-dev" with Python 3.11 installed. The second command will activate the environment so any future conda, python, or pip commands will use this new environment.

If you plan on contributing back to the project you should first fork the repository and clone your fork. The package can then be installed in development mode by doing:

```
conda install --only-deps satpy
pip install -e .
```

The first command will install all dependencies needed by the Satpy conda-forge package, but won't actually install Satpy. The second command should be run from the root of the cloned Satpy repository (where the `setup.py` is) and will install the actual package.

You can now edit the python files in your cloned repository and have them immediately reflected in your conda environment.

All the required dependencies for a full development environment, i.e. running the tests and building the documentation, can be installed with:

```
conda install eccodes
pip install -e ".[all]"
```

2.14.11 Running tests

Satpy tests are written using the third-party pytest package. There is usually no need to run all Satpy tests, but instead only run the tests related to the component you are working on. All tests are automatically run from the GitHub Pull Request using multiple versions of Python, multiple operating systems, and multiple versions of dependency libraries. If you want to run all Satpy tests you will need to install additional dependencies that aren't needed for regular Satpy usage. To install them run:

```
conda install eccodes
pip install -e ".[tests]"
```

Satpy tests can be executed by running:

```
pytest satpy/tests
```

You can also run a specific tests by specifying a sub-directory or module:

```
pytest satpy/tests/reader_tests/test_abi_l1b.py
```

2.14.12 Running benchmarks

Satpy benchmarks are written using the Airspeed Velocity package (`asv`). The benchmarks can be run using:

```
asv run
```

These are pretty computation intensive, and shouldn't be run unless you want to diagnose some performance issue for example.

Once the benchmarks have run, you can use:

```
asv publish
asv preview
```

to have a look at the results. Again, have a look at the *asv* documentation for more information.

2.14.13 Documentation

Satpy's documentation is built using Sphinx. All documentation lives in the `doc/` directory of the project repository. For building the documentation, additional packages are needed. These can be installed with

```
pip install -e ".[all]"
```

After editing the source files there the documentation can be generated locally:

```
cd doc
make html
```

The output of the make command should be checked for warnings and errors. If code has been changed (new functions or classes) then the API documentation files should be regenerated before running the above command:

```
sphinx-apidoc -f -T -o source/api ../satpy ../satpy/tests
```

2.15 satpy

2.15.1 satpy package

Subpackages

satpy.cf package

Submodules

satpy.cf.area module

CF processing of pyresample area information.

satpy.cf.area.**_add_grid_mapping**(*data_arr: DataArray*) → tuple[DataArray, DataArray]

Convert an area to at CF grid mapping.

satpy.cf.area.**_add_lonlat_coords**(*data_arr: DataArray*) → DataArray

Add 'longitude' and 'latitude' coordinates to DataArray.

satpy.cf.area.**_create_grid_mapping**(*area*)

Create the grid mapping instance for *area*.

satpy.cf.area.**area2cf**(*data_arr: DataArray*, *include_lonlats: bool = False*, *got_lonlats: bool = False*) → list[DataArray]

Convert an area to at CF grid mapping or lon and lats.

satpy.cf.attrs module

CF processing of attributes.

class satpy.cf.attrs.**AttributeEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: JSONEncoder

JSON encoder for dataset attributes.

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an RecursionError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is None and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a TypeError.

_encode(*obj*)

Encode the given object as a json-serializable datatype.

default(*obj*)

Return a json-serializable object for *obj*.

In order to facilitate decoding, elements in dictionaries, lists/tuples and multi-dimensional arrays are encoded recursively.

satpy.cf.attrs.**_add_ancillary_variables_attrs**(*data_arr: DataArray*) → None

Replace ancillary_variables DataArray with a list of their name.

satpy.cf.attrs.**_add_history**(*attrs*)

Add 'history' attribute to dictionary.

satpy.cf.attrs.**_drop_attrs**(*data_arr: DataArray*, *user_excluded_attrs: list[str] | None*) → None

Remove undesirable attributes.

satpy.cf.attrs.**_encode_nc_attrs**(*attrs*)

Encode dataset attributes in a netcdf compatible datatype.

Parameters
attrs (*dict*) – Attributes to be encoded

Returns
Encoded (and sorted) attributes

Return type
dict

satpy.cf.attrs.**_encode_numpy_array**(*obj*)

Encode numpy array as a netCDF4 serializable datatype.

satpy.cf.attrs.**_encode_object**(*obj*)

Try to encode *obj* as a netCDF/Zarr compatible datatype which most closely resembles the object's nature.

Raises
ValueError if no such datatype could be found –

satpy.cf.attrs.**_encode_python_objects**(*obj*)

Try to find the datatype which most closely resembles the object's nature.

If on failure, encode as a string. Plain lists are encoded recursively.

satpy.cf.attrs.**_encode_to_cf**(*obj*)

Encode the given object as a netcdf compatible datatype.

satpy.cf.attrs.**_format_prerequisites_attrs**(*data_arr: DataArray*) → None

Reformat prerequisites attribute value to string.

satpy.cf.attrs.**_get_none_attrs**(*data_arr: DataArray*) → list[str]

Remove attribute keys with None value.

satpy.cf.attrs.**_get_satpy_attrs**(*data_arr: DataArray*) → list[str]

Remove _satpy attribute.

satpy.cf.attrs.**_try_decode_object**(*obj*)

Try to decode byte string.

satpy.cf.attrs.**preprocess_attrs**(*data_arr: DataArray*, *flatten_attrs: bool*, *exclude_attrs: list[str] | None*) → DataArray

Preprocess DataArray attributes to be written into CF-compliant netCDF/Zarr.

satpy.cf.attrs.**preprocess_header_attrs**(*header_attrs*, *flatten_attrs=False*)

Prepare file header attributes.

satpy.cf.coords module

Set CF-compliant spatial and temporal coordinates.

satpy.cf.coords.**_add_declared_coordinates**(*data_arrays: dict[str, DataArray]*, *dataarray_name: str*) → dict[str, DataArray]

Add declared coordinates to the dataarray if they exist.

satpy.cf.coords.**_add_xy_geographic_coords_attrs**(*data_arr: DataArray*, *x: str = 'x'*, *y: str = 'y'*) → DataArray

Add relevant attributes to x, y coordinates of a geographic CRS.

satpy.cf.coords.**_add_xy_projected_coords_attrs**(*data_arr: DataArray*, *x: str = 'x'*, *y: str = 'y'*) → DataArray

Add relevant attributes to x, y coordinates of a projected CRS.

satpy.cf.coords.**_get_coordinates_list**(*data_arr: DataArray*) → list[str]

Return a list with the coordinates names specified in the 'coordinates' attribute.

satpy.cf.coords.**_get_is_nondimensional_coords_dict**(*data_arrays: dict[str, DataArray]*) → dict[str, bool]

satpy.cf.coords.**_is_area**(*data_arr: DataArray*) → bool

satpy.cf.coords.**_is_lon_or_lat_dataarray**(*data_arr: DataArray*) → bool

Check if the DataArray represents the latitude or longitude coordinate.

satpy.cf.coords.**_is_projected**(*data_arr: DataArray*) → bool

Guess whether data are projected or not.

satpy.cf.coords.**_is_swath**(*data_arr: DataArray*) → bool

satpy.cf.coords.**_rename_coords**(*data_arrays: dict[str, DataArray]*, *coord_name: str*) → dict[str, DataArray]

Rename coordinates in the datasets.

satpy.cf.coords.**_try_add_coordinate**(*data_arrays: dict[str, DataArray]*, *dataarray_name: str*, *coord: str*) → dict[str, DataArray]

Try to add a coordinate to the dataarray, warn if not possible.

satpy.cf.coords.**_try_get_units_from_coords**(*data_arr: DataArray*) → str | None

Try to retrieve coordinate x/y units.

satpy.cf.coords.**_try_to_get_crs**(*data_arr: DataArray*) → CRS

Try to get a CRS from attributes.

satpy.cf.coords.**_warn_if_pretty_but_not_unique**(*pretty*, *coord_name*)

Warn if coordinates cannot be pretty-formatted due to non-uniqueness.

satpy.cf.coords.**add_coordinates_attrs_coords**(*data_arrays: dict[str, DataArray]*) → dict[str, DataArray]

Add to DataArrays the coordinates specified in the 'coordinates' attribute.

It deal with the 'coordinates' attributes indicating lat/lon coords The 'coordinates' attribute is dropped from each DataArray

If the *coordinates* attribute of a data array links to other dataarrays in the scene, for example *coordinates='lon lat'*, add them as coordinates to the data array and drop that attribute.

In the final call to *xr.Dataset.to_netcdf()* all coordinate relations will be resolved and the *coordinates* attributes be set automatically.

`satpy.cf.coords.add_time_bounds_dimension(ds: Dataset, time: str = 'time') → Dataset`

Add time bound dimension to xr.Dataset.

`satpy.cf.coords.add_xy_coords_attrs(data_arr: DataArray) → DataArray`

Add relevant attributes to x, y coordinates.

`satpy.cf.coords.check_unique_projection_coords(data_arrays: dict[str, DataArray]) → None`

Check that all datasets share the same projection coordinates x/y.

`satpy.cf.coords.ensure_unique_nondimensional_coords(data_arrays: dict[str, DataArray], pretty: bool = False) → dict[str, DataArray]`

Make non-dimensional coordinates unique among all datasets.

Non-dimensional coordinates, such as scanline timestamps, may occur in multiple datasets with the same name and dimension but different values.

In order to avoid conflicts, prepend the dataset name to the coordinate name. If a non-dimensional coordinate is unique among all datasets and `pretty=True`, its name will not be modified.

Since all datasets must have the same projection coordinates, this is not applied to latitude and longitude.

Parameters

- **data_arrays** – Dictionary of (dataset name, dataset)
- **pretty** – Don't modify coordinate names, if possible. Makes the file prettier, but possibly less consistent.

Returns

Dictionary holding the updated datasets

`satpy.cf.coords.has_projection_coords(data_arrays: dict[str, DataArray]) → bool`

Check if DataArray collection has a "longitude" or "latitude" DataArray.

`satpy.cf.coords.set_cf_time_info(data_arr: DataArray, epoch: str | None) → DataArray`

Set CF time attributes and encoding.

It expand the DataArray with a time dimension if does not yet exists.

The function assumes

- that x and y dimensions have at least shape > 1
- the time coordinate has size 1

satpy.cf.data_array module

Utility to generate a CF-compliant DataArray.

`satpy.cf.data_array._handle_data_array_name(original_name, numeric_name_prefix)`

`satpy.cf.data_array._preprocess_data_array_name(dataarray, numeric_name_prefix, include_orig_name)`

Change the DataArray name by prepending numeric_name_prefix if the name is a digit.

satpy.cf.data_array.**make_cf_data_array**(*dataarray*, *epoch=None*, *flatten_attrs=False*, *exclude_attrs=None*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*)

Make the xr.DataArray CF-compliant.

Parameters

- **dataarray** (*xr.DataArray*) – The data array to be made CF-compliant.
- **epoch** (*str, optional*) – Reference time for encoding of time coordinates. If None, the default reference time is defined using *from satpy.cf.coords import EPOCH*.
- **flatten_attrs** (*bool, optional*) – If True, flatten dict-type attributes. Defaults to False.
- **exclude_attrs** (*list, optional*) – List of dataset attributes to be excluded. Defaults to None.
- **include_orig_name** (*bool, optional*) – Include the original dataset name in the netcdf variable attributes. Defaults to True.
- **numeric_name_prefix** (*str, optional*) – Prepend dataset name with this if starting with a digit. Defaults to "CHANNEL_".

Returns
A CF-compliant xr.DataArray.

Return type
xr.DataArray

satpy.cf.datasets module

Utility to generate a CF-compliant Datasets.

satpy.cf.datasets.**_collect_cf_dataset**(*list_dataarrays*, *epoch=None*, *flatten_attrs=False*, *exclude_attrs=None*, *include_lonlats=True*, *pretty=False*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*)

Process a list of xr.DataArray and return a dictionary with CF-compliant xr.Dataset.

Parameters

- **list_dataarrays** (*list*) – List of DataArrays to make CF compliant and merge into an xr.Dataset.
- **epoch** (*str, optional*) – Reference time for encoding the time coordinates. Example format: "seconds since 1970-01-01 00:00:00". If None, the default reference time is defined using *from satpy.cf.coords import EPOCH*.
- **flatten_attrs** (*bool, optional*) – If True, flatten dict-type attributes.
- **exclude_attrs** (*list, optional*) – List of xr.DataArray attribute names to be excluded.
- **include_lonlats** (*bool, optional*) – If True, includes 'latitude' and 'longitude' coordinates also for a satpy.Scene defined on an AreaDefinition. If the 'area' attribute is a SwathDefinition, it always includes latitude and longitude coordinates.
- **pretty** (*bool, optional*) – Don't modify coordinate names, if possible. Makes the file prettier, but possibly less consistent.
- **include_orig_name** (*bool, optional*) – Include the original dataset name as a variable attribute in the xr.Dataset.

- **numeric_name_prefix** (*str, optional*) – Prefix to add to each variable with a name starting with a digit. Use '' or None to leave this out.

Returns
A partially CF-compliant xr.Dataset.

Return type
xr.Dataset

satpy.cf.datasets.**_get_extra_ds**(*dataarray*, *keys=None*)

Get the ancillary_variables DataArrays associated to a dataset.

satpy.cf.datasets.**_get_group_dataarrays**(*group_members*, *list_dataarrays*)

Yield DataArrays that are part of a specific group.

satpy.cf.datasets.**_get_groups**(*groups*, *list_datarrays*)

Return a dictionary with the list of xr.DataArray associated to each group.

If no groups (groups=None), return all DataArray attached to a single None key. Else, collect the DataArrays associated to each group.

satpy.cf.datasets.**collect_cf_datasets**(*list_dataarrays*, *header_attrs=None*, *exclude_attrs=None*, *flatten_attrs=False*, *pretty=True*, *include_lonlats=True*, *epoch=None*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*, *groups=None*)

Process a list of xr.DataArray and return a dictionary with CF-compliant xr.Datasets.

If the xr.DataArrays does not share the same dimensions, it creates a collection of xr.Datasets sharing the same dimensions.

Parameters

- **list_dataarrays** (*list*) – List of DataArrays to make CF compliant and merge into groups of xr.Datasets.
- **header_attrs** (*dict*) – Global attributes of the output xr.Dataset.
- **epoch** (*str, optional*) – Reference time for encoding the time coordinates. Example format: "seconds since 1970-01-01 00:00:00". If None, the default reference time is retrieved using *from satpy.cf.coords import EPOCH*.
- **flatten_attrs** (*bool, optional*) – If True, flatten dict-type attributes.
- **exclude_attrs** (*list, optional*) – List of xr.DataArray attribute names to be excluded.
- **include_lonlats** (*bool, optional*) – If True, includes 'latitude' and 'longitude' coordinates also for a satpy.Scene defined on an AreaDefinition. If the 'area' attribute is a SwathDefinition, it always includes latitude and longitude coordinates.
- **pretty** (*bool, optional*) – Don't modify coordinate names, if possible. Makes the file prettier, but possibly less consistent.
- **include_orig_name** (*bool, optional*) – Include the original dataset name as a variable attribute in the xr.Dataset.
- **numeric_name_prefix** (*str, optional*) – Prefix to add to each variable with a name starting with a digit. Use '' or None to leave this out.
- **groups** (*dict, optional*) – Group datasets according to the given assignment: *{'<group_name>': ['dataset_name1', 'dataset_name2', ...]}*. Used to create grouped netCDFs using the CF_Writer. If None, no groups will be created.

Returns

A tuple containing:

- grouped_datasets (dict): A dictionary of CF-compliant xr.Dataset: {group_name: xr.Dataset}.
- header_attrs (dict): Global attributes to be attached to the xr.Dataset / netCDF4.

Return type
tuple

satpy.cf.encoding module

CF encoding.

`satpy.cf.encoding.`**`_set_default_chunks`**(*encoding*, *dataset*)

Update encoding to preserve current dask chunks.

Existing user-defined chunks take precedence.

`satpy.cf.encoding.`**`_set_default_fill_value`**(*encoding*, *dataset*)

Set default fill values.

Avoid _FillValue attribute being added to coordinate variables (https://github.com/pydata/xarray/issues/1865).

`satpy.cf.encoding.`**`_set_default_time_encoding`**(*encoding*, *dataset*)

Set default time encoding.

Make sure time coordinates and bounds have the same units. Default is xarray's CF datetime encoding, which can be overridden by user-defined encoding.

`satpy.cf.encoding.`**`_update_encoding_dataset_names`**(*encoding*, *dataset*, *numeric_name_prefix*)

Ensure variable names of the encoding dictionary account for numeric_name_prefix.

A lot of channel names in satpy starts with a digit. When preparing CF-compliant datasets, these channels are prefixed with numeric_name_prefix.

If variables names in the encoding dictionary are numeric digits, their name is prefixed with numeric_name_prefix

`satpy.cf.encoding.`**`update_encoding`**(*dataset*, *to_engine_kwargs*, *numeric_name_prefix='CHANNEL_'*)

Update encoding.

Preserve dask chunks, avoid fill values in coordinate variables and make sure that time & time bounds have the same units.

Module contents

Code for generation of CF-compliant datasets.

satpy.composites package

Submodules

satpy.composites.abi module

Composite classes for the ABI instrument.

class satpy.composites.abi.**SimulatedGreen**(*name*, *fractions=(0.465, 0.465, 0.07)*, ***kwargs*)

Bases: *GenericCompositor*

A single-band dataset resembling a Green (0.55 µm) band.

This compositor creates a single band product by combining three other bands in various amounts. The general formula with dependencies (d) and fractions (f) is:

```
result = d1 * f1 + d2 * f2 + d3 * f3
```

See the *fractions* keyword argument for more information. Common used fractions for ABI data with C01, C02, and C03 inputs include:

- SatPy default (historical): (0.465, 0.465, 0.07)
- CIMSS (Kaba): (0.45, 0.45, 0.10)
- EDC: (0.45706946, 0.48358168, 0.06038137)

Initialize fractions for input channels.

Parameters

- **name** (*str*) – Name of this composite
- **fractions** (*iterable*) – Fractions of each input band to include in the result.

satpy.composites.agri module

Composite classes for the AGRI instrument.

class satpy.composites.agri.**SimulatedRed**(*name*, *fractions=(1.0, 0.13, 0.87)*, ***kwargs*)

Bases: *GenericCompositor*

A single-band dataset resembling a Red (0.64 µm) band.

This compositor creates a single band product by combining two other bands by preset amounts. The general formula with dependencies (d) and fractions (f) is:

```
result = (f1 * d1 - f2 * d2) / f3
```

See the *fractions* keyword argument for more information. The default setup is to use:

- f1 = 1.0
- f2 = 0.13
- f3 = 0.87

Initialize fractions for input channels.

Parameters

- **name** (*str*) – Name of this composite
- **fractions** (*iterable*) – Fractions of each input band to include in the result.

satpy.composites.ahi module

Composite classes for AHI.

satpy.composites.cloud_products module

Compositors for cloud products.

class satpy.composites.cloud_products.**CloudCompositorCommonMask**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *SingleBandCompositor*

Put cloud-free pixels as fill_value_color in palette.

Initialise the compositor.

class satpy.composites.cloud_products.**CloudCompositorWithoutCloudfree**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *SingleBandCompositor*

Put cloud-free pixels as fill_value_color in palette.

Initialise the compositor.

class satpy.composites.cloud_products.**PrecipCloudsRGB**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Precipitation clouds compositor.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

satpy.composites.config_loader module

Classes for loading compositor and modifier configuration files.

class satpy.composites.config_loader.**_CompositeConfigHelper**(*loaded_compositors*, *sensor_id_keys*)

Bases: object

Helper class for parsing composite configurations.

The provided *loaded_compositors* dictionary is updated inplace.

_create_comp_from_info(*composite_info*, *loader*)

static _get_compositor_loader_from_config(*composite_name*, *composite_info*)

_handle_inline_comp_dep(*dep_info*, *dep_num*, *parent_name*)

_load_config_composite(*composite_info*)

_load_config_composites(*configured_composites*)

_process_composite_deps(*composite_info*)

parse_config(*configured_composites*, *composite_configs*)

Parse composite configuration dictionary.

class satpy.composites.config_loader.**_ModifierConfigHelper**(*loaded_modifiers*, *sensor_id_keys*)

Bases: object

Helper class for parsing modifier configurations.

The provided *loaded_modifiers* dictionary is updated inplace.

static **_get_modifier_loader_from_config**(*modifier_name*, *modifier_info*)

_load_config_modifier(*modifier_info*)

_load_config_modifiers(*configured_modifiers*)

_process_modifier_deps(*modifier_info*)

parse_config(*configured_modifiers*, *composite_configs*)

Parse modifier configuration dictionary.

satpy.composites.config_loader.**_convert_dep_info_to_data_query**(*dep_info*)

satpy.composites.config_loader.**_get_sensor_id_keys**(*conf*, *parent_id_keys*)

satpy.composites.config_loader.**_load_config**(*composite_configs*)

satpy.composites.config_loader.**_lru_cache_with_config_path**(*func: Callable*)

Use lru_cache but include satpy's current config_path.

satpy.composites.config_loader.**_update_cached_wrapper**(*wrapper*, *cached_func*)

satpy.composites.config_loader.**all_composite_sensors**()

Get all sensor names from available composite configs.

satpy.composites.config_loader.**load_compositor_configs_for_sensor**(*sensor_name: str*) → tuple[dict[str, dict], dict[str, dict], dict]

Load compositor, modifier, and DataID key information from configuration files for the specified sensor.

Parameters

sensor_name – Sensor name that has matching `sensor_name.yaml` config files.

Returns

Where *comps* is a dictionary:

composite ID -> compositor object

And *mods* is a dictionary:

modifier name -> (modifier class, modifiers options)

Add *data_id_keys* is a dictionary:

DataID key -> key properties

Return type
(comps, mods, data_id_keys)

satpy.composites.config_loader.**load_compositor_configs_for_sensors**(*sensor_names: Iterable[str]*) → tuple[dict[str, dict], dict[str, dict]]

Load compositor and modifier configuration files for the specified sensors.

Parameters
sensor_names (*list of strings*) – Sensor names that have matching `sensor_name.yaml` config files.

Returns

Where *comps* is a dictionary:

sensor_name -> composite ID -> compositor object

And *mods* is a dictionary:

sensor_name -> modifier name -> (modifier class, modifiers options)

Return type
(comps, mods)

satpy.composites.glm module

Composite classes for the GLM instrument.

class satpy.composites.glm.**HighlightCompositor**(*name*, *min_highlight=0.0*, *max_highlight=10.0*, *max_factor=(0.8, 0.8, -0.8, 0)*, ***kwargs*)

Bases: *GenericCompositor*

Highlight pixels of a layer by an amount determined by a secondary layer.

The highlighting is applied per channel to either add or subtract an intensity from the primary image. In the addition case, the code is essentially doing:

```
highlight_factor = (highlight_data - min_highlight) / (max_highlight - min_
↪highlight)
channel_result = primary_data + highlight_factor * max_factor
```

The `max_factor` is defined per channel and can be positive for an additive effect, negative for a subtractive effect, or zero for no effect.

Initialize composite with highlight factor options.

Parameters

- **min_highlight** (*float*) – Minimum raw value of the "highlight" data that will be used for linearly scaling the data along with `max_hightlight`.
- **max_highlight** (*float*) – Maximum raw value of the "highlight" data that will be used for linearly scaling the data along with `min_hightlight`.
- **max_factor** (*tuple*) – Maximum effect that the highlight data can have on each channel of the primary image data. This will be multiplied by the linearly scaled highlight data and then added or subtracted from the highlight channels. See class docstring for more information.

By default this is set to `(0.8, 0.8, -0.8, 0)` meaning the Red and Green channel will be added to by at most 0.8, the Blue channel will be subtracted from by at most 0.8, and the Alpha channel will not be effected.

_apply_highlight_effect(*background_data*, *factor*)

static _get_enhanced_background_data(*background_layer*)

_get_highlight_factor(*highlight_data*)

_update_attrs(*new_data*, *background_layer*, *highlight_layer*)

satpy.composites.sar module

Composite classes for the VIIRS instrument.

class satpy.composites.sar.**SARIce**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

The SAR Ice composite.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARIceLegacy**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

The SAR Ice composite, legacy version with dynamic stretching.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARIceLog**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

The SAR Ice composite, using log-scale data.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARQuickLook**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

The SAR QuickLook composite.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.sar.**SARRGB**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

The SAR RGB composite.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

satpy.composites.sar.**_square_root_channels**(**projectables*)

Return the square root of the channels, preserving the attributes.

satpy.composites.sar.**overlay**(*top*, *bottom*, *maxval=None*)

Blending two layers.

from: https://docs.gimp.org/en/gimp-concepts-layer-modes.html

satpy.composites.sar.**soft_light**(*top*, *bottom*, *maxval*)

Apply soft light.

http://www.pegtop.net/delphi/articles/blendmodes/softlight.htm

satpy.composites.spectral module

Composite classes for spectral adjustments.

class satpy.composites.spectral.**GreenCorrector**(**args*, *fractions=(0.85, 0.15)*, ***kwargs*)

Bases: *SpectralBlender*

Previous class used to blend channels for green band corrections.

This method has been refactored to make it more generic. The replacement class is 'SpectralBlender' which computes a weighted average based on N number of channels and N number of corresponding weights/fractions. A new class called 'HybridGreen' has been created, which performs a correction of green bands centered at 0.51 microns following Miller et al. (2016, DOI:10.1175/BAMS-D-15-00154.2) in order to improve true color imagery.

Set default keyword argument values.

class satpy.composites.spectral.**HybridGreen**(**args*, *fraction=0.15*, ***kwargs*)

Bases: *SpectralBlender*

Corrector of the FCI or AHI green band.

The green band in FCI and AHI (and other bands centered at 0.51 microns) deliberately misses the chlorophyll spectral reflectance local maximum at 0.55 microns in order to focus on aerosol and ash rather than on vegetation. This affects true colour RGBs, because vegetation looks brown rather than green and barren surface types typically gets a reddish hue.

To correct for this the hybrid green approach proposed by Miller et al. (2016, DOI:10.1175/BAMS-D-15-00154.2) is used. The basic idea is to include some contribution also from the 0.86 micron channel, which is known for its sensitivity to vegetation. The formula used for this is:

```
hybrid_green = (1 - F) * R(0.51) + F * R(0.86)
```

where F is a constant value, that is set to 0.15 by default in Satpy.

For example, the HybridGreen compositor can be used as follows to construct a hybrid green channel for AHI, with 15% contibution from the near-infrared 0.85 µm band (B04) and the remaining 85% from the native green 0.51 µm band (B02):

```
hybrid_green:
  compositor: !!python/name:satpy.composites.spectral.HybridGreen
  fraction: 0.15
  prerequisites:
    - name: B02
      modifiers: [sunz_corrected, rayleigh_corrected]
    - name: B04
      modifiers: [sunz_corrected, rayleigh_corrected]
  standard_name: toa_bidirectional_reflectance
```

Other examples can be found in the `ahi.yaml` and `ami.yaml` composite files in the satpy distribution.

Set default keyword argument values.

class satpy.composites.spectral.**NDVIHybridGreen**(**args*, *ndvi_min=0.0*, *ndvi_max=1.0*, *limits=(0.15, 0.05)*, *strength=1.0*, ***kwargs*)

Bases: *SpectralBlender*

Construct a NDVI-weighted hybrid green channel.

This green band correction follows the same approach as the HybridGreen compositor, but with a dynamic blend factor f that depends on the pixel-level Normalized Differece Vegetation Index (NDVI). The higher the NDVI, the smaller the contribution from the nir channel will be, following a liner (default) or non-linear relationship between the two ranges *[ndvi_min, ndvi_max]* and *limits*.

As an example, a new green channel using e.g. FCI data and the NDVIHybridGreen compositor can be defined like:

```
ndvi_hybrid_green:
  compositor: !!python/name:satpy.composites.spectral.NDVIHybridGreen
  ndvi_min: 0.0
  ndvi_max: 1.0
  limits: [0.15, 0.05]
  strength: 1.0
  prerequisites:
    - name: vis_05
      modifiers: [sunz_corrected, rayleigh_corrected]
    - name: vis_06
      modifiers: [sunz_corrected, rayleigh_corrected]
    - name: vis_08
      modifiers: [sunz_corrected ]
  standard_name: toa_bidirectional_reflectance
```

In this example, pixels with NDVI=0.0 will be a weighted average with 15% contribution from the near-infrared vis_08 channel and the remaining 85% from the native green vis_05 channel, whereas pixels with NDVI=1.0 will be a weighted average with 5% contribution from the near-infrared vis_08 channel and the remaining 95% from the native green vis_05 channel. For other values of NDVI a linear interpolation between these values will be performed.

A strength larger or smaller than 1.0 will introduce a non-linear relationship between the two ranges *[ndvi_min, ndvi_max]* and *limits*. Hence, a higher strength (> 1.0) will result in a slower transition to higher/lower fractions at the NDVI extremes. Similarly, a lower strength (< 1.0) will result in a faster transition to higher/lower fractions at the NDVI extremes.

Initialize class and set the NDVI limits, blending fraction limits and strength.

_apply_strength(*ndvi*)

Introduce non-linearity by applying strength factor.

The method introduces non-linearity to the ndvi for a non-linear scaling from ndvi to blend fraction in *_compute_blend_fraction*. This can be used for a slower or faster transision to higher/lower fractions at the ndvi extremes. If strength equals 1.0, this operation has no effect on the ndvi.

_compute_blend_fraction(*ndvi*)

Compute pixel-level fraction of NIR signal to blend with native green signal.

This method linearly scales the input ndvi values to pixel-level blend fractions within the range *[limits[0], limits[1]]* following this implementation <https://stats.stackexchange.com/a/281164>.

class satpy.composites.spectral.**SpectralBlender**(**args*, *fractions=()*, ***kwargs*)

Bases: *GenericCompositor*

Construct new channel by blending contributions from a set of channels.

This class can be used to compute weighted average of different channels. Primarily it's used to correct the green band of AHI and FCI in order to allow for proper true color imagery.

Below is an example used to generate a corrected green channel for AHI using a weighted average from three channels, with 63% contribution from the native green channel (B02), 29% from the red channel (B03) and 8% from the near-infrared channel (B04):

```
corrected_green:
  compositor: !!python/name:satpy.composites.spectral.SpectralBlender
  fractions: [0.63, 0.29, 0.08]
  prerequisites:
    - name: B02
      modifiers: [sunz_corrected, rayleigh_corrected]
    - name: B03
      modifiers: [sunz_corrected, rayleigh_corrected]
    - name: B04
      modifiers: [sunz_corrected, rayleigh_corrected]
  standard_name: toa_bidirectional_reflectance
```

Other examples can be found in the``ahi.yaml`` composite file in the satpy distribution.

Set default keyword argument values.

satpy.composites.viirs module

Composite classes for the VIIRS instrument.

class satpy.composites.viirs.**AdaptiveDNB**(**args*, ***kwargs*)

Bases: *HistogramDNB*

Adaptive histogram equalized DNB composite.

The logic for this code was taken from Polar2Grid and was originally developed by Eva Schiffer (SSEC).

This composite separates the DNB data in to 3 main regions: Day, Night, and Mixed. Each region is equalized separately to bring out the most information from the region due to the high dynamic range of the DNB data. Optionally, the mixed region can be separated in to multiple smaller regions by using the *mixed_degree_step* keyword.

Initialize the compositor with values from the user or from the configuration file.

Adaptive histogram equalization and regular histogram equalization can be configured independently for each region: day, night, or mixed. A region can be set to use adaptive equalization "always", or "never", or only when there are multiple regions in a single scene "multiple" via the *adaptive_X* keyword arguments (see below).

Parameters

- **adaptive_day** – one of ("always", "multiple", "never") meaning when adaptive equalization is used.
- **adaptive_mixed** – one of ("always", "multiple", "never") meaning when adaptive equalization is used.
- **adaptive_night** – one of ("always", "multiple", "never") meaning when adaptive equalization is used.

_normalize_dnb_for_mask(*dnb_data*, *sza_data*, *good_mask*, *output_dataset*)

class satpy.composites.viirs.**ERFDNB**(**args*, ***kwargs*)

Bases: *CompositeBase*

Equalized DNB composite using the error function (erf).

The logic for this code was taken from Polar2Grid and was originally developed by Curtis Seaman and Steve Miller. The original code was written in IDL and is included as comments in the code below.

Initialize ERFDNB specific keyword arguments.

_saturation_correction(*dnb_data*, *unit_factor*, *min_val*, *max_val*)

class satpy.composites.viirs.**HistogramDNB**(**args*, ***kwargs*)

Bases: *CompositeBase*

Histogram equalized DNB composite.

The logic for this code was taken from Polar2Grid and was originally developed by Eva Schiffer (SSEC).

This composite separates the DNB data in to 3 main regions: Day, Night, and Mixed. Each region is equalized separately to bring out the most information from the region due to the high dynamic range of the DNB data. Optionally, the mixed region can be separated in to multiple smaller regions by using the *mixed_degree_step* keyword.

Initialize the compositor with values from the user or from the configuration file.

Parameters

- **high_angle_cutoff** – solar zenith angle threshold in degrees, values above this are considered "night"
- **low_angle_cutoff** – solar zenith angle threshold in degrees, values below this are considered "day"
- **mixed_degree_step** – Step interval to separate "mixed" region in to multiple parts by default does whole mixed region

_normalize_dnb_for_mask(*dnb_data*, *sza_data*, *good_mask*, *output_dataset*)

_normalize_dnb_with_day_night_masks(*dnb_data*, *day_mask*, *mixed_mask*, *night_mask*, *output_dataset*)

_run_dnb_normalization(*dnb_data*, *sza_data*)

Scale the DNB data using a histogram equalization method.

Parameters

- **dnb_data** (*ndarray*) – Day/Night Band data array
- **sza_data** (*ndarray*) – Solar Zenith Angle data array

class satpy.composites.viirs.**NCCZinke**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *CompositeBase*

Equalized DNB composite using the Zinke algorithm[1].

References

Initialise the compositor.

static _gain_factor(*theta*)

gain_factor(*theta*)

Compute gain factor in a dask-friendly manner.

class satpy.composites.viirs.**SnowAge**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Create RGB snow product.

Product is based on method presented at the second CSPP/IMAPP users' meeting at Eumetsat in Darmstadt on 14-16 April 2015

Bernard Bellec snow Look-Up Tables V 1.0 (c) Meteo-France These Look-up Tables allow you to create the RGB snow product for SUOMI-NPP VIIRS Imager according to the algorithm presented at the second CSPP/IMAPP users' meeting at Eumetsat in Darmstadt on 14-16 April 2015 The algorithm and the product are described in this presentation : http://www.ssec.wisc.edu/meetings/cspp/2015/Agenda%20PDF/Wednesday/Roquet_snow_product_cspp2015.pdf as well as in the paper http://dx.doi.org/10.1016/j.rse.2017.04.028 For further information you may contact Bernard Bellec at Bernard.Bellec@meteo.fr or Pascale Roquet at Pascale.Roquet@meteo.fr

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

satpy.composites.viirs.**_calculate_weights**(*tile_size*)

Calculate a weight array for bilinear interpolation of histogram tiles.

The weight array will be used to quickly bilinearly-interpolate the histogram equalizations tile size should be the width and height of a tile in pixels.

Returns: 4D weight array where the first 2 dimensions correspond to the
grid of where the tiles are relative to the tile being interpolated.

satpy.composites.viirs.**_check_moon_phase**(*moon_datasets: list[DataArray]*, *start_time: datetime*) → float

Check if we have Moon phase as an input dataset and, if not, calculate it.

[1] Stephan Zinke (2017),
A simplified high and near-constant contrast approach for the display of VIIRS day/night band imagery DOI:10.1080/01431161.2017.1338838

satpy.composites.viirs.**_compute_tile_dist_and_bin_info**(*data: ndarray, valid_data_mask: ndarray, std_mult_cutoff: float, do_log_scale: bool, log_offset: float, clip_limit: float, slope_limit: float, number_of_bins: int, row_tiles: int, col_tiles: int, tile_size: int*)

satpy.composites.viirs.**_get_cumul_bin_info_for_tile**(*num_row_tile, weight_row, num_col_tile, weight_col, all_cumulative_dist_functions, all_bin_information*)

satpy.composites.viirs.**_histogram_equalization_helper**(*valid_data, number_of_bins, clip_limit=None, slope_limit=None*)

Calculate the simplest possible histogram equalization, using only valid data.

Returns
cumulative distribution function and bin information

satpy.composites.viirs.**_histogram_equalize_one_tile**(*data, valid_data_mask, std_mult_cutoff, do_log_scale, log_offset, clip_limit, slope_limit, number_of_bins, num_row_tile, num_col_tile, tile_size*)

satpy.composites.viirs.**_interpolate_local_equalized_tiles**(*data, out, mask_to_equalize, valid_data_mask, do_log_scale, log_offset, tile_weights, all_bin_information, all_cumulative_dist_functions, row_idx, col_idx, tile_size*)

satpy.composites.viirs.**_linear_normalization_from_0to1**(*data, mask, theoretical_max, theoretical_min=0, message='normalizing equalized data to fit in 0 to 1 range'*)

Do a linear normalization so all data is in the 0 to 1 range.

This is a sloppy but fast calculation that relies on parameters giving it the correct theoretical current max and min so it can scale the data accordingly.

satpy.composites.viirs.**histogram_equalization**(*data, mask_to_equalize, number_of_bins=1000, std_mult_cutoff=4.0, do_zerotoone_normalization=True, out=None*)

Perform a histogram equalization on the data.

Data is selected by the mask_to_equalize mask. The data will be separated into number_of_bins levels for equalization and outliers beyond +/- std_mult_cutoff*std will be ignored.

If do_zerotoone_normalization is True the data selected by mask_to_equalize will be returned in the 0 to 1 range. Otherwise the data selected by mask_to_equalize will be returned in the 0 to number_of_bins range.

Note: the data will be changed in place.

satpy.composites.viirs.**local_histogram_equalization**(*data, mask_to_equalize, valid_data_mask=None, number_of_bins=1000, std_mult_cutoff=3.0, do_zerotoone_normalization=True, local_radius_px: int = 300, clip_limit=60.0, slope_limit=3.0, do_log_scale=True, log_offset=1e-05, out=None*)

Equalize the provided data (in the mask_to_equalize) using adaptive histogram equalization.

Tiles of width/height (2 * local_radius_px + 1) will be calculated and results for each pixel will be bilinearly interpolated from the nearest 4 tiles when pixels fall near the edge of the image (there is no adjacent tile) the resultant interpolated sum from the available tiles will be multiplied to account for the weight of any missing tiles:

```
pixel total interpolated value = pixel available interpolated value / (1 - missing␣
↪interpolation weight)
```

If `do_zerotoone_normalization` is True the data will be scaled so that all data in the mask_to_equalize falls between 0 and 1; otherwise the data in mask_to_equalize will all fall between 0 and number_of_bins.

Returns: The equalized data

satpy.composites.viirs.**make_day_night_masks**(*solarZenithAngle*, *good_mask*, *highAngleCutoff*, *lowAngleCutoff*, *stepsDegrees=None*)

Generate masks for day, night, and twilight regions.

Masks are created from the provided solar zenith angle data.

Optionally provide the highAngleCutoff and lowAngleCutoff that define the limits of the terminator region (if no cutoffs are given the DEFAULT_HIGH_ANGLE and DEFAULT_LOW_ANGLE will be used).

Optionally provide the stepsDegrees that define how many degrees each "mixed" mask in the terminator region should be (if no stepsDegrees is given, the whole terminator region will be one mask).

Module contents

Base classes for composite objects.

class satpy.composites.**BackgroundCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

A compositor that overlays one composite on top of another.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

_combine_metadata_with_mode_and_sensor(*foreground: DataArray*, *background: DataArray*) → dict

static _get_merged_image_data(*foreground: DataArray*, *background: DataArray*) → list[DataArray]

class satpy.composites.**CategoricalDataCompositor**(*name*, *lut=None*, ***kwargs*)

Bases: *CompositeBase*

Compositor used to recategorize categorical data using a look-up-table.

Each value in the data array will be recategorized to a new category defined in the look-up-table using the original value as an index for that look-up-table.

Example

data = [[1, 3, 2], [4, 2, 0]] lut = [10, 20, 30, 40, 50] res = [[20, 40, 30], [50, 30, 10]]

Get look-up-table used to recategorize data.

Parameters

lut (*list*) – a list of new categories. The lenght must be greater than the maximum value in the data array that should be recategorized.

static _getitem(*block*, *lut*)

_update_attrs(*new_attrs*)

Modify name and add LUT.

class satpy.composites.**CloudCompositor**(*name*, *transition_min=258.15*, *transition_max=298.15*, *transition_gamma=3.0*, ***kwargs*)

Bases: *GenericCompositor*

Detect clouds based on thresholding and use it as a mask for compositing.

Collect custom configuration values.

Parameters

- **transition_min** (*float*) – Values below or equal to this are clouds -> opaque white
- **transition_max** (*float*) – Values above this are cloud free -> transparent
- **transition_gamma** (*float*) – Gamma correction to apply at the end

class satpy.composites.**ColorizeCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *ColormapCompositor*

A compositor colorizing the data, interpolating the palette colors when needed.

> **Warning:** Deprecated since Satpy 0.39. See the *ColormapCompositor* docstring for documentation on the alternative.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

static _apply_colormap(*colormap*, *data*, *palette*)

class satpy.composites.**ColormapCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

A compositor that uses colormaps.

> **Warning:** Deprecated since Satpy 0.39.

This compositor is deprecated. To apply a colormap, use a *SingleBandCompositor* composite with a *colorize()* or *palettize()* enhancement instead. For example, to make a `cloud_top_height` composite based on a dataset `ctth_alti` palettized by `ctth_alti_pal`, the composite would be:

```
cloud_top_height:
  compositor: !!python/name:satpy.composites.SingleBandCompositor
  prerequisites:
  - ctth_alti
  tandard_name: cloud_top_height
```

and the enhancement:

```
cloud_top_height:
  standard_name: cloud_top_height
  operations:
  - name: palettize
    method: !!python/name:satpy.enhancements.palettize
    kwargs:
      palettes:
        - dataset: ctth_alti_pal
          color_scale: 255
          min_value: 0
          max_value: 255
```

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

_create_composite_from_channels(*channels*, *template*)

static _create_masked_dataarray_like(*array*, *template*, *mask*)

static _get_mask_from_data(*data*)

static build_colormap(*palette*, *dtype*, *info*)

Create the colormap from the *raw_palette* and the valid_range.

Colormaps come in different forms, but they are all supposed to have color values between 0 and 255. The following cases are considered:

- Palettes comprised of only a list of colors. If *dtype* is uint8, the values of the colormap are the enumeration of the colors. Otherwise, the colormap values will be spread evenly from the min to the max of the valid_range provided in *info*.
- Palettes that have a palette_meanings attribute. The palette meanings will be used as values of the colormap.

class satpy.composites.**CompositeBase**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: object

Base class for all compositors and modifiers.

A compositor in Satpy is a class that takes in zero or more input DataArrays and produces a new DataArray with its own identifier (name). The result of a compositor is typically a brand new "product" that represents something different than the inputs that went into the operation.

See the ModifierBase class for information on the similar concept of "modifiers".

Initialise the compositor.

apply_modifier_info(*origin*, *destination*)

Apply the modifier info from *origin* to *destination*.

check_geolocation(*data_arrays*)

Check that the geolocations of the *data_arrays* are compatible.

For the purpose of this method, "compatible" means:

- All arrays should have the same dimensions.
- Either all arrays should have an area, or none should.
- If all have an area, the areas should be all the same.

Parameters

data_arrays (*List[arrays]*) – Arrays to be checked

Raises

- *IncompatibleAreas* – If dimension or areas do not match.
- **ValueError** – If some, but not all data arrays lack an area attribute.

drop_coordinates(*data_arrays*)

Drop negligible non-dimensional coordinates.

Drops negligible coordinates if they do not correspond to any dimension. Negligible coordinates are defined in the *NEGLIGIBLE_COORDS* module attribute.

Parameters

data_arrays (*List[arrays]*) – Arrays to be checked

property id

Return the DataID of the object.

match_data_arrays(*data_arrays*)

Match data arrays so that they can be used together in a composite.

For the purpose of this method, "can be used together" means:

- All arrays should have the same dimensions.
- Either all arrays should have an area, or none should.
- If all have an area, the areas should be all the same.

In addition, negligible non-dimensional coordinates are dropped (see *drop_coordinates()*) and dask chunks are unified (see *satpy.utils.unify_chunks()*).

Parameters

data_arrays (*List[arrays]*) – Arrays to be checked

Returns

Arrays with negligible non-dimensional coordinates removed.

Return type

data_arrays (List[arrays])

Raises

- *IncompatibleAreas* – If dimension or areas do not match.
- **ValueError** – If some, but not all data arrays lack an area attribute.

class satpy.composites.**DayNightCompositor**(*name*, *lim_low=85.0*, *lim_high=88.0*, *day_night='day_night'*, *include_alpha=True*, ***kwargs*)

Bases: *GenericCompositor*

A compositor that blends day data with night data.

Using the *day_night* flag it is also possible to provide only a day product or only a night product and mask out (make transparent) the opposite portion of the image (night or day). See the documentation below for more details.

Collect custom configuration values.

Parameters

- **lim_low** (*float*) – lower limit of Sun zenith angle for the blending of the given channels
- **lim_high** (*float*) – upper limit of Sun zenith angle for the blending of the given channels
- **day_night** (*string*) – "day_night" means both day and night portions will be kept "day_only" means only day portion will be kept "night_only" means only night portion will be kept
- **include_alpha** (*bool*) – This only affects the "day only" or "night only" result. True means an alpha band will be added to the output image for transparency. False means the output is a single-band image with undesired pixels being masked out (replaced with NaNs).

_get_coszen_blending_weights(*projectables: Sequence[DataArray]*) → DataArray

_get_data_for_combined_product(*day_data*, *night_data*)

_get_data_for_single_side_product(*foreground_data: DataArray*, *weights: DataArray*) → tuple[DataArray, DataArray, DataArray]

_get_day_night_data_for_single_side_product(*foreground_data*)

_mask_weights(*weights*)

_mask_weights_with_data(*weights: DataArray*, *day_data: DataArray*, *night_data: DataArray*) → DataArray

_weight_data(*day_data: DataArray*, *night_data: DataArray*, *weights: DataArray*, *attrs: dict*) → list[DataArray]

class satpy.composites.**DifferenceCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *CompositeBase*

Make the difference of two data arrays.

Initialise the compositor.

class satpy.composites.**Filler**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Fix holes in projectable 1 with data from projectable 2.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**FillingCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Make a regular RGB, filling the RGB bands with the first provided dataset's values.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**GenericCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *CompositeBase*

Basic colored composite builder.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

_concat_datasets(*projectables*, *mode*)

_get_sensors(*projectables*)

classmethod infer_mode(*data_arr*)

Guess at the mode for a particular DataArray.

modes = {1: 'L', 2: 'LA', 3: 'RGB', 4: 'RGBA'}

exception satpy.composites.**IncompatibleAreas**

Bases: Exception

Error raised upon compositing things of different shapes.

exception satpy.composites.**IncompatibleTimes**

Bases: Exception

Error raised upon compositing things from different times.

class satpy.composites.**LongitudeMaskingCompositor**(*name*, *lon_min=None*, *lon_max=None*, ***kwargs*)

Bases: *SingleBandCompositor*

Masks areas outside defined longitudes.

Collect custom configuration values.

Parameters

- **lon_min** (*float*) – lower longitude limit
- **lon_max** (*float*) – upper longitude limit

class satpy.composites.**LuminanceSharpeningCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Create a high resolution composite by sharpening a low resolution using high resolution luminance.

This is done by converting to YCbCr colorspace, replacing Y, and convertin back to RGB.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**MaskingCompositor**(*name*, *transparency=None*, *conditions=None*, *mode='LA'*, ***kwargs*)

Bases: *GenericCompositor*

A compositor that masks e.g. IR 10.8 channel data using cloud products from NWC SAF.

Collect custom configuration values.

Kwargs:

transparency (dict): transparency for each cloud type as
key-value pairs in a dictionary. Will be converted to *conditions*. DEPRECATED.

conditions (list): list of three items determining the masking
settings.

mode (str, optional): Image mode to return. For single-band input,
this shall be "LA" (default) or "RGBA". For multi-band input, this argument is ignored as the result is always RGBA.

Each condition in *conditions* consists of three items:

- ***method*: Numpy method name. The following are supported**
  operations: *less*, *less_equal*, *equal*, *greater_equal*, *greater*, *not_equal*, *isnan*, *isfinite*, *isinf*, *isneginf*, or *isposinf*.
- ***value*: threshold value of the *mask* applied with the**
  operator. Can be a string, in which case the corresponding value will be determined from *flag_meanings* and *flag_values* attributes of the mask. NOTE: the *value* should not be given to 'is*` methods.
- ***transparency*: transparency from interval [0 … 100] used**
  for the method/threshold. Value of 100 is fully transparent.

Example:

```
>>> conditions = [{'method': 'greater_equal', 'value': 0,
                   'transparency': 100},
                  {'method': 'greater_equal', 'value': 1,
                   'transparency': 80},
                  {'method': 'greater_equal', 'value': 2,
                   'transparency': 0},
                  {'method': 'isnan',
                   'transparency': 100}]
>>> compositor = MaskingCompositor("masking compositor",
                                   transparency=transparency)
>>> result = compositor([data, mask])
```

This will set transparency of *data* based on the values in the *mask* dataset. Locations where *mask* has values of *0* will be fully transparent, locations with *1* will be semi-transparent and locations with *2* will be fully visible in the resulting image. In the end all *NaN* areas in the mask are set to full transparency. All the unlisted locations will be visible.

The transparency is implemented by adding an alpha layer to the composite. The locations with transparency of *100* will be set to NaN in the data. If the input *data* contains an alpha channel, it will be discarded.

_get_alpha_bands(*data*, *mask_in*, *alpha_attrs*)

Get alpha bands.

From input data, masks, and attributes, get alpha band.

_get_mask(*method*, *value*, *mask_data*)

Get mask array from *mask_data* using *method* and threshold *value*.

The *method* is the name of a numpy function.

_select_data_bands(*data_in*)

Select data to be composited from input data.

From input data, select the bands that need to have masking applied.

_set_data_nans(*data*, *mask*, *attrs*)

Set *data* to nans where *mask* is True.

The attributes *attrs** will be written to each band in *data*.

_supported_modes = {'LA', 'RGBA'}

class satpy.composites.**MultiFiller**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *SingleBandCompositor*

Fix holes in projectable 1 with data from the next projectables.

Initialise the compositor.

satpy.composites.**NEGLIGIBLE_COORDS = ['time']**

Keywords identifying non-dimensional coordinates to be ignored during composite generation.

class satpy.composites.**NaturalEnh**(*name*, *ch16_w=1.3*, *ch08_w=2.5*, *ch06_w=2.2*, **args*, ***kwargs*)

Bases: *GenericCompositor*

Enhanced version of natural color composite by Simon Proud.

Parameters

- **ch16_w** (*float*) – weight for red channel (1.6 um). Default: 1.3
- **ch08_w** (*float*) – weight for green channel (0.8 um). Default: 2.5
- **ch06_w** (*float*) – weight for blue channel (0.6 um). Default: 2.2

Initialize the class.

class satpy.composites.**PaletteCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *ColormapCompositor*

A compositor colorizing the data, not interpolating the palette colors.

> **Warning:** Deprecated since Satpy 0.39. See the *ColormapCompositor* docstring for documentation on the alternative.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

static **_apply_colormap**(*colormap*, *data*, *palette*)

class satpy.composites.**RGBCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Make a composite from three color bands (deprecated).

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**RatioCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *CompositeBase*

Make the ratio of two data arrays.

Initialise the compositor.

class satpy.composites.**RatioSharpenedRGB**(**args*, ***kwargs*)

Bases: *GenericCompositor*

Sharpen RGB bands with ratio of a high resolution band to a lower resolution version.

Any pixels where the ratio is computed to be negative or infinity, it is reset to 1. Additionally, the ratio is limited to 1.5 on the high end to avoid high changes due to small discrepancies in instrument detector footprint. Note that the input data to this compositor must already be resampled so all data arrays are the same shape.

Example:

```
R_lo -  1000m resolution - shape=(2000, 2000)
G - 1000m resolution - shape=(2000, 2000)
B - 1000m resolution - shape=(2000, 2000)
R_hi -  500m resolution - shape=(4000, 4000)

ratio = R_hi / R_lo
new_R = R_hi
new_G = G * ratio
new_B = B * ratio
```

In some cases, there could be multiple high resolution bands:

```
R_lo -  1000m resolution - shape=(2000, 2000)
G_hi - 500m resolution - shape=(4000, 4000)
B - 1000m resolution - shape=(2000, 2000)
R_hi -  500m resolution - shape=(4000, 4000)
```

To avoid the green band getting involved in calculating ratio or sharpening, add "neutral_resolution_band: green" in the YAML config file. This way only the blue band will get sharpened:

```
ratio = R_hi / R_lo
new_R = R_hi
new_G = G_hi
new_B = B * ratio
```

Instanciate the ration sharpener.

_combined_sharpened_info(*info*, *new_attrs*)

_get_and_sharpen_rgb_data_arrays_and_meta(*datasets*, *optional_datasets*)

_sharpen_bands_with_high_res(*bands*, *high_res*)

class satpy.composites.**RealisticColors**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Create a realistic colours composite for SEVIRI.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**SandwichCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Make a sandwich product.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.composites.**SelfSharpenedRGB**(**args*, ***kwargs*)

Bases: *RatioSharpenedRGB*

Sharpen RGB with ratio of a band with a strided-version of itself.

Example:

```
R -  500m resolution - shape=(4000, 4000)
G - 1000m resolution - shape=(2000, 2000)
B - 1000m resolution - shape=(2000, 2000)

ratio = R / four_element_average(R)
new_R = R
new_G = G * ratio
new_B = B * ratio
```

Instanciate the ration sharpener.

static four_element_average_dask(*d*)

Average every 4 elements (2x2) in a 2D array.

class satpy.composites.**SingleBandCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *CompositeBase*

Basic single-band composite builder.

This preserves all the attributes of the dataset it is derived from.

Initialise the compositor.

static _update_missing_metadata(*existing_attrs*, *new_attrs*)

class satpy.composites.**StaticImageCompositor**(*name*, *filename=None*, *url=None*, *known_hash=None*, *area=None*, ***kwargs*)

Bases: *GenericCompositor*, *DataDownloadMixin*

A compositor that loads a static image from disk.

Environment variables in the filename are automatically expanded.

Collect custom configuration values.

Parameters

- **filename** (*str*) – Name to use when storing and referring to the file in the `data_dir` cache. If `url` is provided (preferred), then this is used as the filename in the cache and will be appended to `<data_dir>/composites/<class_name>/`. If `url` is provided and `filename` is not then the `filename` will be guessed from the `url`. If `url` is not provided, then it is assumed `filename` refers to a local file. If the `filename` does not come with an absolute path, `data_dir` will be used as the directory path. Environment variables are expanded.
- **url** (*str*) – URL to remote file. When the composite is created the file will be downloaded and cached in Satpy's `data_dir`. Environment variables are expanded.
- **known_hash** (*str or None*) – Hash of the remote file used to verify a successful download. If not provided then the download will not be verified. See *satpy.aux_download.register_file()* for more information.
- **area** (*str*) – Name of area definition for the image. Optional for images with built-in area definitions (geotiff).

Use cases:

1. url + no filename: Satpy determines the filename based on the filename in the URL, then downloads the URL, and saves it to <data_dir>/<filename>. If the file already exists and known_hash is also provided, then the pooch library compares the hash of the file to the known_hash. If it does not match, then the URL is re-downloaded. If it matches then no download.
2. url + relative filename: Same as case 1 but filename is already provided so download goes to <data_dir>/<filename>. Same hashing behavior. This does not check for an absolute path.
3. No url + absolute filename: No download, filename is passed directly to generic_image reader. No hashing is done.
4. No url + relative filename: Check if <data_dir>/<filename> exists. If it does then make filename an absolute path. If it doesn't, then keep it as is and let the exception at the bottom of the method get raised.

static _check_relative_filename(*filename*)

_get_cache_filename_and_url(*filename*, *url*)

_retrieve_data_file()

register_data_files(*data_files*)

Tell Satpy about files we may want to download.

class satpy.composites.**SumCompositor**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *CompositeBase*

Make the sum of two data arrays.

Initialise the compositor.

satpy.composites.**_apply_palette_to_image**(*img*)

satpy.composites.**_get_band_names**(*day_data*, *night_data*)

satpy.composites.**_get_data_from_enhanced_image**(*dset*, *convert_p*)

satpy.composites.**_get_flag_value**(*mask*, *val*)

Get a numerical value of the named flag.

This function assumes the naming used in product generated with NWC SAF GEO/PPS softwares.

satpy.composites.**_get_sharpening_ratio**(*high_res*, *low_res*)

satpy.composites.**_get_single_band_data**(*data*, *band*)

satpy.composites.**_get_single_channel**(*data: DataArray*) → DataArray

satpy.composites.**_get_weight_mask_for_daynight_product**(*weights*, *data_a*, *data_b*)

satpy.composites.**_get_weight_mask_for_single_side_product**(*data_a*, *data_b*)

satpy.composites.**_insert_palette_colors**(*channels*, *palette*)

satpy.composites.**_mean4**(*data*, *offset=(0, 0)*, *block_id=None*)

satpy.composites.**add_alpha_bands**(*data*)

Only used for DayNightCompositor.

Add an alpha band to L or RGB composite as prerequisites for the following band matching to make the masked-out area transparent.

satpy.composites.**add_bands**(*data*, *bands*)

Add bands so that they match *bands*.

satpy.composites.**check_times**(*projectables*)

Check that *projectables* have compatible times.

satpy.composites.**enhance2dataset**(*dset*, *convert_p=False*)

Return the enhancement dataset *dset* as an array.

If *convert_p* is True, enhancements generating a P mode will be converted to RGB or RGBA.

satpy.composites.**sub_arrays**(*proj1*, *proj2*)

Substract two DataArrays and combine their attrs.

satpy.composites.**zero_missing_data**(*data1*, *data2*)

Replace NaN values with zeros in data1 if the data is valid in data2.

satpy.dataset package

Submodules

satpy.dataset.anc_vars module

Utilities for dealing with ancillary variables.

satpy.dataset.anc_vars.**dataset_walker**(*datasets*)

Walk through *datasets* and their ancillary data.

Yields datasets and their parent.

satpy.dataset.anc_vars.**replace_anc**(*dataset*, *parent_dataset*)

Replace *dataset* the *parent_dataset*'s *ancillary_variables* field.

satpy.dataset.data_dict module

Classes and functions related to a dictionary with DataID keys.

class satpy.dataset.data_dict.**DatasetDict**

Bases: dict

Special dictionary object that can handle dict operations based on dataset name, wavelength, or DataID.

Note: Internal dictionary keys are *DataID* objects.

_create_dataid_key(*key*, *value_info*)

Create a DataID key from dictionary.

_create_id_keys_from_dict(*value_info_dict*)

Create id_keys from dict.

contains(*item*)

Check contains when we know the *exact* DataID.

get(*key*, *default=None*)

Get value with optional default.

get_key(*match_key*, *num_results=1*, *best=True*, ***dfilter*)

Get multiple fully-specified keys that match the provided query.

Parameters

- **key** (DataID) – DataID of query parameters to use for searching. Any parameter that is *None* is considered a wild card and any match is accepted. Can also be a string representing the dataset name or a number representing the dataset wavelength.
- **num_results** (*int*) – Number of results to return. If *0* return all, if *1* return only that element, otherwise return a list of matching keys.
- ****dfilter** (*dict*) – See *get_key* function for more information.

getitem(*item*)

Get Node when we know the *exact* DataID.

keys(*names=False*, *wavelengths=False*)

Give currently contained keys.

exception satpy.dataset.data_dict.**TooManyResults**

Bases: KeyError

Special exception when one key maps to multiple items in the container.

satpy.dataset.data_dict.**get_best_dataset_key**(*key*, *choices*)

Choose the "best" *DataID* from *choices* based on *key*.

To see how the keys are sorted, refer to *:meth:satpy.datasets.DataQuery.sort_dataids*.

This function assumes *choices* has already been filtered to only include datasets that match the provided *key*.

Parameters

- **key** (DataQuery) – Query parameters to sort *choices* by.
- **choices** (*iterable*) – *DataID* objects to sort through to determine the best dataset.

Returns: List of best *DataID`s from `choices*. If there is more
than one element this function could not choose between the available datasets.

satpy.dataset.data_dict.**get_key**(*key*, *key_container*, *num_results=1*, *best=True*, *query=None*, ***kwargs*)

Get the fully-specified key best matching the provided key.

Only the best match is returned if *best* is *True* (default). See *get_best_dataset_key* for more information on how this is determined.

query is provided as a convenience to filter by multiple parameters at once without having to filter by multiple *key* inputs.

Parameters

- **key** (DataID) – DataID of query parameters to use for searching. Any parameter that is *None* is considered a wild card and any match is accepted.
- **key_container** (*dict or set*) – Container of DataID objects that uses hashing to quickly access items.
- **num_results** (*int*) – Number of results to return. Use *0* for all matching results. If *1* then the single matching key is returned instead of a list of length 1. (default: 1)
- **best** (*bool*) – Sort results to get "best" result first (default: True). See *get_best_dataset_key* for details.
- **query** (DataQuery) – filter for the key which can contain for example:

  resolution (float, int, or list): Resolution of the dataset in
  dataset units (typically meters). This can also be a list of these numbers.

  calibration (str or list): Dataset calibration
  (ex.'reflectance'). This can also be a list of these strings.

  polarization (str or list): Dataset polarization
  (ex.'V'). This can also be a list of these strings.

  level (number or list): Dataset level (ex. 100). This can also be a
  list of these numbers.

  modifiers (list): Modifiers applied to the dataset. Unlike
  resolution and calibration this is the exact desired list of modifiers for one dataset, not a list of possible modifiers.

Returns
Matching key(s)

Return type
list or *DataID*

Raises: KeyError if no matching results or if more than one result is
found when *num_results* is *1*.

satpy.dataset.dataid module

Dataset identifying objects.

class satpy.dataset.dataid.**DataID**(*id_keys*, ***keyval_dict*)

Bases: dict

Identifier for all *DataArray* objects.

DataID is a dict that holds identifying and classifying information about a DataArray.

Init the DataID.

The *id_keys* dictionary has to be formed as described in *Satpy internal workings: having a look under the hood*. The other keyword arguments are values to be assigned to the keys. Note that *None* isn't a valid value and will simply be ignored.

_asdict()

_find_modifiers_key()

_immutable(**args*, ***kws*) → NoReturn

Raise and error.

_replace(***kwargs*)

Make a new instance with replaced items.

classmethod _unpickle(*id_keys*, *keyval*)

Create a new instance of the DataID after pickling.

clear(**args*, ***kws*) → NoReturn

Raise and error.

convert_dict(*keyvals*)

Convert a dictionary's values to the types defined in this object's id_keys.

create_filter_query_without_required_fields(*query*)

Remove the required fields from *query*.

create_less_modified_query()

Create a query with one less modifier.

static fix_id_keys(*id_keys*)

Flesh out enums in the id keys as gotten from a config.

classmethod from_dataarray(*array*, *default_keys={'name': {'required': True}, 'resolution': {'transitive': True}}*)

Get the DataID using the dataarray attributes.

from_dict(*keyvals*)

Create a DataID from a dictionary.

property id_keys

Get the id_keys.

is_modified()

Check if this is modified.

classmethod new_id_from_dataarray(*array*, *default_keys={'name': {'required': True}, 'resolution': {'transitive': True}}*)

Create a new DataID from a dataarray's attributes.

pop(**args*, ***kws*) → NoReturn

Raise and error.

popitem(**args*, ***kws*) → NoReturn

Raise and error.

setdefault(**args*, ***kws*) → NoReturn

Raise and error.

to_dict()

Convert the ID to a dict.

update(**args*, ***kws*) → NoReturn

Raise and error.

class satpy.dataset.dataid.**DataQuery**(***kwargs*)

Bases: object

The data query object.

A DataQuery can be used in Satpy to query for a Dataset. This way a fully qualified DataID can be found even if some DataID elements are unknown. In this case a * signifies something that is unknown or not applicable to the requested Dataset.

Initialize the query.

static _add_absolute_distance(*dataid*, *key*, *distance*)

static _add_distance_from_query(*dataid_val*, *requested_val*, *distance*)

_asdict()

_match_dataid(*dataid*)

Match the dataid with the current query.

_match_query_value(*key*, *id_val*)

_shares_required_keys(*dataid*)

Check if dataid shares required keys with the current query.

_to_trimmed_dict()

create_less_modified_query()

Create a query with one less modifier.

filter_dataids(*dataid_container*)

Filter DataIDs based on this query.

classmethod from_dict(*the_dict*)

Convert a dict to an ID.

get(*key*, *default=None*)

Get an item.

is_modified()

Check if this is modified.

items()

Get the items of this query.

sort_dataids(*dataids*)

Sort the DataIDs based on this query.

Returns the sorted dataids and the list of distances.

The sorting is performed based on the types of the keys to search on (as they are defined in the DataIDs from *dataids*). If that type defines a *distance* method, then it is used to find how 'far' the DataID is from the current query. If the type is a number, a simple subtraction is performed. For other types, the distance is 0 if the values are identical, np.inf otherwise.

For example, with the default DataID, we use the following criteria:

1. Central wavelength is nearest to the *key* wavelength if specified.
2. Least modified dataset if *modifiers* is *None* in *key*. Otherwise, the modifiers are ignored.
3. Highest calibration if *calibration* is *None* in *key*. Calibration priority is the order of the calibration list defined as reflectance, brightness temperature, radiance counts if not overridden in the reader configuration.
4. Best resolution (smallest number) if *resolution* is *None* in *key*. Otherwise, the resolution is ignored.

sort_dataids_with_preference(*all_ids*, *preference*)

Sort *all_ids* given a sorting *preference* (DataQuery or None).

to_dict(*trim=True*)

Convert the ID to a dict.

class satpy.dataset.dataid.**ModifierTuple**(*iterable=()*, /)

Bases: tuple

A tuple holder for modifiers.

classmethod convert(*modifiers*)

Convert *modifiers* to this type if possible.

class satpy.dataset.dataid.**ValueList**(*value*)

Bases: IntEnum

A static value list.

This class is meant to be used for dynamically created Enums. Due to this it should not be used as a normal Enum class or there may be some unexpected behavior. For example, this class contains custom pickling and unpickling handling that may break in subclasses.

classmethod _unpickle(*enum_name*, *enum_members*, *enum_member*)

Create dynamic class that was previously pickled.

See `__reduce_ex__()` for implementation details.

classmethod convert(*value*)

Convert value to an instance of this class.

class satpy.dataset.dataid.**WavelengthRange**(*min*, *central*, *max*, *unit='µm'*)

Bases: *WavelengthRange*

A named tuple for wavelength ranges.

The elements of the range are min, central and max values, and optionally a unit (defaults to µm). No clever unit conversion is done here, it's just used for checking that two ranges are comparable.

Create new instance of WavelengthRange(min, central, max, unit)

classmethod _read_cf_from_string_export(*blob*)

Read blob as a string created by *to_cf*.

classmethod _read_cf_from_string_list(*blob*)

Read blob as a list of strings (legacy formatting).

classmethod convert(*wl*)

Convert *wl* to this type if possible.

distance(*value*)

Get the distance from value.

classmethod from_cf(*blob*)

Return a WavelengthRange from a cf blob.

to_cf()

Serialize for cf export.

satpy.dataset.dataid.**_create_id_dict_from_any_key**(*dataset_key*)

satpy.dataset.dataid.**_generalize_value_for_comparison**(*val*)

Get a generalize value for comparisons.

satpy.dataset.dataid.**_update_dict_with_filter_query**(*ds_dict*, *filter_query*)

satpy.dataset.dataid.**create_filtered_query**(*dataset_key*, *filter_query*)

Create a DataQuery matching *dataset_key* and *filter_query*.

If a property is specified in both *dataset_key* and *filter_query*, the former has priority.

satpy.dataset.dataid.**default_co_keys_config = {'name': {'required': True}, 'resolution': {'transitive': True}}**

Default ID keys for coordinate DataArrays.

satpy.dataset.dataid.**default_id_keys_config = {'calibration': {'enum': ['reflectance', 'brightness_temperature', 'radiance', 'radiance_wavenumber', 'counts'], 'transitive': True}, 'modifiers': {'default': (), 'type': <class 'satpy.dataset.dataid.ModifierTuple'>}, 'name': {'required': True}, 'resolution': {'transitive': False}, 'wavelength': {'type': <class 'satpy.dataset.dataid.WavelengthRange'>}}**

Default ID keys DataArrays.

satpy.dataset.dataid.**get_keys_from_config**(*common_id_keys*, *config*)

Gather keys for a new DataID from the ones available in configured dataset.

satpy.dataset.dataid.**minimal_default_keys_config = {'name': {'required': True}, 'resolution': {'transitive': True}}**

Minimal ID keys for DataArrays, for example composites.

satpy.dataset.dataid.**wlklass**

alias of *WavelengthRange*

satpy.dataset.metadata module

Utilities for merging metadata from various sources.

satpy.dataset.metadata.**_all_arrays_equal**(*arrays*)

Check if the arrays are equal.

If the arrays are lazy, just check if they have the same identity.

satpy.dataset.metadata.**_all_close**(*values*)

satpy.dataset.metadata.**_all_dicts_equal**(*dicts*)

satpy.dataset.metadata.**_all_equal**(*values*)

satpy.dataset.metadata.**_all_identical**(*values*)

Check that the identities of all values are the same.

satpy.dataset.metadata.**_all_list_of_arrays_equal**(*array_lists*)

Check that the lists of arrays are equal.

satpy.dataset.metadata.**_all_non_dicts_equal**(*values*)

satpy.dataset.metadata.**_all_values_equal**(*values*)

satpy.dataset.metadata.**_are_values_combinable**(*values*)

Check if the *values* can be combined.

satpy.dataset.metadata.**_combine_shared_info**(*shared_keys*, *info_dicts*, *average_times*)

satpy.dataset.metadata.**_contain_arrays**(*values*)

satpy.dataset.metadata.**_contain_collections_of_arrays**(*values*)

satpy.dataset.metadata.**_contain_dicts**(*values*)

satpy.dataset.metadata.**_dict_equal**(*d1*, *d2*)

Check that two dictionaries are equal.

Nested dictionaries are flattened to facilitate comparison.

satpy.dataset.metadata.**_dict_keys_equal**(*d1*, *d2*)

satpy.dataset.metadata.**_get_valid_dicts**(*metadata_objects*)

Get the valid dictionaries matching the metadata_objects.

satpy.dataset.metadata.**_is_all_arrays**(*value*)

satpy.dataset.metadata.**_is_array**(*val*)

Check if val is an array.

satpy.dataset.metadata.**_is_equal**(*a*, *b*, *comp_func*)

satpy.dataset.metadata.**_is_non_empty_collection**(*value*)

satpy.dataset.metadata.**_pairwise_all**(*func*, *values*)

satpy.dataset.metadata.**_shared_keys**(*info_dicts*)

satpy.dataset.metadata.**average_datetimes**(*datetime_list*)

Average a series of datetime objects.

Note: This function assumes all datetime objects are naive and in the same time zone (UTC).

Parameters
datetime_list (*iterable*) – Datetime objects to average

Returns: Average datetime as a datetime object

satpy.dataset.metadata.**combine_metadata**(**metadata_objects*, *average_times=True*)

Combine the metadata of two or more Datasets.

If the values corresponding to any keys are not equal or do not exist in all provided dictionaries then they are not included in the returned dictionary. By default any keys with the word 'time' in them and consisting of datetime objects will be averaged. This is to handle cases where data were observed at almost the same time but not exactly. In the interest of time, lazy arrays are compared by object identity rather than by their contents.

Parameters

- ***metadata_objects** – MetadataObject or dict objects to combine
- **average_times** (*bool*) – Average any keys with 'time' in the name

Returns
the combined metadata

Return type
dict

Module contents

Classes and functions related to data identification and querying.

satpy.demo package

Submodules

satpy.demo._google_cloud_platform module

satpy.demo._google_cloud_platform.**_download_gcs_files**(*globbed_files*, *fs*, *base_dir*, *force*)

satpy.demo._google_cloud_platform.**get_bucket_files**(*glob_pattern*, *base_dir*, *force=False*, *pattern_slice=None*)

Download files from Google Cloud Storage.

Parameters

- **glob_pattern** (*str or list*) – Glob pattern string or series of patterns used to search for on Google Cloud Storage. The pattern should include the "gs://" protocol prefix. If a list of lists, then the results of each sublist pattern are concatenated and the result is treated as

one pattern result. This is important for things like `pattern_slice` and complicated glob patterns not supported by GCP.

- **base_dir** (*str*) – Root directory to place downloaded files on the local system.
- **force** (*bool*) – Force re-download of data regardless of its existence on the local system. Warning: May delete non-demo files stored in download directory.
- **pattern_slice** (*slice*) – Slice object to limit the number of files returned by each glob pattern.

`satpy.demo._google_cloud_platform.is_google_cloud_instance()`

Check if we are on a GCP virtual machine.

satpy.demo.abi_l1b module

Demo data download helper functions for ABI L1b data.

`satpy.demo.abi_l1b.get_hurricane_florence_abi(base_dir=None, method=None, force=False, channels=None, num_frames=10)`

Get GOES-16 ABI (Meso sector) data from 2018-09-11 13:00Z to 17:00Z.

Parameters

- **base_dir** (*str*) – Base directory for downloaded files.
- **method** (*str*) – Force download method for the data if not already cached. Allowed options are: 'gcsfs'. Default of `None` will choose the best method based on environment settings.
- **force** (*bool*) – Force re-download of data regardless of its existence on the local system. Warning: May delete non-demo files stored in download directory.
- **channels** (*list*) – Channels to include in download. Defaults to all 16 channels.
- **num_frames** (*int or slice*) – Number of frames to download. Maximum 240 frames. Default 10 frames.

Size per frame (all channels): ~15MB

Total size (default 10 frames, all channels): ~124MB

Total size (240 frames, all channels): ~3.5GB

`satpy.demo.abi_l1b.get_us_midlatitude_cyclone_abi(base_dir=None, method=None, force=False)`

Get GOES-16 ABI (CONUS sector) data from 2019-03-14 00:00Z.

Parameters

- **base_dir** (*str*) – Base directory for downloaded files.
- **method** (*str*) – Force download method for the data if not already cached. Allowed options are: 'gcsfs'. Default of `None` will choose the best method based on environment settings.
- **force** (*bool*) – Force re-download of data regardless of its existence on the local system. Warning: May delete non-demo files stored in download directory.

Total size: ~110MB

satpy.demo.ahi_hsd module

Demo data download helper functions for AHI HSD data.

satpy.demo.ahi_hsd.**download_typhoon_surigae_ahi**(*base_dir=None*, *channels=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)*, *segments=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)*)

Download Himawari 8 data.

This scene shows the Typhoon Surigae.

satpy.demo.fci module

Demo FCI data download.

satpy.demo.fci.**_unpack_tarfile_to**(*filename*, *subdir*)

Unpack content of tarfile in filename to subdir.

satpy.demo.fci.**download_fci_test_data**(*base_dir=None*)

Download FCI test data.

Download the nominal FCI test data from July 2020.

satpy.demo.fci.**get_fci_test_data_dir**(*base_dir=None*)

Get directory for FCI test data.

satpy.demo.seviri_hrit module

Demo data download for SEVIRI HRIT files.

satpy.demo.seviri_hrit.**_create_full_set**()

Create the full set dictionary.

satpy.demo.seviri_hrit.**_generate_filenames**(*pattern*, *channel*, *segments*)

Generate the filenames for *channel* and *segments*.

satpy.demo.seviri_hrit.**download_seviri_hrit_20180228_1500**(*base_dir=None*, *subset=None*)

Download the SEVIRI HRIT files for 2018-02-28T15:00.

subset is a dictionary with the channels as keys and granules to download as values, eg:

```
{"HRV": [1, 2, 3], "IR_108": [1, 2], "EPI": None}
```

satpy.demo.seviri_hrit.**generate_subset_of_filenames**(*subset=None*, *base_dir=''*)

Generate SEVIRI HRIT filenames.

satpy.demo.utils module

Utilities for demo data download.

satpy.demo.utils.**download_url**(*source*, *target*)

Download a url in stream mode.

satpy.demo.viirs_sdr module

Demo data download for VIIRS SDR HDF5 files.

satpy.demo.viirs_sdr.**_get_filenames_to_download**(*channels*, *granules*)

satpy.demo.viirs_sdr.**_yield_specific_granules**(*filenames*, *granules*)

satpy.demo.viirs_sdr.**get_viirs_sdr_20170128_1229**(*base_dir=None*, *channels=('I01', 'I02', 'I03', 'I04', 'I05', 'M01', 'M02', 'M03', 'M04', 'M05', 'M06', 'M07', 'M08', 'M09', 'M10', 'M11', 'M12', 'M13', 'M14', 'M15', 'M16', 'DNB')*, *granules=(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)*)

Get VIIRS SDR files for 2017-01-28 12:29 to 12:43.

These files are downloaded from Zenodo. You can see the full file listing here: https://zenodo.org/record/263296

Specific channels can be specified with the `channels` keyword argument. By default, all channels (all I bands, M bands, and DNB bands) will be downloaded. Channels are referred to by their band type and channel number (ex. "I01" or "M16" or "DNB"). Terrain-corrected geolocation files are always downloaded when the corresponding band data is specified.

The `granules` argument will control which granules ("time steps") are downloaded. There are 10 available and the keyword argument can be specified as a tuple of integers from 1 to 10.

This full dataset is ~10.1GB.

Notes

File list was retrieved using the zenodo API.

```
import requests
viirs_listing = requests.get("https://zenodo.org/api/records/263296")
viirs_dict = json.loads(viirs_listing.content)
print("\n".join(sorted(x['links']['self'] for x in viirs_dict['files'])))
```

Module contents

Demo data download helper functions.

Each `get_*` function below downloads files to a local directory and returns a list of paths to those files. Some (not all) functions have multiple options for how the data is downloaded (via the `method` keyword argument) including:

- **gcsfs:**
  Download data from a public google cloud storage bucket using the `gcsfs` package.

- **unidata_thredds:**
  Access data using OpenDAP or similar method from Unidata's public THREDDS server (https://thredds.unidata.ucar.edu/thredds/catalog.html).
- **uwaos_thredds:**
  Access data using OpenDAP or similar method from the University of Wisconsin - Madison's AOS department's THREDDS server.
- **http:**
  A last resort download method when nothing else is available of a tarball or zip file from one or more servers available to the Satpy project.
- **uw_arcdata:**
  A network mount available on many servers at the Space Science and Engineering Center (SSEC) at the University of Wisconsin - Madison. This is method is mainly meant when tutorials are taught at the SSEC using a Jupyter Hub server.

To use these functions, do:

```
>>> from satpy import Scene, demo
>>> filenames = demo.get_us_midlatitude_cyclone_abi()
>>> scn = Scene(reader='abi_l1b', filenames=filenames)
```

satpy.enhancements package

Submodules

satpy.enhancements.abi module

Enhancement functions specific to the ABI sensor.

satpy.enhancements.abi.**_cimss_true_color_contrast**(*img_data*)

Perform per-chunk enhancement.

Code ported from Kaba Bah's AWIPS python plugin for creating the CIMSS Natural (True) Color image in AWIPS. AWIPS provides that python code the image data on a 0-255 scale. Satpy gives this function the data on a 0-1.0 scale (assuming linear stretching and sqrt enhancements have already been applied).

satpy.enhancements.abi.**cimss_true_color_contrast**(*img*, ***kwargs*)

Scale data based on CIMSS True Color recipe for AWIPS.

satpy.enhancements.atmosphere module

Enhancements related to visualising atmospheric phenomena.

satpy.enhancements.atmosphere.**_calc_essl_blue**(*ratio*)

Calculate values for blue based on scaled and clipped ratio.

satpy.enhancements.atmosphere.**_calc_essl_green**(*ratio*)

Calculate values for green based on scaled and clipped ratio.

satpy.enhancements.atmosphere.**_calc_essl_red**(*ratio*)

Calculate values for red based on scaled and clipped ratio.

satpy.enhancements.atmosphere.**_is_fci_test_data**(*data*)

Check if we are working with FCI test data.

satpy.enhancements.atmosphere.**_scale_and_clip**(*ratio*, *low*, *high*)

Scale ratio values to [0, 1] and clip values outside this range.

satpy.enhancements.atmosphere.**essl_moisture**(*img*, *low=1.1*, *high=1.6*) → None

Low level moisture by European Severe Storms Laboratory (ESSL).

Expects a mode L image with data corresponding to the ratio of the calibrated reflectances for the 0.86 µm and 0.906 µm channel.

This composite and its colorisation were developed by ESSL.

Ratio values are scaled from the range `[low, high]`, which is by default between 1.1 and 1.6, but might be tuned based on region or sensor, to `[0, 1]`. Values outside this range are clipped. Color values for red, green, and blue are calculated as follows, where `x` is the ratio between the 0.86 µm and 0.905 µm channels:

$$R = \max(1.375 - 2.67x, -0.75 + x)$$
$$G = 1 - \frac{8x}{7}$$
$$B = \max(0.75 - 1.5x, 0.25 - (x - 0.75)^2)$$

The value of `img.data` is modified in-place.

A color interpretation guide is pending further adjustments to the parameters for current and future sensors.

Parameters

- **img** – XRImage containing the relevant composite
- **low** – optional, low end for scaling, defaults to 1.1
- **high** – optional, high end for scaling, defaults to 1.6

satpy.enhancements.mimic module

Mimic TPW Color enhancements.

satpy.enhancements.mimic.**nrl_colors**(*img*, ***kwargs*)

TPW color table based on NRL Color table (0-76 mm).

satpy.enhancements.mimic.**total_precipitable_water**(*img*, ***kwargs*)

Palettizes images from MIMIC TPW data.

This modifies the image's data so the correct colors can be applied to it, and then palettizes the image.

satpy.enhancements.viirs module

Enhancements specific to the VIIRS instrument.

satpy.enhancements.viirs.**_water_detection**(*img_data*)

satpy.enhancements.viirs.**water_detection**(*img*, ***kwargs*)

Palettizes images from VIIRS flood data.

This modifies the image's data so the correct colors can be applied to it, and then palettizes the image.

Module contents

Enhancements.

satpy.enhancements.**_bt_threshold**(*band_data*, *threshold*, *high_coeffs*, *low_coeffs*)

satpy.enhancements.**_cira_stretch**(*band_data*)

satpy.enhancements.**_compute_luminance_from_rgb**(*r*, *g*, *b*)

Compute the luminance of the image.

satpy.enhancements.**_create_colormap_from_dataset**(*img*, *dataset*, *color_scale*)

Create a colormap from an auxiliary variable in a source file.

satpy.enhancements.**_jma_true_color_reproduction**(*img_data*, *platform=None*)

Convert from AHI RGB space to sRGB space.

The conversion matrices for this are supplied per-platform. The matrices are computed using the method described in the paper: 'True Color Imagery Rendering for Himawari-8 with a Color Reproduction Approach Based on the CIE XYZ Color System' (DOI:10.2151/jmsj.2018-049).

satpy.enhancements.**_lookup_table**(*band_data*, *luts=None*, *index=-1*)

satpy.enhancements.**_merge_colormaps**(*kwargs*, *img=None*)

Merge colormaps listed in kwargs.

satpy.enhancements.**_piecewise_linear**(*band_data*, *xp*, *fp*)

satpy.enhancements.**_srgb_gamma**(*arr*)

Apply the srgb gamma.

satpy.enhancements.**_three_d_effect**(*band_data*, *kernel=None*, *mode=None*, *index=None*)

satpy.enhancements.**_three_d_effect_delayed**(*band_data*, *kernel*, *mode*)

Kernel for running delayed 3D effect creation.

satpy.enhancements.**btemp_threshold**(*img*, *min_in*, *max_in*, *threshold*, *threshold_out=None*, ***kwargs*)

Scale data linearly in two separate regions.

This enhancement scales the input data linearly by splitting the data into two regions; min_in to threshold and threshold to max_in. These regions are mapped to 1 to threshold_out and threshold_out to 0 respectively, resulting in the data being "flipped" around the threshold. A default threshold_out is set to *176.0 / 255.0* to match the behavior of the US National Weather Service's forecasting tool called AWIPS.

Parameters

- **img** (*XRImage*) – Image object to be scaled
- **min_in** (*float*) – Minimum input value to scale
- **max_in** (*float*) – Maximum input value to scale
- **threshold** (*float*) – Input value where to split data in to two regions
- **threshold_out** (*float*) – Output value to map the input *threshold* to. Optional, defaults to 176.0 / 255.0.

satpy.enhancements.**cira_stretch**(*img*, ***kwargs*)

Logarithmic stretch adapted to human vision.

Applicable only for visible channels.

satpy.enhancements.**colorize**(*img*, ***kwargs*)

Colorize the given image.

Parameters
img – image to be colorized

Kwargs:
palettes: colormap(s) to use

The *palettes* kwarg can be one of the following:

- a trollimage.colormap.Colormap object
- **list of dictionaries with each of one of the following forms:**
  - **{'filename': '/path/to/colors.npy',**
    'min_value': <float, min value to match colors to>, 'max_value': <float, min value to match colors to>, 'reverse': <bool, reverse the colormap if True (default: False)}
  - **{'colors': <trollimage.colormap.Colormap instance>,**
    'min_value': <float, min value to match colors to>, 'max_value': <float, min value to match colors to>, 'reverse': <bool, reverse the colormap if True (default: False)}
  - **{'colors': <tuple of RGB(A) tuples>,**
    'min_value': <float, min value to match colors to>, 'max_value': <float, min value to match colors to>, 'reverse': <bool, reverse the colormap if True (default: False)}
  - **{'colors': <tuple of RGB(A) tuples>,**
    'values': <tuple of values to match colors to>, 'min_value': <float, min value to match colors to>, 'max_value': <float, min value to match colors to>, 'reverse': <bool, reverse the colormap if True (default: False)}
  - **{'dataset': <str, referring to dataset containing palette>,**
    'color_scale': <int, value to be interpreted as white>, 'min_value': <float, see above>, 'max_value': <float, see above>}

If multiple palettes are supplied, they are concatenated before applied.

satpy.enhancements.**create_colormap**(*palette*, *img=None*)

Create colormap of the given numpy file, color vector, or colormap.

Parameters
palette (*dict*) – Information describing how to create a colormap object. See below for more details.

From a file

Colormaps can be loaded from `.npy`, `.npz`, or comma-separated text files. Numpy (npy/npz) files should be 2D arrays with rows for each color. Comma-separated files should have a row for each color with each column representing a single value/channel. The filename to load can be provided with the `filename` key in the provided palette information. A filename ending with `.npy` or `.npz` is read as a numpy file with `numpy.load()`. All other extensions are read as a comma-separated file. For `.npz` files the data must be stored as a positional list where the first element represents the colormap to use. See `numpy.savez()` for more information. The path to the colormap can be relative if it is stored in a directory specified by *Component Configuration Path*. Otherwise it should be an absolute path.

The colormap is interpreted as 1 of 4 different "colormap modes": `RGB`, `RGBA`, `VRGB`, or `VRGBA`. The colormap mode can be forced with the `colormap_mode` key in the provided palette information. If it is not provided then a default will be chosen based on the number of columns in the array (3: RGB, 4: VRGB, 5: VRGBA).

The "V" in the possible colormap modes represents the control value of where that color should be applied. If "V" is not provided in the colormap data it defaults to the row index in the colormap array (0, 1, 2, ...) divided by the total number of colors to produce a number between 0 and 1. See the "Set Range" section below for more information. The remaining elements in the colormap array represent the Red (R), Green (G), and Blue (B) color to be mapped to.

See the "Color Scale" section below for more information on the value range of provided numbers.

From a list

Colormaps can be loaded from lists of colors provided by the `colors` key in the provided dictionary. Each element in the list represents a single color to be mapped to and can be 3 (RGB) or 4 (RGBA) elements long. By default the value or control point for a color is determined by the index in the list (0, 1, 2, ...) divided by the total number of colors to produce a number between 0 and 1. This can be overridden by providing a `values` key in the provided dictionary. See the "Set Range" section below for more information.

See the "Color Scale" section below for more information on the value range of provided numbers.

From a builtin colormap

Colormaps can be loaded by name from the builtin colormaps in the `trollimage` package. Specify the name with the `colors` key in the provided dictionary (ex. `{'colors': 'blues'}`). See Colormap for the full list of available colormaps.

From an auxiliary variable

If the colormap is defined in the same dataset as the data to which the colormap shall be applied, this can be indicated with `{'dataset': 'palette_variable'}`, where `'palette_variable'` is the name of the variable containing the palette. This variable must be an auxiliary variable to the dataset to which the colours are applied. When using this, it is important that one should **not** set `min_value` and `max_value` as those will be taken from the `valid_range` attribute on the dataset and if those differ from `min_value` and `max_value`, the resulting colors will not match the ones in the palette.

Color Scale

By default colors are expected to be in a 0-255 range. This can be overridden by specifying `color_scale` in the provided colormap information. A common alternative to 255 is 1 to specify floating point numbers between 0 and 1. The resulting Colormap uses the normalized color values (0-1).

Set Range

By default the control points or values of the Colormap are between 0 and 1. This means that data values being mapped to a color must also be between 0 and 1. When this is not the case, the expected input range of the data can be used to configure the Colormap and change the control point values. To do this specify the input data range with `min_value` and `max_value`. See `trollimage.colormap.Colormap.set_range()` for more information.

`satpy.enhancements.`**`exclude_alpha`**(*func*)

Exclude the alpha channel from the DataArray before further processing.

`satpy.enhancements.`**`gamma`**(*img*, ***kwargs*)

Perform gamma correction.

`satpy.enhancements.`**`invert`**(*img*, **args*)

Perform inversion.

`satpy.enhancements.`**`jma_true_color_reproduction`**(*img*)

Apply CIE XYZ matrix and return True Color Reproduction data.

Himawari-8 True Color Reproduction Approach Based on the CIE XYZ Color System Hidehiko MURATA, Kotaro SAITOH, and Yasuhiko SUMIDA Meteorological Satellite Center, Japan Meteorological Agency NOAA

National Environmental Satellite, Data, and Information Service Colorado State University—CIRA https://www.jma.go.jp/jma/jma-eng/satellite/introduction/TCR.html

satpy.enhancements.**lookup**(*img*, ***kwargs*)

Assign values to channels based on a table.

satpy.enhancements.**on_dask_array**(*func*)

Pass the underlying dask array to *func* instead of the xarray.DataArray.

satpy.enhancements.**on_separate_bands**(*func*)

Apply *func* one band of the DataArray at a time.

If this decorator is to be applied along with *on_dask_array*, this decorator has to be applied first, eg:

```
@on_separate_bands
@on_dask_array
def my_enhancement_function(data):
    ...
```

satpy.enhancements.**palettize**(*img*, ***kwargs*)

Palettize the given image (no color interpolation).

Arguments as for `colorize()`.

NB: to retain the palette when saving the resulting image, pass `keep_palette=True` to the save method (either via the Scene class or directly in trollimage).

satpy.enhancements.**piecewise_linear_stretch**(*img: XRImage, xp: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes], fp: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes], reference_scale_factor: Number | None = None, **kwargs*) → DataArray

Apply 1D linear interpolation.

This uses `numpy.interp()` mapped over the provided dask array chunks.

Parameters

- **img** – Image data to be scaled. It is assumed the data is already normalized between 0 and 1.
- **xp** – Input reference values of the image data points used for interpolation. This is passed directly to `numpy.interp()`.
- **fp** – Target reference values of the output image data points used for interpolation. This is passed directly to `numpy.interp()`.
- **reference_scale_factor** – Divide `xp` and `fp` by this value before using them for interpolation. This is a convenience to make matching normalized image data to interp coordinates or to avoid floating point precision errors in YAML configuration files. If not provided, `xp` and `fp` will not be modified.

Examples

This example YAML uses a 'crude' stretch to pre-scale the RGB data and then uses reference points in a 0-255 range.

```yaml
true_color_linear_interpolation:
  sensor: abi
  standard_name: true_color
  operations:
  - name: reflectance_range
    method: !!python/name:satpy.enhancements.stretch
    kwargs: {stretch: 'crude', min_stretch: 0., max_stretch: 100.}
  - name: Linear interpolation
    method: !!python/name:satpy.enhancements.piecewise_linear_stretch
    kwargs:
     xp: [0., 25., 55., 100., 255.]
     fp: [0., 90., 140., 175., 255.]
     reference_scale_factor: 255
```

This example YAML does the same as the above on the C02 channel, but the interpolation reference points are already adjusted for the input reflectance (%) data and the output range (0 to 1).

```yaml
c02_linear_interpolation:
  sensor: abi
  standard_name: C02
  operations:
  - name: Linear interpolation
    method: !!python/name:satpy.enhancements.piecewise_linear_stretch
    kwargs:
     xp: [0., 9.8039, 21.5686, 39.2157, 100.]
     fp: [0., 0.3529, 0.5490, 0.6863, 1.0]
```

`satpy.enhancements.`**`reinhard_to_srgb`**(*img*, *saturation=1.25*, *white=100*, ***kwargs*)

Stretch method based on the Reinhard algorithm, using luminance.

Parameters

- **saturation** – Saturation enhancement factor. Less is grayer. Neutral is 1.
- **white** – the reflectance luminance to set to white (in %).

Reinhard, Erik & Stark, Michael & Shirley, Peter & Ferwerda, James. (2002). Photographic Tone Reproduction For Digital Images. ACM Transactions on Graphics. :doi: *21. 10.1145/566654.566575*

`satpy.enhancements.`**`stretch`**(*img*, ***kwargs*)

Perform stretch.

`satpy.enhancements.`**`three_d_effect`**(*img*, ***kwargs*)

Create 3D effect using convolution.

`satpy.enhancements.`**`using_map_blocks`**(*func*)

Run the provided function using `dask.array.core.map_blocks()`.

This means dask will call the provided function with a single chunk as a numpy array.

satpy.modifiers package

Submodules

satpy.modifiers._crefl module

Classes related to the CREFL (corrected reflectance) modifier.

class satpy.modifiers._crefl.**ReflectanceCorrector**(**args*, *dem_filename=None*, *dem_sds='averaged elevation'*, *url=None*, *known_hash=None*, ***kwargs*)

Bases: *ModifierBase*, *DataDownloadMixin*

Corrected Reflectance (crefl) modifier.

Uses a python rewrite of the C CREFL code written for VIIRS and MODIS.

Initialize the compositor with values from the user or from the configuration file.

If *dem_filename* can't be found or opened then correction is done assuming TOA or sealevel options.

Parameters

- **dem_filename** (*str*) – DEPRECATED
- **url** (*str*) – URL or local path to the Digital Elevation Model (DEM) HDF4 file. If unset (None or empty string), then elevation is assumed to be 0 everywhere.
- **known_hash** (*str*) – Optional SHA256 checksum to verify the download of `url`.
- **dem_sds** (*str*) – Name of the variable in the elevation file to load.

_call_crefl(*refl_data*, *angles*)

_extract_angle_data_arrays(*datasets*, *optional_datasets*)

_get_average_elevation()

_get_registered_dem_cache_key()

static _read_fill_value_from_hdf4(*var*, *dtype*)

static _read_var_from_hdf4_file(*local_filename*, *var_name*)

static _read_var_from_hdf4_file_netcdf4(*local_filename*, *var_name*)

static _read_var_from_hdf4_file_pyhdf(*local_filename*, *var_name*)

satpy.modifiers._crefl_utils module

Shared utilities for correcting reflectance data using the 'crefl' algorithm.

The CREFL algorithm in this module is based on the NASA CREFL SPA software, the NASA CVIIRS SPA, and customizations of these algorithms for ABI/AHI by Ralph Kuehn and Min Oo at the Space Science and Engineering Center (SSEC).

The CREFL SPA documentation page describes the algorithm by saying:

The CREFL_SPA processes MODIS Aqua and Terra Level 1B DB data to create the MODIS Level 2 Corrected Reflectance product. The algorithm performs a simple atmospheric correction with MODIS visible, near-infrared, and short-wave infrared bands (bands 1 through 16).

It corrects for molecular (Rayleigh) scattering and gaseous absorption (water vapor and ozone) using climatological values for gas contents. It requires no real-time input of ancillary data. The algorithm performs no aerosol correction. The Corrected Reflectance products created by CREFL_SPA are very similar to the MODIS Land Surface Reflectance product (MOD09) in clear atmospheric conditions, since the algorithms used to derive both are based on the 6S Radiative Transfer Model. The products show differences in the presence of aerosols, however, because the MODIS Land Surface Reflectance product uses a more complex atmospheric correction algorithm that includes a correction for aerosols.

The additional logic to support ABI (AHI support not included) was originally written by Ralph Kuehn and Min Oo at SSEC. Additional modifications were performed by Martin Raspaud, David Hoese, and Will Roberts to make the code work together and be more dask compatible.

The AHI/ABI implementation is based on the MODIS collection 6 algorithm, where a spherical-shell atmosphere was assumed rather than a plane-parallel. See Appendix A in: "The Collection 6 MODIS aerosol products over land and ocean" Atmos. Meas. Tech., 6, 2989–3034, 2013 www.atmos-meas-tech.net/6/2989/2013/ DOI:10.5194/amt-6-2989-2013.

The original CREFL code is similar to what is described in appendix A1 (page 74) of the ATBD for the MODIS MOD04/MYD04 data product.

class satpy.modifiers._crefl_utils.**_ABIAtmosphereVariables**(*G_O3*, *G_H2O*, *G_O2*, **args*)

Bases: *_AtmosphereVariables*

_get_th2o()

_get_to2()

_get_to3()

class satpy.modifiers._crefl_utils.**_ABICREFLRunner**(*refl_data_arr*)

Bases: *_CREFLRunner*

_run_crefl(*mus*, *muv*, *phi*, *solar_zenith*, *sensor_zenith*, *height*, *coeffs*)

property coeffs_cls: Type[*_Coefficients*]

class satpy.modifiers._crefl_utils.**_ABICoefficients**(*wavelength_range*, *resolution=0*)

Bases: *_Coefficients*

COEFF_INDEX_MAP: dict[int, dict[tuple | str, int]] = {2000: {'C01': 0, 'C02': 1, 'C03': 2, 'C05': 3, 'C06': 4, (0.45, 0.47, 0.49, 'µm'): 0, (0.59, 0.64, 0.69, 'µm'): 1, (0.8455, 0.865, 0.8845, 'µm'): 2, (1.58, 1.61, 1.64, 'µm'): 3, (2.225, 2.25, 2.275, 'µm'): 4}}

LUTS: list[ndarray] = [array([0.0024111 , 0.00431497, 0.0079258 , 0.0093392 , 0.0253]), array([0.001236 , 0.0037296 , 0.00017772, 0.0104899 , 0.0163]), array([4.2869000e-03, 1.4107995e-02, 8.0243190e-04, 0.0000000e+00, 2.0000000e-05]), array([0.18472 , 0.052349 , 0.015845 , 0.0013074 , 0.00031129])]

RG_FUDGE = 0.55

class satpy.modifiers._crefl_utils.**_AtmosphereVariables**(*mus*, *muv*, *phi*, *height*, *ah2o*, *bh2o*, *ao3*, *tau*)

Bases: object

_get_th2o()

_get_to2()

_get_to3()

class satpy.modifiers._crefl_utils.**_CREFLRunner**(*refl_data_arr*)

Bases: object

_height_from_avg_elevation(*avg_elevation: ndarray | None*) → Array | float

Get digital elevation map data for our granule with ocean fill value set to 0.

_run_crefl(*mus*, *muv*, *phi*, *solar_zenith*, *sensor_zenith*, *height*, *coeffs*)

property coeffs_cls: Type[*_Coefficients*]

class satpy.modifiers._crefl_utils.**_Coefficients**(*wavelength_range*, *resolution=0*)

Bases: object

COEFF_INDEX_MAP: dict[int, dict[tuple | str, int]] = {}

LUTS: list[ndarray] = []

_find_coefficient_index(*wavelength_range*, *resolution=0*)

Return index in to coefficient arrays for this band's wavelength.

This function search through the *COEFF_INDEX_MAP* dictionary and finds the first key where the nominal wavelength of *wavelength_range* falls between the minimum wavelength and maximum wavelength of the key. *wavelength_range* can also be the standard name of the band. For example, "M05" for VIIRS or "1" for MODIS.

Parameters

- **wavelength_range** – 3-element tuple of (min wavelength, nominal wavelength, max wavelength) or the string name of the band.
- **resolution** – resolution of the band to be corrected

Returns

index in to coefficient arrays like *aH2O*, *aO3*, etc. None is returned if no matching wavelength is found

satpy.modifiers._crefl_utils.**_G_calc**(*zenith*, *a_coeff*)

class satpy.modifiers._crefl_utils.**_MODISAtmosphereVariables**(**args*)

Bases: *_VIIRSAtmosphereVariables*

_get_th2o()

_get_to3()

class satpy.modifiers._crefl_utils.**_MODISCREFLRunner**(*refl_data_arr*)

Bases: *_VIIRSMODISCREFLRunner*

_run_crefl(*mus*, *muv*, *phi*, *solar_zenith*, *sensor_zenith*, *height*, *coeffs*)

property coeffs_cls: Type[*_Coefficients*]

class satpy.modifiers._crefl_utils.**_MODISCoefficients**(*wavelength_range*, *resolution=0*)

Bases: *_Coefficients*

```
COEFF_INDEX_MAP: dict[int, dict[tuple | str, int]] = {250: {'1': 0, '2': 1, '3':
2, '4': 3, '5': 4, '6': 5, '7': 6, (0.459, 0.469, 0.479, 'µm'): 2, (0.545,
0.555, 0.565, 'µm'): 3, (0.62, 0.645, 0.67, 'µm'): 0, (0.841, 0.8585, 0.876,
'µm'): 1, (1.23, 1.24, 1.25, 'µm'): 4, (1.628, 1.64, 1.652, 'µm'): 5, (2.105,
2.13, 2.155, 'µm'): 6}, 500: {'1': 0, '2': 1, '3': 2, '4': 3, '5': 4, '6':
5, '7': 6, (0.459, 0.469, 0.479, 'µm'): 2, (0.545, 0.555, 0.565, 'µm'): 3, (0.62,
0.645, 0.67, 'µm'): 0, (0.841, 0.8585, 0.876, 'µm'): 1, (1.23, 1.24, 1.25, 'µm'):
4, (1.628, 1.64, 1.652, 'µm'): 5, (2.105, 2.13, 2.155, 'µm'): 6}, 1000: {'1': 0,
'2': 1, '3': 2, '4': 3, '5': 4, '6': 5, '7': 6, (0.459, 0.469, 0.479, 'µm'):
2, (0.545, 0.555, 0.565, 'µm'): 3, (0.62, 0.645, 0.67, 'µm'): 0, (0.841, 0.8585,
0.876, 'µm'): 1, (1.23, 1.24, 1.25, 'µm'): 4, (1.628, 1.64, 1.652, 'µm'): 5,
(2.105, 2.13, 2.155, 'µm'): 6}}
```

```
LUTS: list[ndarray] = [array([-5.60723, -5.25251, 0. , 0. , -6.29824, -7.70944,
-3.91877, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]), array([0.820175, 0.725159,
0. , 0. , 0.865732, 0.966947, 0.745342, 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.
]), array([0.0715289 , 0. , 0.00743232, 0.089691 , 0. , 0. , 0. , 0.001 , 0.00383 ,
0.0225 , 0.0663 , 0.0836 , 0.0485 , 0.0395 , 0.0119 , 0.00263 ]), array([0.051 ,
0.01631, 0.19325, 0.09536, 0.00366, 0.00123, 0.00043, 0.3139 , 0.2375 , 0.1596 ,
0.1131 , 0.0994 , 0.0446 , 0.0416 , 0.0286 , 0.0155 ])]
```

class satpy.modifiers._crefl_utils.**_VIIRSAtmosphereVariables**(**args*)

Bases: *_AtmosphereVariables*

_compute_airmass()

_get_th2o()

_get_to3()

class satpy.modifiers._crefl_utils.**_VIIRSCREFLRunner**(*refl_data_arr*)

Bases: *_VIIRSMODISCREFLRunner*

_run_crefl(*mus*, *muv*, *phi*, *solar_zenith*, *sensor_zenith*, *height*, *coeffs*)

property coeffs_cls: Type[*_Coefficients*]

class satpy.modifiers._crefl_utils.**_VIIRSCoefficients**(*wavelength_range*, *resolution=0*)

Bases: *_Coefficients*

```
COEFF_INDEX_MAP: dict[int, dict[tuple | str, int]] = {500: {'I01': 7, 'I02': 8,
'I03': 9, (0.6, 0.64, 0.68, 'µm'): 7, (0.845, 0.865, 0.884, 'µm'): 8, (1.58,
1.61, 1.64, 'µm'): 9}, 1000: {'M03': 2, 'M04': 3, 'M05': 0, 'M07': 1, 'M08':
4, 'M10': 5, 'M11': 6, (0.478, 0.488, 0.498, 'µm'): 2, (0.545, 0.555, 0.565,
'µm'): 3, (0.662, 0.672, 0.682, 'µm'): 0, (0.846, 0.865, 0.885, 'µm'): 1, (1.23,
1.24, 1.25, 'µm'): 4, (1.58, 1.61, 1.64, 'µm'): 5, (2.225, 2.25, 2.275, 'µm'):
6}}
```

```
LUTS: list[ndarray] = [array([4.06601e-04, 1.59330e-03, 0.00000e+00, 1.78644e-05,
2.96457e-03, 6.17252e-04, 9.96563e-04, 2.22253e-03, 9.40050e-04, 5.63288e-04,
0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00, 0.00000e+00]),
array([0.812659, 0.832931, 1. , 0.867785, 0.806816, 0.944958, 0.78812 , 0.791204,
0.900564, 0.942907, 0. , 0. , 0. , 0. , 0. , 0. ]), array([0.0433461, 0. ,
0.0178299, 0.0853012, 0. , 0. , 0. , 0.0813531, 0. , 0. , 0.0663 , 0.0836 , 0.0485 ,
0.0395 , 0.0119 , 0.00263 ]), array([0.0435 , 0.01582, 0.16176, 0.0974 , 0.00369,
0.00132, 0.00033, 0.05373, 0.01561, 0.00129, 0.1131 , 0.0994 , 0.0446 , 0.0416 ,
0.0286 , 0.0155 ])]
```

class satpy.modifiers._crefl_utils.**_VIIRSMODISCREFLRunner**(*refl_data_arr*)

Bases: *_CREFLRunner*

_run_crefl(*mus*, *muv*, *phi*, *solar_zenith*, *sensor_zenith*, *height*, *coeffs*)

satpy.modifiers._crefl_utils.**_chand**(*phi*, *muv*, *mus*, *taur*)

satpy.modifiers._crefl_utils.**_correct_refl**(*refl*, *tOG*, *rhoray*, *TtotraytH2O*, *sphalb*)

satpy.modifiers._crefl_utils.**_csalbr**(*tau*)

satpy.modifiers._crefl_utils.**_run_crefl**(*refl*, *mus*, *muv*, *phi*, *height*, *sensor_name*, **coeffs*)

satpy.modifiers._crefl_utils.**_run_crefl_abi**(*refl*, *mus*, *muv*, *phi*, *solar_zenith*, *sensor_zenith*, *height*, **coeffs*)

satpy.modifiers._crefl_utils.**_runner_class_for_sensor**(*sensor_name: str*) → Type[*_CREFLRunner*]

satpy.modifiers._crefl_utils.**_space_mask_height**(*lon*, *lat*, *avg_elevation*)

satpy.modifiers._crefl_utils.**run_crefl**(*refl*, *sensor_azimuth*, *sensor_zenith*, *solar_azimuth*, *solar_zenith*, *avg_elevation=None*)

Run main crefl algorithm.

All input parameters are per-pixel values meaning they are the same size and shape as the input reflectance data, unless otherwise stated.

Parameters

- **refl** – tuple of reflectance band arrays
- **sensor_azimuth** – input swath sensor azimuth angle array
- **sensor_zenith** – input swath sensor zenith angle array
- **solar_azimuth** – input swath solar azimuth angle array
- **solar_zenith** – input swath solar zenith angle array
- **avg_elevation** – average elevation (usually pre-calculated and stored in CMGDEM.hdf)

satpy.modifiers.angles module

Utilties for getting various angles for a dataset..

class satpy.modifiers.angles.**ZarrCacheHelper**(*func: ~typing.Callable*, *cache_config_key: str*, *uncacheable_arg_types=(<class 'pyresample.geometry.SwathDefinition'>, <class 'xarray.core.dataarray.DataArray'>, <class 'dask.array.core.Array'>)*, *sanitize_args_func: ~typing.Callable | None = None*, *cache_version: int = 1*)

Bases: object

Helper for caching function results to on-disk zarr arrays.

It is recommended to use this class through the *cache_to_zarr_if()* decorator rather than using it directly.

Currently the cache does not perform any limiting or removal of cache content. That is left up to the user to manage. Caching is based on arguments passed to the decorated function but will only be performed if the

arguments are of a certain type (see `uncacheable_arg_types`). The cache value to use is purely based on the hash value of all of the provided arguments along with the "cache version" (see below).

Note that the zarr format requires regular chunking of data. That is, chunks must be all the same size per dimension except for the last chunk. To work around this limitation, this class will determine a good regular chunking based on the existing chunking scheme, rechunk the input arguments, and then rechunk the results before returning them to the user. This rechunking is only done if caching is enabled.

Parameters

- **func** – Function that will be called to generate the value to cache.
- **cache_config_key** – Name of the boolean `satpy.config` parameter to use to determine if caching should be done.
- **uncacheable_arg_types** – Types that if present in the passed arguments should trigger caching to *not* happen. By default this includes `SwathDefinition`, `xr.DataArray`, and `da.Array` objects.
- **sanitize_args_func** – Optional function to call to sanitize provided arguments before they are considered for caching. This can be used to make arguments more "cacheable" by replacing them with similar values that will result in more cache hits. Note that the sanitized arguments are only passed to the underlying function if caching will be performed, otherwise the original arguments are passed.
- **cache_version** – Version number used to distinguish one version of a decorated function from future versions.

Notes

- Caching only supports dask array values.
- This helper allows for an additional `cache_dir` parameter to override the use of the `satpy.config` `cache_dir` parameter.

Examples

To use through the *cache_to_zarr_if()* decorator:

```
@cache_to_zarr_if("cache_my_stuff")
def generate_my_stuff(area_def: AreaDefinition, some_factor: int) -> da.Array:
    # Generate
    return my_dask_arr
```

To use the decorated function:

```
with satpy.config.set(cache_my_stuff=True):
    my_stuff = generate_my_stuff(area_def, 5)
```

Hold on to provided arguments for future use.

_cache_and_read(*args*, *cache_dir*)

_cache_results(*res*, *zarr_file_pattern*)

static _get_cache_dir_from_config(*cache_dir: str | None*) → str

_get_zarr_file_pattern(*sanitized_args*, *cache_dir*)

static _warn_if_irregular_input_chunks(*args*, *modified_args*)

_zarr_pattern(*arg_hash*, *cache_version: None | int | str = None*) → str

cache_clear(*cache_dir: str | None = None*)

Remove all on-disk files associated with this function.

Intended to mimic the `functools.cache()` behavior.

satpy.modifiers.angles.**_chunks_are_irregular**(*chunks_tuple: tuple*) → bool

Determine if an array is irregularly chunked.

Zarr does not support saving data in irregular chunks. Regular chunking is when all chunks are the same size (except for the last one).

satpy.modifiers.angles.**_cos_zen_ndarray**(*lons*, *lats*, *utc_time*)

satpy.modifiers.angles.**_dim_index_with_default**(*dims: tuple*, *dim_name: str*, *default: int*) → int

satpy.modifiers.angles.**_geo_chunks_from_data_arr**(*data_arr: DataArray*) → tuple

satpy.modifiers.angles.**_geo_dask_to_data_array**(*arr: Array*) → DataArray

satpy.modifiers.angles.**_get_cos_sza**(*utc_time*, *lons*, *lats*)

satpy.modifiers.angles.**_get_output_chunks_from_func_arguments**(*args*)

Determine what the desired output chunks are.

It is assumed a tuple of tuples of integers is defining chunk sizes. If a tuple like this is not found then arguments are checked for array-like objects with a `.chunks` attribute.

satpy.modifiers.angles.**_get_sensor_angles**(*data_arr: DataArray*) → tuple[DataArray, DataArray]

satpy.modifiers.angles.**_get_sensor_angles_ndarray**(*lons*, *lats*, *start_time*, *sat_lon*, *sat_lat*, *sat_alt*) → ndarray

satpy.modifiers.angles.**_get_sun_angles**(*data_arr: DataArray*) → tuple[DataArray, DataArray]

satpy.modifiers.angles.**_get_sun_azimuth_ndarray**(*lons: ndarray*, *lats: ndarray*, *start_time: datetime*) → ndarray

satpy.modifiers.angles.**_hash_args**(**args*, *unhashable_types=(<class 'pyresample.geometry.SwathDefinition'>, <class 'xarray.core.dataarray.DataArray'>, <class 'dask.array.core.Array'>)*)

satpy.modifiers.angles.**_is_chunk_tuple**(*some_obj: Any*) → bool

satpy.modifiers.angles.**_regular_chunks_from_irregular_chunks**(*old_chunks: tuple[tuple[int, ...], ...]*) → tuple[tuple[int, ...], ...]

satpy.modifiers.angles.**_sanitize_args_with_chunks**(**args*)

satpy.modifiers.angles.**_sanitize_observer_look_args**(**args*)

satpy.modifiers.angles.**_sunzen_corr_cos_ndarray**(*data: ndarray*, *cos_zen: ndarray*, *limit: float*, *max_sza: float | None*) → ndarray

satpy.modifiers.angles.**_sunzen_reduction_ndarray**(*data: ndarray, sunz: ndarray, limit: float, max_sza: float, strength: float*) → ndarray

satpy.modifiers.angles.**cache_to_zarr_if**(*cache_config_key: str, uncacheable_arg_types=(<class 'pyresample.geometry.SwathDefinition'>, <class 'xarray.core.dataarray.DataArray'>, <class 'dask.array.core.Array'>), sanitize_args_func: ~typing.Callable | None = None*) → Callable

Decorate a function and cache the results as a zarr array on disk.

This only happens if the `satpy.config` boolean value for the provided key is `True` as well as some other conditions. See *ZarrCacheHelper* for more information. Most importantly, this decorator does not limit how many items can be cached and does not clear out old entries. It is up to the user to manage the size of the cache.

satpy.modifiers.angles.**compute_relative_azimuth**(*sat_azi: DataArray, sun_azi: DataArray*) → DataArray

Compute the relative azimuth angle.

Parameters

- **sat_azi** – DataArray for the satellite azimuth angles, typically in 0-360 degree range.
- **sun_azi** – DataArray for the solar azimuth angles, should be in same range as sat_azi.

Returns

A DataArray containing the relative azimuth angle in the 0-180 degree range.

NOTE: Relative azimuth is defined such that: Relative azimuth is 0 when sun and satellite are aligned on one side of a pixel (back scatter). Relative azimuth is 180 when sun and satellite are directly opposite each other (forward scatter).

satpy.modifiers.angles.**get_angles**(*data_arr: DataArray*) → tuple[DataArray, DataArray, DataArray, DataArray]

Get sun and satellite azimuth and zenith angles.

Note that this function can benefit from the `satpy.config` parameters *cache_lonlats* and *cache_sensor_angles* being set to `True`.

Parameters

data_arr – DataArray to get angles for. Information extracted from this object are `.attrs["area"]`,``.attrs["start_time"]``, and `.attrs["orbital_parameters"]`. See *satpy.utils.get_satpos()* and *Metadata* for more information. Additionally, the dask array chunk size is used when generating new arrays. The actual data of the object is not used.

Returns

Four DataArrays representing sensor azimuth angle, sensor zenith angle, solar azimuth angle, and solar zenith angle. All values are in degrees. Sensor angles are provided in the [0, 360] degree range. Solar angles are provided in the [-180, 180] degree range.

satpy.modifiers.angles.**get_cos_sza**(*data_arr: DataArray*) → DataArray

Generate the cosine of the solar zenith angle for the provided data.

Returns

DataArray with the same shape as `data_arr`.

satpy.modifiers.angles.**get_satellite_zenith_angle**(*data_arr: DataArray*) → DataArray

Generate satellite zenith angle for the provided data.

Note that this function can benefit from the `satpy.config` parameters *cache_lonlats* and *cache_sensor_angles* being set to `True`. Values are in degrees.

satpy.modifiers.angles.**sunzen_corr_cos**(*data: Array, cos_zen: Array, limit: float = 88.0, max_sza: float | None = 95.0*) → Array

Perform Sun zenith angle correction.

The correction is based on the provided cosine of the zenith angle (`cos_zen`). The correction is limited to `limit` degrees (default: 88.0 degrees). For larger zenith angles, the correction is the same as at the `limit` if `max_sza` is *None*. The default behavior is to gradually reduce the correction past `limit` degrees up to `max_sza` where the correction becomes 0. Both `data` and `cos_zen` should be 2D arrays of the same shape.

satpy.modifiers.angles.**sunzen_reduction**(*data: Array, sunz: Array, limit: float = 55.0, max_sza: float = 90.0, strength: float = 1.5*) → Array

Reduced strength of signal at high sun zenith angles.

satpy.modifiers.atmosphere module

Modifiers related to atmospheric corrections or adjustments.

class satpy.modifiers.atmosphere.**CO2Corrector**(*name, prerequisites=None, optional_prerequisites=None, **kwargs*)

Bases: `ModifierBase`

CO2 correction of the brightness temperature of the MSG 3.9um channel.

$$T4_CO2corr = (BT(IR3.9)^4 + Rcorr)^0.25Rcorr = BT(IR10.8)^4 - (BT(IR10.8) - dt_CO2)^4dt_CO2 = (BT(IR10.8) - B$$

Derived from D. Rosenfeld, "CO2 Correction of Brightness Temperature of Channel IR3.9" .. rubric:: References

- https://resources.eumetrain.org/IntGuide/PowerPoints/Channels/conversion.ppt

Initialise the compositor.

class satpy.modifiers.atmosphere.**PSPAtmosphericalCorrection**(*name, prerequisites=None, optional_prerequisites=None, **kwargs*)

Bases: `ModifierBase`

Correct for atmospheric effects.

Initialise the compositor.

class satpy.modifiers.atmosphere.**PSPRayleighReflectance**(*name, prerequisites=None, optional_prerequisites=None, **kwargs*)

Bases: `ModifierBase`

Pyspectral-based rayleigh corrector for visible channels.

It is possible to use `reduce_lim_low`, `reduce_lim_high` and `reduce_strength` together to reduce rayleigh correction at high solar zenith angle and make the image transition from rayleigh-corrected to partially/none rayleigh-corrected at day/night edge, therefore producing a more natural look, which could be especially helpful for geostationary satellites. This reduction starts at solar zenith angle of `reduce_lim_low`, and ends in `reduce_lim_high`. It's linearly scaled between these two angles. The `reduce_strength` controls the amount of the reduction. When the solar zenith angle reaches `reduce_lim_high`, the rayleigh correction will remain `(1 - reduce_strength)` of its initial reduce_strength at `reduce_lim_high`.

To use this function in a YAML configuration file:

```yaml
rayleigh_corrected_reduced:
  modifier: !!python/name:satpy.modifiers.PSPRayleighReflectance
  atmosphere: us-standard
  aerosol_type: rayleigh_only
  reduce_lim_low: 70
  reduce_lim_high: 95
  reduce_strength: 0.6
  prerequisites:
    - name: B03
      modifiers: [sunz_corrected]
  optional_prerequisites:
    - satellite_azimuth_angle
    - satellite_zenith_angle
    - solar_azimuth_angle
    - solar_zenith_angle
```

In the case above, rayleigh correction is reduced gradually starting at solar zenith angle 70°. When reaching 95°, the correction will only remain 40% its initial strength at 95°.

Initialise the compositor.

satpy.modifiers.atmosphere.**_call_mapped_correction**(*satz*, *band_data*, *corrector*, *band_name*)

satpy.modifiers.base module

Base modifier classes and utilities.

class satpy.modifiers.base.**ModifierBase**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *CompositeBase*

Base class for all modifiers.

A modifier in Satpy is a class that takes one input DataArray to be changed along with zero or more other input DataArrays used to perform these changes. The result of a modifier typically has a lot of the same metadata (name, units, etc) as the original DataArray, but the data is different. A modified DataArray can be differentiated from the original DataArray by the *modifiers* property of its *DataID*.

See the *CompositeBase* class for information on the similar concept of "compositors".

Initialise the compositor.

satpy.modifiers.filters module

Tests for image filters.

class satpy.modifiers.filters.**Median**(*median_filter_params*, ***kwargs*)

Bases: *ModifierBase*

Apply a median filter to the band.

Create the instance.

Parameters

median_filter_params – The arguments to pass to dask-image's median_filter function. For example, {size: 3} makes give the median filter a kernel of size 3.

satpy.modifiers.geometry module

Modifier classes for corrections based on sun and other angles.

class satpy.modifiers.geometry.**EffectiveSolarPathLengthCorrector**(*correction_limit=88.0*, ***kwargs*)

Bases: *SunZenithCorrectorBase*

Special sun zenith correction with the method proposed by Li and Shibata.

(2006): https://doi.org/10.1175/JAS3682.1

In addition to adjusting the provided reflectances by the cosine of the solar zenith angle, this modifier forces all reflectances beyond a solar zenith angle of *max_sza* to 0 to reduce noise in the final data. It also gradually reduces the amount of correction done between `correction_limit` and `max_sza`. If `max_sza` is `None` then a constant correction is applied to zenith angles beyond `correction_limit`.

To set `max_sza` to `None` in a YAML configuration file use:

```
effective_solar_pathlength_corrected:
  modifier: !!python/name:satpy.modifiers.EffectiveSolarPathLengthCorrector
  max_sza: !!null
  optional_prerequisites:
  - solar_zenith_angle
```

Collect custom configuration values.

Parameters

- **correction_limit** (*float*) – Maximum solar zenith angle to apply the correction in degrees. Pixels beyond this limit have a constant correction applied. Default 88.
- **max_sza** (*float*) – Maximum solar zenith angle in degrees that is considered valid and correctable. Default 95.0.

_apply_correction(*proj*, *coszen*)

class satpy.modifiers.geometry.**SunZenithCorrector**(*correction_limit=88.0*, ***kwargs*)

Bases: *SunZenithCorrectorBase*

Standard sun zenith correction using `1 / cos(sunz)`.

In addition to adjusting the provided reflectances by the cosine of the solar zenith angle, this modifier forces all reflectances beyond a solar zenith angle of `max_sza` to 0. It also gradually reduces the amount of correction done between `correction_limit` and `max_sza`. If `max_sza` is `None` then a constant correction is applied to zenith angles beyond `correction_limit`.

To set `max_sza` to `None` in a YAML configuration file use:

```
sunz_corrected:
  modifier: !!python/name:satpy.modifiers.SunZenithCorrector
  max_sza: !!null
  optional_prerequisites:
  - solar_zenith_angle
```

Collect custom configuration values.

Parameters

- **correction_limit** (*float*) – Maximum solar zenith angle to apply the correction in degrees. Pixels beyond this limit have a constant correction applied. Default 88.

- **max_sza** (*float*) – Maximum solar zenith angle in degrees that is considered valid and correctable. Default 95.0.

_apply_correction(*proj*, *coszen*)

class satpy.modifiers.geometry.**SunZenithCorrectorBase**(*max_sza=95.0*, ***kwargs*)

Bases: *ModifierBase*

Base class for sun zenith correction modifiers.

Collect custom configuration values.

Parameters

max_sza (*float*) – Maximum solar zenith angle in degrees that is considered valid and correctable. Default 95.0.

_apply_correction(*proj*, *coszen*)

class satpy.modifiers.geometry.**SunZenithReducer**(*correction_limit=80.0*, *max_sza=90*, *strength=1.3*, ***kwargs*)

Bases: *SunZenithCorrectorBase*

Reduce signal strength at large sun zenith angles.

Within a given sunz interval [correction_limit, max_sza] the strength of the signal is reduced following the formula:

res = signal * reduction_factor

where reduction_factor is a pixel-level value ranging from 0 to 1 within the sunz interval.

The *strength* parameter can be used for a non-linear reduction within the sunz interval. A strength larger than 1.0 will decelerate the signal reduction towards the sunz interval extremes, whereas a strength smaller than 1.0 will accelerate the signal reduction towards the sunz interval extremes.

Collect custom configuration values.

Parameters

- **correction_limit** (*float*) – Solar zenith angle in degrees where to start the signal reduction.
- **max_sza** (*float*) – Maximum solar zenith angle in degrees where to apply the signal reduction. Beyond this solar zenith angle the signal will become zero.
- **strength** (*float*) – The strength of the non-linear signal reduction.

_apply_correction(*proj*, *coszen*)

satpy.modifiers.parallax module

Parallax correction.

Routines related to parallax correction using datasets involving height, such as cloud top height.

The geolocation of (geostationary) satellite imagery is calculated by agencies or in satpy readers with the assumption of a clear view from the satellite to the geoid. When a cloud blocks the view of the Earth surface or the surface is above sea level, the geolocation is not accurate for the cloud or mountain top. This module contains routines to correct imagery such that pixels are shifted or interpolated to correct for this parallax effect.

Parallax correction is currently only supported for (cloud top) height that arrives on an AreaDefinition, such as is standard for geostationary satellites. Parallax correction with data described by a SwathDefinition, such as is common for polar satellites, is not (yet) supported.

See also the ../modifiers page in the documentation for an introduction to parallax correction as a modifier in Satpy.

exception satpy.modifiers.parallax.**IncompleteHeightWarning**

Bases: UserWarning

Raised when heights only partially overlap with area to be corrected.

exception satpy.modifiers.parallax.**MissingHeightError**

Bases: ValueError

Raised when heights do not overlap with area to be corrected.

class satpy.modifiers.parallax.**ParallaxCorrection**(*base_area*, *debug_mode=False*)

Bases: object

Parallax correction calculations.

This class contains higher-level functionality to wrap the parallax correction calculations in *get_parallax_corrected_lonlats()*. The class is initialised using a base area, which is the area for which a corrected geolocation will be calculated. The resulting object is a callable. Calling the object with an array of (cloud top) heights returns a SwathDefinition describing the new , corrected geolocation. The cloud top height should cover at least the area for which the corrected geolocation will be calculated.

Note that the ctth dataset must contain satellite location metadata, such as set in the orbital_parameters dataset attribute that is set by many Satpy readers. It is essential that the datasets to be corrected are coming from the same platform as the provided cloud top height.

A note on the algorithm and the implementation. Parallax correction is inherently an inverse problem. The reported geolocation in satellite data files is the true location plus the parallax error. Therefore, this class first calculates the true geolocation (using *get_parallax_corrected_lonlats()*), which gives a shifted longitude and shifted latitude on an irregular grid. The difference between the original and the shifted grid is the parallax error or shift. The magnitude of this error can be estimated with *get_surface_parallax_displacement()*. With this difference, we need to invert the parallax correction to calculate the corrected geolocation. Due to parallax correction, high clouds shift a lot, low clouds shift a little, and cloud-free pixels shift not at all. The shift may result in zero, one, two, or more source pixel onto a destination pixel. Physically, this corresponds to the situation where a narrow but high cloud is viewed at a large angle. The cloud may occupy two or more pixels when viewed at a large angle, but only one when viewed straight from above. To accurately reproduce this perspective, the parallax correction uses the BucketResampler class, specifically the get_abs_max() method, to retain only the largest absolute shift (corresponding to the highest cloud) within each pixel. Any other resampling method at this step would yield incorrect results. When cloud moves over clear-sky, the clear-sky pixel is unshifted and the shift is located exactly in the centre of the grid box, so nearest-neighbour resampling would lead to such shifts being deselected. Other resampling methods would average large shifts with small shifts, leading to unpredictable results. Now the reprojected shifts can be applied to the original lat/lon, returning a new SwathDefinition. This is is the object returned by *corrected_area()*.

This procedure can be configured as a modifier using the *ParallaxCorrectionModifier* class. However, the modifier can only be applied to one dataset at the time, which may not provide optimal performance, although dask should reuse identical calculations between multiple channels.

Initialise parallax correction class.

Parameters

- **base_area** (AreaDefinition) – Area for which calculated geolocation will be calculated.
- **debug_mode** (*bool*) – Store diagnostic information in self.diagnostics. This attribute always apply to the most recently applied operation only.

_check_overlap(*cth_dataset*)

Ensure cth_dataset is usable for parallax correction.

Checks the coverage of `cth_dataset` compared to the `base_area`. If the entirety of `base_area` is covered by `cth_dataset`, do nothing. If only part of `base_area` is covered by `cth_dataset`, raise a *IncompleteHeightWarning*. If none of `base_area` is covered by `cth_dataset`, raise a *MissingHeightError*.

_get_corrected_lon_lat(*base_lon*, *base_lat*, *shifted_area*)

Calculate the corrected lon/lat based from the shifted area.

After calculating the shifted area based on `get_parallax_corrected_lonlats()`, we invert the parallax error and estimate where those pixels came from. For details on the algorithm, see the class docstring.

static _get_swathdef_from_lon_lat(*lon*, *lat*)

Return a SwathDefinition from lon/lat.

Turn ndarrays describing lon/lat into xarray with dimensions y, x, then use these to create a `SwathDefinition`.

_prepare_cth_dataset(*cth_dataset*, *resampler='nearest'*, *radius_of_influence=50000*, *lonlat_chunks=1024*)

Prepare CTH dataset.

Set cloud top height to zero wherever lat/lon are valid but CTH is undefined. Then resample onto the base area.

corrected_area(*cth_dataset*, *cth_resampler='nearest'*, *cth_radius_of_influence=50000*, *lonlat_chunks=1024*)

Return the parallax corrected SwathDefinition.

Using the cloud top heights provided in `cth_dataset`, calculate the `pyresample.geometry.SwathDefinition` that estimates the geolocation for each pixel if it had been viewed from straight above (without parallax error). The cloud top height will first be resampled onto the area passed upon class initialisation in `__init__()`. Pixels that are invisible after parallax correction are not retained but get geolocation NaN.

Parameters

- **cth_dataset** (`DataArray`) – Cloud top height in meters. The variable attributes must contain an `area` attribute describing the geolocation in a pyresample-aware way, and they must contain satellite orbital parameters. The dimensions must be `(y, x)`. For best performance, this should be a dask-based `DataArray`.
- **cth_resampler** (*`string, optional`*) – Resampler to use when resampling the (cloud top) height to the base area. Defaults to "nearest".
- **cth_radius_of_influence** (*`number, optional`*) – Radius of influence to use when resampling the (cloud top) height to the base area. Defaults to 50000.
- **lonlat_chunks** (*`int, optional`*) – Chunking to use when calculating lon/lats. Probably the default (1024) should be fine.

Returns

`SwathDefinition` describing parallax corrected geolocation.

class satpy.modifiers.parallax.**ParallaxCorrectionModifier**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *`ModifierBase`*

Modifier for parallax correction.

Apply parallax correction as a modifier. Uses the *ParallaxCorrection* class, which in turn uses the *get_parallax_corrected_lonlats()* function. See the documentation there for details on the behaviour.

To use this, add to `composites/visir.yaml` within `SATPY_CONFIG_PATH` something like:

```
sensor_name: visir

modifiers:
  parallax_corrected:
    modifier: !!python/name:satpy.modifiers.parallax.ParallaxCorrectionModifier
    prerequisites:
      - "ctth_alti"
    dataset_radius_of_influence: 50000

composites:

  parallax_corrected_VIS006:
    compositor: !!python/name:satpy.composites.SingleBandCompositor
    prerequisites:
      - name: VIS006
        modifiers: [parallax_corrected]
```

Here, `ctth_alti` is CTH provided by the `nwcsaf-geo` reader, so to use it one would have to pass both on scene creation:

```
sc = Scene({"seviri_l1b_hrit": files_l1b, "nwcsaf-geo": files_l2})
sc.load(["parallax_corrected_VIS006"])
```

The modifier takes optional global parameters, all of which are optional. They affect various steps in the algorithm. Setting them may impact performance:

cth_resampler
Resampler to use when resampling (cloud top) height to the base area. Defaults to "nearest".

cth_radius_of_influence
Radius of influence to use when resampling the (cloud top) height to the base area. Defaults to 50000.

lonlat_chunks
Chunk size to use when obtaining longitudes and latitudes from the area definition. Defaults to 1024. If you set this to None, then parallax correction will involve premature calculation. Changing this may or may not make parallax correction slower or faster.

dataset_radius_of_influence
Radius of influence to use when resampling the dataset onto the swathdefinition describing the parallax-corrected area. Defaults to 50000. This always uses nearest neighbour resampling.

Alternately, you can use the lower-level API directly with the *ParallaxCorrection* class, which may be more efficient if multiple datasets need to be corrected. RGB Composites cannot be modified in this way (i.e. you can't replace "VIS006" by "natural_color"). To get a parallax corrected RGB composite, create a new composite where each input has the modifier applied. The parallax calculation should only occur once, because calculations are happening via dask and dask should reuse the calculation.

Initialise the compositor.

_get_corrector(*base_area*)

satpy.modifiers.parallax.**_calculate_slant_cloud_distance**(*height*, *elevation*)

Calculate slant cloud to ground distance.

From (cloud top) height and satellite elevation, calculate the slant cloud-to-ground distance along the line of sight of the satellite.

satpy.modifiers.parallax.**_get_parallax_shift_xyz**(*sat_lon*, *sat_lat*, *sat_alt*, *lon*, *lat*, *parallax_distance*)

Calculate the parallax shift in cartesian coordinates.

From satellite position and cloud position, get the parallax shift in cartesian coordinates:

Parameters

- **sat_lon** (*number*) – Satellite longitude in geodetic coordinates [degrees]
- **sat_lat** (*number*) – Satellite latitude in geodetic coordinates [degrees]
- **sat_alt** (*number*) – Satellite altitude above the Earth surface [m]
- **lon** (*array or number*) – Longitudes of pixel or pixels to be corrected, in geodetic coordinates [degrees]
- **lat** (*array or number*) – Latitudes of pixel/pixels to be corrected, in geodetic coordinates [degrees]
- **parallax_distance** (*array or number*) – Cloud to ground distance with parallax effect [m].

Returns

Parallax shift in cartesian coordinates in meter.

satpy.modifiers.parallax.**_get_satellite_elevation**(*sat_lon*, *sat_lat*, *sat_alt*, *lon*, *lat*)

Get satellite elevation.

Get the satellite elevation from satellite lon/lat/alt for positions lon/lat.

satpy.modifiers.parallax.**_get_satpos_from_cth**(*cth_dataset*)

Obtain satellite position from CTH dataset, height in meter.

From a CTH dataset, obtain the satellite position lon, lat, altitude/m, either directly from orbital parameters, or, when missing, from the platform name using pyorbital and skyfield.

satpy.modifiers.parallax.**get_parallax_corrected_lonlats**(*sat_lon*, *sat_lat*, *sat_alt*, *lon*, *lat*, *height*)

Calculate parallax corrected lon/lats.

Satellite geolocation generally assumes an unobstructed view of a smooth Earth surface. In reality, this view may be obstructed by clouds or mountains.

If the view of a pixel at location (lat, lon) is blocked by a cloud at height h, this function calculates the (lat, lon) coordinates of the cloud above/in front of the invisible surface.

For scenes that are only partly cloudy, the user might set the cloud top height for clear-sky pixels to NaN. This function will return a corrected lat/lon as NaN as well. The user can use the original lat/lon for those pixels or use the higher level `ParallaxCorrection` class.

This function assumes a spherical Earth.

Note: Be careful with units! This code expects `sat_alt` and `height` to be in meter above the Earth's surface. You may have to convert your input correspondingly. Cloud Top Height is usually reported in meters above the Earth's surface, rarely in km. Satellite altitude may be reported in either m or km, but orbital parameters are usually in relation to the Earth's centre. The Earth radius from pyresample is reported in km.

Parameters

- **sat_lon** (*number*) – Satellite longitude in geodetic coordinates [degrees]
- **sat_lat** (*number*) – Satellite latitude in geodetic coordinates [degrees]
- **sat_alt** (*number*) – Satellite altitude above the Earth surface [m]
- **lon** (*array or number*) – Longitudes of pixel or pixels to be corrected, in geodetic coordinates [degrees]
- **lat** (*array or number*) – Latitudes of pixel/pixels to be corrected, in geodetic coordinates [degrees]
- **height** (*array or number*) – Heights of pixels on which the correction will be based. Typically this is the cloud top height. [m]

Returns

Corrected geolocation
Corrected geolocation `(lon, lat)` in geodetic coordinates for the pixel(s) to be corrected. [degrees]

Return type
tuple[float, float]

satpy.modifiers.parallax.**get_surface_parallax_displacement**(*sat_lon*, *sat_lat*, *sat_alt*, *lon*, *lat*, *height*)

Calculate surface parallax displacement.

Calculate the displacement due to parallax error. Input parameters are identical to *get_parallax_corrected_lonlats()*.

Returns
parallax displacement in meter

Return type
number or array

satpy.modifiers.spectral module

Modifier classes dealing with spectral domain changes or corrections.

class satpy.modifiers.spectral.**NIREmissivePartFromReflectance**(*sunz_threshold=None*, ***kwargs*)

Bases: *NIRReflectance*

Get the emissive part of NIR bands.

Collect custom configuration values.

Parameters
sunz_threshold – The threshold sun zenith angle used when deriving the near infrared reflectance. Above this angle the derivation will assume this sun-zenith everywhere. Default None, in which case the default threshold defined in Pyspectral will be used.

_get_emissivity_as_dask(*da_nir*, *da_tb11*, *da_tb13_4*, *da_sun_zenith*, *metadata*)

Get the emissivity from pyspectral.

_get_emissivity_as_dataarray(*nir*, *da_tb11*, *da_tb13_4*, *da_sun_zenith*)

Get the emissivity as a dataarray.

class satpy.modifiers.spectral.**NIRReflectance**(*sunz_threshold=85.0*, *masking_limit=88.0*, ***kwargs*)

Bases: *ModifierBase*

Get the reflective part of NIR bands.

Collect custom configuration values.

Parameters

- **sunz_threshold** – The threshold sun zenith angle used when deriving the near infrared reflectance. Above this angle the derivation will assume this sun-zenith everywhere. Unless overridden, the default threshold of 85.0 degrees will be used.
- **masking_limit** – Mask the data (set to NaN) above this Sun zenith angle. By default the limit is at 88.0 degrees. If set to *None*, no masking is done.

MASKING_LIMIT = 88.0

TERMINATOR_LIMIT = 85.0

_create_modified_dataarray(*reflectance*, *base_dataarray*)

_get_nir_inputs(*projectables*, *optional_datasets*)

_get_reflectance_as_dask(*da_nir*, *da_tb11*, *da_tb13_4*, *da_sun_zenith*, *metadata*)

Calculate 3.x reflectance in % with pyspectral from dask arrays.

_get_reflectance_as_dataarray(*nir*, *da_tb11*, *da_tb13_4*, *da_sun_zenith*)

Get the reflectance as a dataarray.

static _get_sun_zenith_from_provided_data(*nir*, *optional_datasets*, *dtype*)

Get the sunz from available data or compute it if unavailable.

static _get_tb13_4_from_optionals(*optional_datasets*)

_init_reflectance_calculator(*metadata*)

Initialize the 3.x reflectance derivations.

Module contents

Modifier classes and other related utilities.

satpy.multiscene package

Submodules

satpy.multiscene._blend_funcs module

satpy.multiscene._blend_funcs.**_combine_stacked_attrs**(*collected_attrs: Sequence[Mapping]*, *combine_times: bool*) → dict

satpy.multiscene._blend_funcs.**_fill_weights_for_invalid_dataset_pixels**(*datasets: Sequence[DataArray]*, *weights: Sequence[DataArray]*) → Iterable[DataArray]

Replace weight valus with 0 where data values are invalid/null.

satpy.multiscene._blend_funcs.**_get_combined_start_end_times**(*metadata_objects: Iterable[Mapping]*) → tuple[datetime | None, datetime | None]

Get the start and end times attributes valid for the entire dataset series.

satpy.multiscene._blend_funcs.**_get_weighted_blending_func**(*blend_type: str*) → Callable

satpy.multiscene._blend_funcs.**_stack_blend_by_weights**(*datasets: Sequence[DataArray], weights: Sequence[DataArray], combine_times: bool*) → DataArray

Stack datasets blending overlap using weights.

satpy.multiscene._blend_funcs.**_stack_no_weights**(*datasets: Sequence[DataArray], combine_times: bool*) → DataArray

satpy.multiscene._blend_funcs.**_stack_select_by_weights**(*datasets: Sequence[DataArray], weights: Sequence[DataArray], combine_times: bool*) → DataArray

Stack datasets selecting pixels using weights.

satpy.multiscene._blend_funcs.**_stack_with_weights**(*datasets: Sequence[DataArray], weights: Sequence[DataArray], combine_times: bool, blend_type: str*) → DataArray

satpy.multiscene._blend_funcs.**stack**(*data_arrays: Sequence[DataArray], weights: Sequence[DataArray] | None = None, combine_times: bool = True, blend_type: str = 'select_with_weights'*) → DataArray

Combine a series of datasets in different ways.

By default, DataArrays are stacked on top of each other, so the last one applied is on top. Each DataArray is assumed to represent the same geographic region, meaning they have the same area. If a sequence of weights is provided then they must have the same shape as the area. Weights with greater than 2 dimensions are not currently supported.

When weights are provided, the DataArrays will be combined according to those weights. Data can be integer category products (ex. cloud type), single channels (ex. radiance), or a multi-band composite (ex. an RGB or RGBA true_color). In the latter case, the weight array is applied to each band (R, G, B, A) in the same way. The result will be a composite DataArray where each pixel is constructed in a way depending on `blend_type`.

Blend type can be one of the following:

- select_with_weights: The input pixel with the maximum weight is chosen.
- blend_with_weights: The final pixel is a weighted average of all valid input pixels.

satpy.multiscene._blend_funcs.**temporal_rgb**(*data_arrays: Sequence[DataArray]*) → DataArray

Combine a series of datasets as a temporal RGB.

The first dataset is used as the Red component of the new composite, the second as Green and the third as Blue. All the other datasets are discarded.

satpy.multiscene._blend_funcs.**timeseries**(*datasets*)

Expand dataset with and concatenate by time dimension.

satpy.multiscene._multiscene module

MultiScene object to work with multiple timesteps of satellite data.

class satpy.multiscene._multiscene.**MultiScene**(*scenes=None*)

Bases: object

Container for multiple *Scene* objects.

Initialize MultiScene and validate sub-scenes.

Parameters

scenes (*iterable*) – *Scene* objects to operate on (optional)

Note: If the *scenes* passed to this object are a generator then certain operations performed will try to preserve that generator state. This may limit what properties or methods are available to the user. To avoid this behavior compute the passed generator by converting the passed scenes to a list first: `MultiScene(list(scenes))`.

_all_same_area(*dataset_ids*)

Return True if all areas for the provided IDs are equal.

static _call_scene_func(*gen*, *func_name*, *create_new_scene*, **args*, ***kwargs*)

Abstract method for running a Scene method on each Scene.

_distribute_frame_compute(*writers*, *frame_keys*, *frames_to_write*, *client*, *batch_size=1*)

Use `dask.distributed` to compute multiple frames at a time.

_distribute_save_datasets(*scenes_iter*, *client*, *batch_size=1*, ***kwargs*)

Distribute save_datasets across a cluster.

static _format_decoration(*ds*, *decorate*)

Maybe format decoration.

If the nested dictionary in decorate (argument to `save_animation`) contains a text to be added, format those based on dataset parameters.

_generate_scene_func(*gen*, *func_name*, *create_new_scene*, **args*, ***kwargs*)

Abstract method for running a Scene method on each Scene.

Additionally, modifies current MultiScene or creates a new one if needed.

_get_animation_frames(*all_datasets*, *shape*, *fill_value=None*, *ignore_missing=False*, *enh_args=None*)

Create enhanced image frames to save to a file.

_get_animation_info(*all_datasets*, *filename*, *fill_value=None*)

Determine filename and shape of animation to be created.

_get_client(*client=True*)

Determine what dask distributed client to use.

_get_single_frame(*ds*, *enh_args*, *fill_value*)

Get single frame from dataset.

Yet a single image frame from a dataset.

_get_writers_and_frames(*filename*, *datasets*, *fill_value*, *ignore_missing*, *enh_args*, *imio_args*)

Get writers and frames.

Helper function for save_animation.

static _simple_frame_compute(*writers*, *frame_keys*, *frames_to_write*)

Compute frames the plain dask way.

_simple_save_datasets(*scenes_iter*, ***kwargs*)

Run save_datasets on each Scene.

property all_same_area

Determine if all contained Scenes have the same 'area'.

blend(*blend_function: Callable[[...], DataArray] | None = None*) → *Scene*

Blend the datasets into one scene.

Reduce the `MultiScene` to a single `Scene`. Datasets occurring in each scene will be passed to a blending function, which shall take as input a list of datasets (`xarray.DataArray` objects) and shall return a single dataset (`xarray.DataArray` object). The blend method then assigns those datasets to the blended scene.

Blending functions provided in this module are `stack()` (the default), `timeseries()`, and `temporal_rgb()`, but the Python built-in function `sum()` also works and may be appropriate for some types of data.

Note: Blending is not currently optimized for generator-based MultiScene.

---

crop(**args*, ***kwargs*)

Crop the multiscene and return a new cropped multiscene.

property first_scene

First Scene of this MultiScene object.

classmethod from_files(*files_to_sort: Collection[str], reader: str | Collection[str] | None = None, ensure_all_readers: bool = False, scene_kwargs: Mapping | None = None, **kwargs*)

Create multiple Scene objects from multiple files.

Parameters

- **files_to_sort** – files to read
- **reader** – reader or readers to use
- **ensure_all_readers** – If True, limit to scenes where all readers have at least one file. If False (default), include all scenes where at least one reader has at least one file.
- **scene_kwargs** – additional arguments to pass on to `Scene.__init__()` for each created scene.

This uses the `satpy.readers.group_files()` function to group files. See this function for more details on additional possible keyword arguments. In particular, it is strongly recommended to pass *"group_keys"* when using multiple instruments.

New in version 0.12.

group(*groups*)

Group datasets from the multiple scenes.

By default, *MultiScene* only operates on dataset IDs shared by all scenes. Using this method you can specify groups of datasets that shall be treated equally by *MultiScene*. Even if their dataset IDs differ (for example because the names or wavelengths are slightly different). Groups can be specified as a dictionary *{group_id: dataset_names}* where the keys must be of type *DataQuery*, for example:

```
groups={
    DataQuery('my_group', wavelength=(10, 11, 12)): ['IR_108', 'B13', 'C13']
}
```

property is_generator

Contained Scenes are stored as a generator.

load(**args*, ***kwargs*)

Load the required datasets from the multiple scenes.

property loaded_dataset_ids

Union of all Dataset IDs loaded by all children.

resample(*destination=None*, ***kwargs*)

Resample the multiscene.

save_animation(*filename*, *datasets=None*, *fps=10*, *fill_value=None*, *batch_size=1*, *ignore_missing=False*, *client=True*, *enh_args=None*, ***kwargs*)

Save series of Scenes to movie (MP4) or GIF formats.

Supported formats are dependent on the *imageio* library and are determined by filename extension by default.

Note: Starting with `imageio` 2.5.0, the use of FFMPEG depends on a separate `imageio-ffmpeg` package.

By default all datasets available will be saved to individual files using the first Scene's datasets metadata to format the filename provided. If a dataset is not available from a Scene then a black array is used instead (np.zeros(shape)).

This function can use the `dask.distributed` library for improved performance by computing multiple frames at a time (see *batch_size* option below). If the distributed library is not available then frames will be generated one at a time, one product at a time.

Parameters

- **filename** (*str*) – Filename to save to. Can include python string formatting keys from dataset `.attrs` (ex. "{name}_{start_time:%Y%m%d_%H%M%S.gif")
- **datasets** (*list*) – DataIDs to save (default: all datasets)
- **fps** (*int*) – Frames per second for produced animation
- **fill_value** (*int*) – Value to use instead creating an alpha band.
- **batch_size** (*int*) – Number of frames to compute at the same time. This only has effect if the *dask.distributed* package is installed. This will default to 1. Setting this to 0 or less will attempt to process all frames at once. This option should be used with care to avoid memory issues when trying to improve performance. Note that this is the total number of frames for all datasets, so when saving 2 datasets this will compute `(batch_size / 2)` frames for the first dataset and `(batch_size / 2)` frames for the second dataset.
- **ignore_missing** (*bool*) – Don't include a black frame when a dataset is missing from a child scene.
- **client** (*bool or dask.distributed.Client*) – Dask distributed client to use for computation. If this is `True` (default) then any existing clients will be used. If this is `False` or `None` then a client will not be created and `dask.distributed` will not be used. If this is a dask `Client` object then it will be used for distributed computation.

- **enh_args** (*Mapping*) – Optional, arguments passed to *satpy.writers.get_enhanced_image()*. If this includes a keyword "decorate", in any text added to the image, string formatting will be applied based on dataset attributes. For example, passing `enh_args={"decorate": {"decorate": [{"text": {"txt": "{start_time:%H:%M}"}}]}}` will replace the decorated text accordingly.
- **kwargs** – Additional keyword arguments to pass to *imageio.get_writer*.

save_datasets(*client=True*, *batch_size=1*, ***kwargs*)

Run save_datasets on each Scene.

Note that some writers may not be multi-process friendly and may produce unexpected results or fail by raising an exception. In these cases `client` should be set to `False`. This is currently a known issue for basic 'geotiff' writer work loads.

Parameters

- **batch_size** (*int*) – Number of scenes to compute at the same time. This only has effect if the *dask.distributed* package is installed. This will default to 1. Setting this to 0 or less will attempt to process all scenes at once. This option should be used with care to avoid memory issues when trying to improve performance.
- **client** (*bool or dask.distributed.Client*) – Dask distributed client to use for computation. If this is `True` (default) then any existing clients will be used. If this is `False` or `None` then a client will not be created and `dask.distributed` will not be used. If this is a dask `Client` object then it will be used for distributed computation.
- **kwargs** – Additional keyword arguments to pass to *save_datasets()*. Note `compute` can not be provided.

property scenes

Get list of Scene objects contained in this MultiScene.

Note: If the Scenes contained in this object are stored in a generator (not list or tuple) then accessing this property will load/iterate through the generator possibly

property shared_dataset_ids

Dataset IDs shared by all children.

class satpy.multiscene._multiscene.**_GroupAliasGenerator**(*scene*, *groups*)

Bases: `object`

Add group aliases to a scene.

Initialize the alias generator.

_drop_id_attrs(*dataset*)

_duplicate_dataset_with_different_id(*dataset_id*, *alias_id*)

_duplicate_dataset_with_group_alias(*group_id*, *group_members*)

_get_dataset_id_of_group_members_in_scene(*group_members*)

_get_id_attrs(*dataset*)

_prepare_dataset_for_duplication(*dataset*, *alias_id*)

duplicate_datasets_with_group_alias()

Duplicate datasets to be grouped with a group alias.

class satpy.multiscene._multiscene.**_SceneGenerator**(*scene_gen*)

Bases: object

Fancy way of caching Scenes from a generator.

_create_cached_iter()

Iterate over the provided scenes, caching them for later.

property first

First element in the generator.

satpy.multiscene._multiscene.**_group_datasets_in_scenes**(*scenes*, *groups*)

Group different datasets in multiple scenes by adding aliases.

Parameters

- **scenes** (*iterable*) – Scenes to be processed.
- **groups** (*dict*) – Groups of datasets that shall be treated equally by MultiScene. Keys specify the groups, values specify the dataset names to be grouped. For example:

```
from satpy import DataQuery
groups = {DataQuery(name='odd'): ['ds1', 'ds3'],
          DataQuery(name='even'): ['ds2', 'ds4']}
```

Module contents

Functions and classes related to MultiScene functionality.

satpy.readers package

Subpackages

satpy.readers.gms package

Submodules

satpy.readers.gms.gms5_vissr_format module

GMS-5 VISSR archive data format.

Reference: VISSR Format Description

satpy.readers.gms.gms5_vissr_l1b module

Reader for GMS-5 VISSR Level 1B data.

Introduction

The `gms5_vissr_l1b` reader can decode, navigate and calibrate Level 1B data from the Visible and Infrared Spin Scan Radiometer (VISSR) in *VISSR archive format*. Corresponding platforms are GMS-5 (Japanese Geostationary Meteorological Satellite) and GOES-09 (2003-2006 backup after MTSAT-1 launch failure).

VISSR has four channels, each stored in a separate file:

```
VISSR_20020101_0031_IR1.A.IMG
VISSR_20020101_0031_IR2.A.IMG
VISSR_20020101_0031_IR3.A.IMG
VISSR_20020101_0031_VIS.A.IMG
```

This is how to read them with Satpy:

```
from satpy import Scene
import glob

filenames = glob.glob(""/data/VISSR*")
scene = Scene(filenames, reader="gms5-vissr_l1b")
scene.load(["VIS", "IR1"])
```

References:

Details about platform, instrument and data format can be found in the following references:

- VISSR Format Description
- GMS User Guide

Compression

Gzip-compressed VISSR files can be decompressed on the fly using *FSFile*:

```
import fsspec
from satpy import Scene
from satpy.readers import FSFile

filename = "VISSR_19960217_2331_IR1.A.IMG.gz"
open_file = fsspec.open(filename, compression="gzip")
fs_file = FSFile(open_file)
scene = Scene([fs_file], reader="gms5-vissr_l1b")
scene.load(["IR1"])
```

Calibration

Sensor counts are calibrated by looking up reflectance/temperature values in the calibration tables included in each file. See section 2.2 in the VISSR user guide.

Navigation

VISSR images are oversampled and not rectified.

Oversampling

VISSR oversamples the viewed scene in E-W direction by a factor of ~1.46: IR/VIS pixels are 14/3.5 urad on a side, but the instrument samples every 9.57/2.39 urad in E-W direction. That means pixels are actually overlapping on the ground.

This cannot be represented by a pyresample area definition, so each dataset is accompanied by 2-dimensional longitude and latitude coordinates. For resampling purpose a full disc area definition with uniform sampling is provided via

```
scene[dataset].attrs["area_def_uniform_sampling"]
```

Rectification

VISSR images are not rectified. That means lon/lat coordinates are different

1) for all channels of the same repeat cycle, even if their spatial resolution is identical (IR channels)
2) for different repeat cycles, even if the channel is identical

However, the above area definition is using the nominal subsatellite point as projection center. As this rarely changes, the area definition is pretty constant.

Performance

Navigation of VISSR images is computationally expensive, because for each pixel the view vector of the (rotating) instrument needs to be intersected with the earth, including interpolation of attitude and orbit prediction. For IR channels this takes about 10 seconds, for VIS channels about 160 seconds.

Space Pixels

VISSR produces data for pixels outside the Earth disk (i.e. atmospheric limb or deep space pixels). By default, these pixels are masked out as they contain data of limited or no value, but some applications do require these pixels. To turn off masking, set `mask_space=False` upon scene creation:

```
import satpy
import glob

filenames = glob.glob("VISSR*.IMG")
scene = satpy.Scene(filenames,
                    reader="gms5-vissr_l1b",
```

(continues on next page)

(continued from previous page)

```
                    reader_kwargs={"mask_space": False})
scene.load(["VIS", "IR1])
```

Metadata

Dataset attributes include metadata such as time and orbital parameters, see *Metadata*.

Partial Scans

Between 2001 and 2003 VISSR also recorded partial scans of the northern hemisphere. On demand a special Typhoon schedule would be activated between 03:00 and 05:00 UTC.

class satpy.readers.gms.gms5_vissr_l1b.**AreaDefEstimator**(*coord_conv_params*, *metadata*)

Bases: object

Estimate area definition for VISSR images.

Initialize the area definition estimator.

Parameters

- **coord_conv_params** – Coordinate conversion parameters
- **metadata** – VISSR file metadata

_get_name_dict(*dataset_id*)

_get_proj4_dict()

_get_proj_dict(*dataset_id*)

_get_shape_dict(*dataset_id*)

full_disk_size = {'IR': 2366, 'VIS': 9464}

get_area_def_uniform_sampling(*dataset_id*)

Get full disk area definition with uniform sampling.

Parameters
dataset_id – ID of the corresponding dataset.

class satpy.readers.gms.gms5_vissr_l1b.**Calibrator**(*calib_table*)

Bases: object

Calibrate VISSR data to reflectance or brightness temperature.

Reference: Section 2.2 in the VISSR User Guide.

Initialize the calibrator.

Parameters
calib_table – Calibration table

_calibrate(*counts*)

_convert_to_percent(*res*)

_lookup_calib_table(*counts*, *calib_table*)

_make_data_array(*interp*, *counts*)

_postproc(*res*, *calibration*)

calibrate(*counts*, *calibration*)

Transform counts to given calibration level.

class satpy.readers.gms.gms5_vissr_l1b.**GMS5VISSRFileHandler**(*filename*, *filename_info*, *filetype_info*, *mask_space=True*)

Bases: BaseFileHandler

File handler for GMS-5 VISSR data in VISSR archive format.

Initialize the file handler.

Parameters

- **filename** – Name of file to be read
- **filename_info** – Information obtained from filename
- **filetype_info** – Information about file type
- **mask_space** – Mask space pixels.

_attach_lons_lats(*dataset*, *dataset_id*)

_calibrate(*counts*, *dataset_id*)

static _concat_orbit_prediction(*orb_pred_1*, *orb_pred_2*)

Concatenate orbit prediction data.

It is split over two image parameter blocks in the header.

property _coord_conv

_get_acq_time(*dask_array*)

_get_actual_shape()

_get_area_def_uniform_sampling(*dataset_id*)

_get_attitude_prediction()

_get_calibration_table(*dataset_id*)

static _get_channel_type(*parameter_block_size*)

_get_counts(*image_data*)

_get_earth_ellipsoid()

_get_frame_parameters_key()

_get_image_coords(*data*)

_get_image_data()

_get_image_data_type_specs()

_get_image_offset(*dataset_id*)

_get_line_number(*dask_array*)

_get_lons_lats(*dataset*, *dataset_id*)

_get_mda()

_get_navigation_parameters(*dataset_id*)

_get_nominal_shape()

_get_orbit_prediction()

_get_orbital_parameters()

_get_predicted_navigation_params()

Get predictions of time-dependent navigation parameters.

_get_proj_params(*dataset_id*)

_get_scanning_angles(*dataset_id*)

_get_static_navigation_params(*dataset_id*)

Get static navigation parameters.

Note that, "central_line_number_of_vissr_frame" is different for each channel, even if their spatial resolution is identical. For example:

VIS: 5513.0 IR1: 1378.5 IR2: 1378.7 IR3: 1379.1001

_get_time_parameters()

_make_counts_data_array(*image_data*)

_make_lons_lats_data_array(*lons*, *lats*)

_mask_space_pixels(*dataset*, *space_masker*)

property _mode_block

_read_control_block(*file_obj*)

_read_header(*filename*)

_read_image_data()

static _read_image_param(*file_obj*, *param*, *channel_type*)

Read a single image parameter block from the header.

_read_image_params(*file_obj*, *channel_type*)

Read image parameters from the header.

_update_attrs(*dataset*, *dataset_id*, *ds_info*)

property end_time

Nominal end time of the dataset.

get_dataset(*dataset_id*, *ds_info*)

Get dataset from file.

property start_time

Nominal start time of the dataset.

class satpy.readers.gms.gms5_vissr_l1b.**SpaceMasker**(*image_data*, *channel*)

Bases: object

Mask pixels outside the earth disk.

Initialize the space masker.

Parameters

- **image_data** – Image data
- **channel** – Channel name

_correct_vis_edges(*edges*)

Correct VIS edges.

VIS data contains earth edges of IR channel. Compensate for that by scaling with a factor of 4 (1 IR pixel ~ 4 VIS pixels).

_fill_value = -1

_get_earth_edges()

_get_earth_edges_per_scan_line(*cardinal*)

_get_earth_mask()

mask_space(*dataset*)

Mask space pixels in the given dataset.

satpy.readers.gms.gms5_vissr_l1b.**_get_alternative_channel_name**(*dataset_id*)

satpy.readers.gms.gms5_vissr_l1b.**_recarr2dict**(*arr*, *preserve=None*)

satpy.readers.gms.gms5_vissr_l1b.**get_earth_mask**(*shape*, *earth_edges*, *fill_value=-1*)

Get binary mask where 1/0 indicates earth/space.

Parameters

- **shape** – Image shape
- **earth_edges** – First and last earth pixel in each scanline
- **fill_value** – Fill value for scanlines not intersecting the earth.

satpy.readers.gms.gms5_vissr_l1b.**is_vis_channel**(*channel_name*)

Check if it's the visible channel.

satpy.readers.gms.gms5_vissr_l1b.**read_from_file_obj**(*file_obj*, *dtype*, *count*, *offset=0*)

Read data from file object.

Parameters

- **file_obj** – An open file object.
- **dtype** – Data type to be read.
- **count** – Number of elements to be read.
- **offset** – Byte offset where to start reading.

satpy.readers.gms.gms5_vissr_navigation module

GMS-5 VISSR Navigation.

Reference: GMS User Guide, Appendix E, S-VISSR Mapping.

class satpy.readers.gms.gms5_vissr_navigation.**Attitude**(*angle_between_earth_and_sun*, *angle_between_sat_spin_and_z_axis*, *angle_between_sat_spin_and_yz_plane*)

Bases: tuple

Attitude parameters.

Units: radians

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('angle_between_earth_and_sun', 'angle_between_sat_spin_and_z_axis', 'angle_between_sat_spin_and_yz_plane')

classmethod _make(*iterable*)

Make a new Attitude object from a sequence or iterable

_replace(***kwds*)

Return a new Attitude object replacing specified fields with new values

angle_between_earth_and_sun

Alias for field number 0

angle_between_sat_spin_and_yz_plane

Alias for field number 2

angle_between_sat_spin_and_z_axis

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**AttitudePrediction**(*prediction_times*, *attitude*)

Bases: object

Attitude prediction.

Use .to_numba() to pass this object to jitted methods. This extra layer avoids usage of jitclasses and having to re-implement np.unwrap in numba.

Initialize attitude prediction.

In order to accelerate interpolation, the 2-pi periodicity of angles is unwrapped here already (that means phase jumps greater than pi are wrapped to their 2*pi complement).

Parameters

- **prediction_times** – Timestamps of predicted attitudes
- **attitude** (Attitude) – Attitudes at prediction times

_unwrap_angles(*attitude*)

to_numba()

Convert to numba-compatible type.

satpy.readers.gms.gms5_vissr_navigation.**EARTH_POLAR_RADIUS = 6356751.301568781**

Constants taken from JMA's Msial library.

class satpy.readers.gms.gms5_vissr_navigation.**EarthEllipsoid**(*flattening*, *equatorial_radius*)

Bases: tuple

Earth ellipsoid.

Parameters

- **flattening** – Ellipsoid flattening
- **equatorial_radius** – Equatorial radius (meters)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('flattening', 'equatorial_radius')

classmethod _make(*iterable*)

Make a new EarthEllipsoid object from a sequence or iterable

_replace(**kwds*)

Return a new EarthEllipsoid object replacing specified fields with new values

equatorial_radius

Alias for field number 1

flattening

Alias for field number 0

class satpy.readers.gms.gms5_vissr_navigation.**ImageNavigationParameters**(*static*, *predicted*)

Bases: tuple

Navigation parameters for the entire image.

Parameters

- **static** (StaticNavigationParameters) – Static parameters.
- **predicted** (PredictedNavigationParameters) – Predicted time-dependent parameters.

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('static', 'predicted')

classmethod _make(*iterable*)

Make a new ImageNavigationParameters object from a sequence or iterable

_replace(**kwds*)

Return a new ImageNavigationParameters object replacing specified fields with new values

predicted

Alias for field number 1

static

Alias for field number 0

class satpy.readers.gms.gms5_vissr_navigation.**ImageOffset**(*line_offset*, *pixel_offset*)

Bases: tuple

Image offset

Parameters

- **line_offset** – Line offset from image center
- **pixel_offset** – Pixel offset from image center

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('line_offset', 'pixel_offset')

classmethod _make(*iterable*)

Make a new ImageOffset object from a sequence or iterable

_replace(**kwds*)

Return a new ImageOffset object replacing specified fields with new values

line_offset

Alias for field number 0

pixel_offset

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**Orbit**(*angles*, *sat_position*, *nutation_precession*)

Bases: tuple

Orbital Parameters

Parameters

- **angles** (OrbitAngles) – Orbit angles
- **sat_position** (Vector3D) – Satellite position
- **nutation_precession** – Nutation and precession matrix (3x3)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('angles', 'sat_position', 'nutation_precession')

classmethod _make(*iterable*)

Make a new Orbit object from a sequence or iterable

_replace(**kwds*)

Return a new Orbit object replacing specified fields with new values

angles

Alias for field number 0

nutation_precession

Alias for field number 2

sat_position

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**OrbitAngles**(*greenwich_sidereal_time*, *declination_from_sat_to_sun*, *right_ascension_from_sat_to_sun*)

Bases: tuple

Orbit angles.

Units: radians

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('greenwich_sidereal_time', 'declination_from_sat_to_sun', 'right_ascension_from_sat_to_sun')

classmethod _make(*iterable*)

Make a new OrbitAngles object from a sequence or iterable

_replace(***kwds*)

Return a new OrbitAngles object replacing specified fields with new values

declination_from_sat_to_sun

Alias for field number 1

greenwich_sidereal_time

Alias for field number 0

right_ascension_from_sat_to_sun

Alias for field number 2

class satpy.readers.gms.gms5_vissr_navigation.**OrbitPrediction**(*prediction_times*, *angles*, *sat_position*, *nutation_precession*)

Bases: object

Orbit prediction.

Use .to_numba() to pass this object to jitted methods. This extra layer avoids usage of jitclasses and having to re-implement np.unwrap in numba.

Initialize orbit prediction.

In order to accelerate interpolation, the 2-pi periodicity of angles is unwrapped here already (that means phase jumps greater than pi are wrapped to their 2*pi complement).

Parameters

- **prediction_times** – Timestamps of orbit prediction.
- **angles** (OrbitAngles) – Orbit angles
- **sat_position** (Vector3D) – Satellite position
- **nutation_precession** – Nutation and precession matrix.

_unwrap_angles(*angles*)

to_numba()

Convert to numba-compatible type.

class satpy.readers.gms.gms5_vissr_navigation.**Pixel**(*line*, *pixel*)

Bases: tuple

A VISSR pixel.

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('line', 'pixel')

classmethod _make(*iterable*)

Make a new Pixel object from a sequence or iterable

_replace(***kwds*)

Return a new Pixel object replacing specified fields with new values

line

Alias for field number 0

pixel

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**PixelNavigationParameters**(*attitude*, *orbit*, *proj_params*)

Bases: tuple

Navigation parameters for a single pixel.

Parameters

- **attitude** (Attitude) – Attitude parameters
- **orbit** (Orbit) – Orbit parameters
- **proj_params** (ProjectionParameters) – Projection parameters

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('attitude', 'orbit', 'proj_params')

classmethod _make(*iterable*)

Make a new PixelNavigationParameters object from a sequence or iterable

_replace(***kwds*)

Return a new PixelNavigationParameters object replacing specified fields with new values

attitude

Alias for field number 0

orbit

Alias for field number 1

proj_params

Alias for field number 2

class satpy.readers.gms.gms5_vissr_navigation.**PredictedNavigationParameters**(*attitude*, *orbit*)

Bases: tuple

Predictions of time-dependent navigation parameters.

They need to be evaluated for each pixel.

Parameters

- **attitude** (AttitudePrediction) – Attitude prediction
- **orbit** (OrbitPrediction) – Orbit prediction

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('attitude', 'orbit')

classmethod _make(*iterable*)

Make a new PredictedNavigationParameters object from a sequence or iterable

_replace(***kwds*)

Return a new PredictedNavigationParameters object replacing specified fields with new values

attitude

Alias for field number 0

orbit

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**ProjectionParameters**(*image_offset*, *scanning_angles*, *earth_ellipsoid*)

Bases: tuple

Projection parameters.

Parameters

- **image_offset** (ImageOffset) – Image offset
- **scanning_angles** (ScanningAngles) – Scanning angles
- **earth_ellipsoid** (EarthEllipsoid) – Earth ellipsoid

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('image_offset', 'scanning_angles', 'earth_ellipsoid')

classmethod _make(*iterable*)

Make a new ProjectionParameters object from a sequence or iterable

_replace(***kwds*)

Return a new ProjectionParameters object replacing specified fields with new values

earth_ellipsoid

Alias for field number 2

image_offset

Alias for field number 0

scanning_angles

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**Satpos**(*x*, *y*, *z*)

Bases: tuple

A 3D vector.

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('x', 'y', 'z')

classmethod _make(*iterable*)

Make a new Satpos object from a sequence or iterable

_replace(***kwds*)

Return a new Satpos object replacing specified fields with new values

x

Alias for field number 0

y

Alias for field number 1

z

Alias for field number 2

class satpy.readers.gms.gms5_vissr_navigation.**ScanningAngles**(*stepping_angle*, *sampling_angle*, *misalignment*)

Bases: tuple

Scanning angles

Parameters

- **stepping_angle** – Scanning angle along line (rad)
- **sampling_angle** – Scanning angle along pixel (rad)
- **misalignment** – Misalignment matrix (3x3)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('stepping_angle', 'sampling_angle', 'misalignment')

classmethod _make(*iterable*)

Make a new ScanningAngles object from a sequence or iterable

_replace(***kwds*)

Return a new ScanningAngles object replacing specified fields with new values

misalignment

Alias for field number 2

sampling_angle

Alias for field number 1

stepping_angle

Alias for field number 0

class satpy.readers.gms.gms5_vissr_navigation.**ScanningParameters**(*start_time_of_scan*, *spinning_rate*, *num_sensors*, *sampling_angle*)

Bases: tuple

Create new instance of ScanningParameters(start_time_of_scan, spinning_rate, num_sensors, sampling_angle)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('start_time_of_scan', 'spinning_rate', 'num_sensors', 'sampling_angle')

classmethod _make(*iterable*)

Make a new ScanningParameters object from a sequence or iterable

_replace(***kwds*)

Return a new ScanningParameters object replacing specified fields with new values

num_sensors

Alias for field number 2

sampling_angle

Alias for field number 3

spinning_rate

Alias for field number 1

start_time_of_scan

Alias for field number 0

class satpy.readers.gms.gms5_vissr_navigation.**StaticNavigationParameters**(*proj_params*, *scan_params*)

Bases: tuple

Navigation parameters which are constant for the entire scan.

Parameters

- **proj_params** (ProjectionParameters) – Projection parameters
- **scan_params** (ScanningParameters) – Scanning parameters

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('proj_params', 'scan_params')

classmethod _make(*iterable*)

Make a new StaticNavigationParameters object from a sequence or iterable

_replace(***kwds*)

Return a new StaticNavigationParameters object replacing specified fields with new values

proj_params

Alias for field number 0

scan_params

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**Vector2D**(*x*, *y*)

Bases: tuple

A 2D vector.

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('x', 'y')

classmethod _make(*iterable*)

Make a new Vector2D object from a sequence or iterable

_replace(***kwds*)

Return a new Vector2D object replacing specified fields with new values

x

Alias for field number 0

y

Alias for field number 1

class satpy.readers.gms.gms5_vissr_navigation.**Vector3D**(*x*, *y*, *z*)

Bases: tuple

A 3D vector.

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('x', 'y', 'z')

classmethod _make(*iterable*)

Make a new Vector3D object from a sequence or iterable

_replace(***kwds*)

Return a new Vector3D object replacing specified fields with new values

x

Alias for field number 0

y

Alias for field number 1

z

Alias for field number 2

class satpy.readers.gms.gms5_vissr_navigation.**_AttitudePrediction**(*prediction_times*, *attitude*)

Bases: tuple

Create new instance of _AttitudePrediction(prediction_times, attitude)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('prediction_times', 'attitude')

classmethod _make(*iterable*)

Make a new _AttitudePrediction object from a sequence or iterable

_replace(**kwds*)

Return a new _AttitudePrediction object replacing specified fields with new values

attitude

Alias for field number 1

prediction_times

Alias for field number 0

class satpy.readers.gms.gms5_vissr_navigation.**_OrbitPrediction**(*prediction_times*, *angles*, *sat_position*, *nutation_precession*)

Bases: tuple

Create new instance of _OrbitPrediction(prediction_times, angles, sat_position, nutation_precession)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('prediction_times', 'angles', 'sat_position', 'nutation_precession')

classmethod _make(*iterable*)

Make a new _OrbitPrediction object from a sequence or iterable

_replace(**kwds*)

Return a new _OrbitPrediction object replacing specified fields with new values

angles

Alias for field number 1

nutation_precession

Alias for field number 3

prediction_times

Alias for field number 0

sat_position

Alias for field number 2

satpy.readers.gms.gms5_vissr_navigation.**_correct_nutation_precession**(*vector*, *nutation_precession*)

satpy.readers.gms.gms5_vissr_navigation.**_find_enclosing_index**(*x*, *x_sample*)

Find where x_sample encloses x.

satpy.readers.gms.gms5_vissr_navigation.**_get_abc_helper**(*view_vector*, *sat_pos*, *ellipsoid*)

Get a,b,c helper variables.

Reference: Appendix E, Equation (26) in the GMS user guide.

satpy.readers.gms.gms5_vissr_navigation.**_get_distance_to_intersection**(*view_vector*, *sat_pos*, *ellipsoid*)

Get distance to intersection with the earth.

If the instrument is pointing towards the earth, there will be two intersections with the surface. Choose the one on the instrument-facing side of the earth.

satpy.readers.gms.gms5_vissr_navigation.**_get_distances_to_intersections**(*view_vector*, *sat_pos*, *ellipsoid*)

Get distances to intersections with the earth's surface.

Returns

Distances to two intersections with the surface.

satpy.readers.gms.gms5_vissr_navigation.**_get_earth_fixed_coords**(*point*, *unit_vector_x*, *unit_vector_y*, *unit_vector_z*)

satpy.readers.gms.gms5_vissr_navigation.**_get_lons_lats_numba**(*lines_2d*, *pixels_2d*, *nav_params*)

satpy.readers.gms.gms5_vissr_navigation.**_get_map_blocks_kwargs**(*chunks*)

satpy.readers.gms.gms5_vissr_navigation.**_get_pixel_navigation_parameters**(*point*, *im_nav_params*)

satpy.readers.gms.gms5_vissr_navigation.**_get_relative_observation_time**(*point*, *scan_params*)

satpy.readers.gms.gms5_vissr_navigation.**_get_satellite_unit_vector_x**(*unit_vector_z*, *attitude*, *orbit*)

satpy.readers.gms.gms5_vissr_navigation.**_get_satellite_unit_vector_y**(*unit_vector_x*, *unit_vector_z*)

satpy.readers.gms.gms5_vissr_navigation.**_get_satellite_unit_vector_z**(*attitude*, *orbit*)

satpy.readers.gms.gms5_vissr_navigation.**_get_satellite_z_axis_1950**(*angle_between_sat_spin_and_z_axis*, *angle_between_sat_spin_and_yz_plane*)

Get satellite z-axis (spin) in mean of 1950 coordinates.

satpy.readers.gms.gms5_vissr_navigation.**_get_unit_vector_x**(*sat_sun_vec*, *unit_vector_z*, *angle_between_earth_and_sun*)

satpy.readers.gms.gms5_vissr_navigation.**_get_uz_cross_satsun**(*unit_vector_z*, *sat_sun_vec*)

satpy.readers.gms.gms5_vissr_navigation.**_get_vector_from_satellite_to_sun**(*declination_from_sat_to_sun*, *right_ascension_from_sat_to_sun*)

satpy.readers.gms.gms5_vissr_navigation.**_interpolate**(*x*, *x_sample*, *y_sample*)

satpy.readers.gms.gms5_vissr_navigation.**_interpolate_nearest**(*x*, *x_sample*, *y_sample*)

satpy.readers.gms.gms5_vissr_navigation.**_interpolate_orbit_angles**(*observation_time*, *orbit_prediction*)

satpy.readers.gms.gms5_vissr_navigation.**_interpolate_sat_position**(*observation_time*, *orbit_prediction*)

satpy.readers.gms.gms5_vissr_navigation.**_make_nav_params_numba_compatible**(*nav_params*)

satpy.readers.gms.gms5_vissr_navigation.**_rotate_to_greenwich**(*vector*, *greenwich_sidereal_time*)

satpy.readers.gms.gms5_vissr_navigation.**_wrap_2pi**(*values*)

Wrap values to interval [-pi, pi].

Source: https://stackoverflow.com/a/15927914/5703449

satpy.readers.gms.gms5_vissr_navigation.**cross_product**(*a*, *b*)

Compute vector product a x b.

satpy.readers.gms.gms5_vissr_navigation.**get_lon_lat**(*pixel*, *nav_params*)

Get longitude and latitude coordinates for a given image pixel.

Parameters

- **pixel** (Pixel) – Point in image coordinates.
- **nav_params** (PixelNavigationParameters) – Navigation parameters for a single pixel.

Returns

Longitude and latitude in degrees.

satpy.readers.gms.gms5_vissr_navigation.**get_lons_lats**(*lines*, *pixels*, *nav_params*)

Compute lon/lat coordinates given VISSR image coordinates.

Parameters

- **lines** – VISSR image lines
- **pixels** – VISSR image pixels
- **nav_params** – Image navigation parameters

satpy.readers.gms.gms5_vissr_navigation.**get_observation_time**(*point*, *scan_params*)

Calculate observation time of a VISSR pixel.

satpy.readers.gms.gms5_vissr_navigation.**interpolate_angles**(*x*, *x_sample*, *y_sample*)

Linear interpolation of angles.

Requires 2-pi periodicity to be unwrapped before (for performance reasons). Interpolated angles are wrapped back to [-pi, pi] to restore periodicity.

satpy.readers.gms.gms5_vissr_navigation.**interpolate_attitude_prediction**(*attitude_prediction*, *observation_time*)

Interpolate attitude prediction at given observation time.

satpy.readers.gms.gms5_vissr_navigation.**interpolate_continuous**(*x*, *x_sample*, *y_sample*)

Linear interpolation of continuous quantities.

Numpy equivalent would be np.interp(..., left=np.nan, right=np.nan), but numba currently doesn't support those keyword arguments.

satpy.readers.gms.gms5_vissr_navigation.**interpolate_navigation_prediction**(*attitude_prediction*, *orbit_prediction*, *observation_time*)

Interpolate predicted navigation parameters.

satpy.readers.gms.gms5_vissr_navigation.**interpolate_nearest**(*x*, *x_sample*, *y_sample*)

Nearest neighbour interpolation.

satpy.readers.gms.gms5_vissr_navigation.**interpolate_orbit_prediction**(*orbit_prediction*, *observation_time*)

Interpolate orbit prediction at the given observation time.

satpy.readers.gms.gms5_vissr_navigation.**intersect_with_earth**(*view_vector*, *sat_pos*, *ellipsoid*)

Intersect instrument viewing vector with the earth's surface.

Reference: Appendix E, section 2.11 in the GMS user guide.

Parameters

- **view_vector** (Vector3D) – Instrument viewing vector in earth-fixed coordinates.
- **sat_pos** (Vector3D) – Satellite position in earth-fixed coordinates.
- **ellipsoid** (EarthEllipsoid) – Earth ellipsoid.

Returns

Intersection (Vector3D) with the earth's surface.

satpy.readers.gms.gms5_vissr_navigation.**matrix_vector**(*m*, *v*)

Multiply (3,3)-matrix and Vector3D.

satpy.readers.gms.gms5_vissr_navigation.**normalize_vector**(*v*)

Normalize a Vector3D.

satpy.readers.gms.gms5_vissr_navigation.**transform_earth_fixed_to_geodetic_coords**(*point*, *earth_flattening*)

Transform from earth-fixed to geodetic coordinates.

Parameters

- **point** (Vector3D) – Point in earth-fixed coordinates.
- **earth_flattening** – Flattening of the earth.

Returns

Geodetic longitude and latitude (degrees).

satpy.readers.gms.gms5_vissr_navigation.**transform_image_coords_to_scanning_angles**(*point*, *image_offset*, *scanning_angles*)

Transform image coordinates to scanning angles.

Parameters

- **point** (Pixel) – Point in image coordinates.

- **image_offset** (ImageOffset) – Image offset.
- **scanning_angles** (ScanningAngles) – Scanning angles.

Returns
Scanning angles (x, y) at the pixel center (rad).

satpy.readers.gms.gms5_vissr_navigation.**transform_satellite_to_earth_fixed_coords**(*point*, *orbit*, *attitude*)

Transform from earth-fixed to satellite angular momentum coordinates.

Parameters

- **point** (Vector3D) – Point in satellite angular momentum coordinates.
- **orbit** (Orbit) – Orbital parameters
- **attitude** (Attitude) – Attitude parameters

Returns
Point (Vector3D) in earth-fixed coordinates.

satpy.readers.gms.gms5_vissr_navigation.**transform_scanning_angles_to_satellite_coords**(*angles*, *misalignment*)

Transform scanning angles to satellite angular momentum coordinates.

Parameters

- **angles** (Vector2D) – Scanning angles in radians.
- **misalignment** – Misalignment matrix (3x3)

Returns
View vector (Vector3D) in satellite angular momentum coordinates.

Module contents

GMS reader module.

Submodules

satpy.readers._geos_area module

Geostationary Projection / Area computations.

This module computes properties and area definitions for geostationary satellites. It is designed to be a common module that can be called by all geostationary satellite readers and uses commonly-included parameters such as the CFAC/LFAC values, satellite position, etc, to compute the correct area definition.

satpy.readers._geos_area.**get_area_definition**(*pdict*, *a_ext*)

Get the area definition for a geo-sat.

Parameters

- **pdict** – A dictionary containing common parameters: nlines: Number of lines in image ncols: Number of columns in image ssp_lon: Subsatellite point longitude (deg) a: Earth equatorial radius (m) b: Earth polar radius (m) h: Platform height (m) a_name: Area name a_desc: Area description p_id: Projection id
- **a_ext** – A four element tuple containing the area extent (scan angle) for the scene in radians

Returns
An area definition for the scene

Return type
a_def

Note: The AreaDefinition *proj_id* attribute is being deprecated.

---

satpy.readers._geos_area.**get_area_extent**(*pdict*)

Get the area extent seen by a geostationary satellite.

Parameters
pdict – A dictionary containing common parameters: nlines: Number of lines in image ncols: Number of columns in image cfac: Column scaling factor lfac: Line scaling factor coff: Column offset factor loff: Line offset factor scandir: 'N2S' for standard (N->S), 'S2N' for inverse (S->N)

Returns
An area extent for the scene

Return type
aex

satpy.readers._geos_area.**get_geos_area_naming**(*input_dict*)

Get a dictionary containing formatted AreaDefinition naming.

Parameters
input_dict – dict Dictionary with keys *platform_name*, *instrument_name*, *service_name*, *service_desc*, *resolution* . The resolution is expected in meters.

Returns
area_naming_dict with *area_id*, *description* keys, values are strings.

Note: The AreaDefinition *proj_id* attribute is being deprecated and is therefore not formatted here. An empty string is to be used until the attribute is fully removed.

---

satpy.readers._geos_area.**get_resolution_and_unit_strings**(*resolution*)

Get the resolution value and unit as strings.

If the resolution is larger than 1000 m, use kilometer as unit. If lower, use meter.

Parameters
resolution – scalar Resolution in meters.

Returns
Dictionary with *value* and *unit* keys, values are strings.

satpy.readers._geos_area.**get_xy_from_linecol**(*line*, *col*, *offsets*, *factors*)

Get the intermediate coordinates from line & col.

Intermediate coordinates are actually the instruments scanning angles.

satpy.readers._geos_area.**make_ext**(*ll_x*, *ur_x*, *ll_y*, *ur_y*, *h*)

Create the area extent from computed ll and ur.

Parameters

- **ll_x** – The lower left x coordinate (m)
- **ur_x** – The upper right x coordinate (m)
- **ll_y** – The lower left y coordinate (m)
- **ur_y** – The upper right y coordinate (m)
- **h** – The satellite altitude above the Earth's surface

Returns
An area extent for the scene

Return type
aex

satpy.readers._geos_area.**sampling_to_lfac_cfac**(*sampling*)

Convert angular sampling to line/column scaling factor (aka LFAC/CFAC).

Reference: MSG Ground Segment LRIT HRIT Mission Specific Implementation, Appendix E.2.

Parameters
sampling – float Angular sampling (rad)

Returns
Line/column scaling factor (deg-1)

satpy.readers.aapp_l1b module

Reader for aapp level 1b data.

Options for loading:

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

https://nwp-saf.eumetsat.int/site/download/documentation/aapp/NWPSAF-MF-UD-003_Formats_v8.0.pdf

class satpy.readers.aapp_l1b.**AAPPL1BaseFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

A base file handler for the AAPP level-1 formats.

Initialize AAPP level-1 file handler object.

_calibrate_active_channel_data(*key*)

Calibrate active channel data only.

_get_platform_name(*platform_names_lookup*)

Get the platform name from the file header.

_set_filedata_layout()

Set the file data type/layout.

_update_dataset_attributes(*dataset*, *key*, *info*)

property end_time

Get the time of the final observation.

get_dataset(*key*, *info*)

Get a dataset from the file.

read()

Read the data.

property start_time

Get the time of the first observation.

class satpy.readers.aapp_l1b.**AVHRRAAPPL1BFile**(*filename*, *filename_info*, *filetype_info*)

Bases: *AAPPL1BaseFileHandler*

Reader for AVHRR L1B files created from the AAPP software.

Initialize object information by reading the input file.

_calibrate_active_channel_data(*key*)

Calibrate active channel data only.

static _convert_binary_channel_status_to_activation_dict(*status*)

static _create_40km_interpolator(*lines*, **arrays_40km*, *geolocation=False*)

_get_active_channels()

_get_all_interpolated_angles_uncached()

_get_all_interpolated_coordinates_uncached()

_get_channel_binary_status_from_header()

_get_coordinates_in_degrees()

_get_tiepoint_angles_in_degrees()

_interpolate_arrays(**input_arrays*, *geolocation=False*)

_set_filedata_layout()

Set the file data type/layout.

available_datasets(*configured_datasets=None*)

Get the available datasets.

calibrate(*dataset_id*, *pre_launch_coeffs=False*, *calib_coeffs=None*)

Calibrate the data.

get_angles(*angle_id*)

Get sun-satellite viewing angles.

navigate(*coordinate_id*)

Get the longitudes and latitudes of the scene.

satpy.readers.aapp_l1b.**_ir_calibrate**(*header*, *data*, *irchn*, *calib_type*, *mask=True*)

Calibrate for IR bands.

calib_type in brightness_temperature, radiance, count

satpy.readers.aapp_l1b.**_vis_calibrate**(*data*, *chn*, *calib_type*, *pre_launch_coeffs=False*, *calib_coeffs=None*, *mask=True*)

Calibrate visible channel data.

calib_type in count, reflectance, radiance.

satpy.readers.aapp_l1b.**create_xarray**(*arr*)

Create an *xarray.DataArray*.

satpy.readers.aapp_l1b.**get_aapp_chunks**(*shape*)

Get chunks from a given shape adapted for AAPP data.

satpy.readers.aapp_l1b.**get_avhrr_lac_chunks**(*shape*, *dtype*)

Get chunks from a given shape adapted for full-resolution AVHRR data.

satpy.readers.aapp_mhs_amsub_l1c module

Reader for the AAPP AMSU-B/MHS level-1c data.

https://nwp-saf.eumetsat.int/site/download/documentation/aapp/NWPSAF-MF-UD-003_Formats_v8.0.pdf

class satpy.readers.aapp_mhs_amsub_l1c.**MHS_AMSUB_AAPPL1CFile**(*filename*, *filename_info*, *filetype_info*)

Bases: *AAPPL1BaseFileHandler*

Reader for AMSU-B/MHS L1C files created from the AAPP software.

Initialize object information by reading the input file.

_calibrate_active_channel_data(*key*)

Calibrate active channel data only.

_get_coordinates_in_degrees()

_get_sensorname()

Get the sensor name from the header.

_set_filedata_layout()

Set the file data type/layout.

calibrate(*dataset_id*)

Calibrate the data.

get_angles(*angle_id*)

Get sun-satellite viewing angles.

navigate(*coordinate_id*)

Get the longitudes and latitudes of the scene.

satpy.readers.aapp_mhs_amsub_l1c.**_calibrate**(*data*, *chn*, *calib_type*, *mask=True*)

Calibrate channel data.

calib_type in brightness_temperature.

satpy.readers.abi_base module

Advance Baseline Imager reader base class for the Level 1b and l2+ reader.

class satpy.readers.abi_base.**NC_ABI_BASE**(*filename*, *filename_info*, *filetype_info*)

Bases: BaseFileHandler

Base reader for ABI L1B L2+ NetCDF4 files.

Open the NetCDF file with xarray and prepare the Dataset for reading.

_adjust_coords(*data*, *item*)

Handle coordinates (and recursive fun).

_adjust_data(*data*, *item*)

Adjust data with typing, scaling and filling.

_chunk_bytes_for_resolution() → int

Get a best-guess optimal chunk size for resolution-based chunking.

First a chunk size is chosen for the provided Dask setting *array.chunk-size* and then aligned with a hardcoded on-disk chunk size of 226. This is then adjusted to match the current resolution.

This should result in 500 meter data having 4 times as many pixels per dask array chunk (2 in each dimension) as 1km data and 8 times as many as 2km data. As data is combined or upsampled geographically the arrays should not need to be rechunked. Care is taken to make sure that array chunks are aligned with on-disk file chunks at all resolutions, but at the cost of flexibility due to a hardcoded on-disk chunk size of 226 elements per dimension.

_get_areadef_fixedgrid(*key*)

Get the area definition of the data at hand.

Note this method takes special care to round and cast numbers to new data types so that the area definitions for different resolutions (different bands) should be equal. Without the special rounding in *__getitem__* and this method the area extents can be 0 to 1.0 meters off depending on how the calculations are done.

_get_areadef_latlon(*key*)

Get the area definition of the data at hand.

static _rename_dims(*nc*)

property end_time

End time of the current file's observations.

get_area_def(*key*)

Get the area definition of the data at hand.

get_dataset(*key*, *info*)

Load a dataset.

property nc

Get the xarray dataset for this file.

property sensor

Get sensor name for current file handler.

spatial_resolution_to_number()

Convert the 'spatial_resolution' global attribute to meters.

property start_time

Start time of the current file's observations.

satpy.readers.abi_l1b module

Advance Baseline Imager reader for the Level 1b format.

The files read by this reader are described in the official PUG document:

https://www.goes-r.gov/users/docs/PUG-L1b-vol3.pdf

class satpy.readers.abi_l1b.**NC_ABI_L1B**(*filename*, *filename_info*, *filetype_info*, *clip_negative_radiances=None*)

Bases: *NC_ABI_BASE*

File reader for individual ABI L1B NetCDF4 files.

Open the NetCDF file with xarray and prepare the Dataset for reading.

_adjust_attrs(*data*, *key*)

_get_minimum_radiance(*data*)

Estimate minimum radiance from Rad DataArray.

_ir_calibrate(*data*)

Calibrate IR channels to BT.

_rad_calibrate(*data*)

Calibrate any channel to radiances.

This no-op method is just to keep the flow consistent - each valid cal type results in a calibration method call

_raw_calibrate(*data*)

Calibrate any channel to raw counts.

Useful for cases where a copy requires no calibration.

_vis_calibrate(*data*)

Calibrate visible channels to reflectance.

get_dataset(*key*, *info*)

Load a dataset.

satpy.readers.abi_l2_nc module

Advance Baseline Imager NOAA Level 2+ products reader.

The files read by this reader are described in the official PUG document:

https://www.goes-r.gov/products/docs/PUG-L2+-vol5.pdf

class satpy.readers.abi_l2_nc.**NC_ABI_L2**(*filename*, *filename_info*, *filetype_info*)

Bases: *NC_ABI_BASE*

Reader class for NOAA ABI l2+ products in netCDF format.

Open the NetCDF file with xarray and prepare the Dataset for reading.

static _remove_problem_attrs(*variable*)

_update_data_arr_with_filename_attrs(*variable*)

available_datasets(*configured_datasets=None*)

Add resolution to configured datasets.

get_dataset(*key*, *info*)

Load a dataset.

satpy.readers.acspo module

ACSPO SST Reader.

See the following page for more information:

https://podaac.jpl.nasa.gov/dataset/VIIRS_NPP-OSPO-L2P-v2.3

class satpy.readers.acspo.**ACSPOFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

ACSPO L2P SST File Reader.

Initialize object.

static _parse_datetime(*datestr*)

property end_time

Get final observation time of data.

get_dataset(*dataset_id*, *ds_info*)

Load data array and metadata from file on disk.

get_metadata(*dataset_id*, *ds_info*)

Collect various metadata about the specified dataset.

get_shape(*ds_id*, *ds_info*)

Get numpy array shape for the specified dataset.

Parameters

- **ds_id** (*DataID*) – ID of dataset that will be loaded
- **ds_info** (*dict*) – Dictionary of dataset information from config file

Returns

(rows, cols)

Return type

tuple

property platform_name

Get satellite name for this file's data.

property sensor_name

Get instrument name for this file's data.

property start_time

Get first observation time of data.

satpy.readers.agri_l1 module

Advanced Geostationary Radiation Imager reader for the Level_1 HDF format.

The files read by this reader are described in the official Real Time Data Service:

http://fy4.nsmc.org.cn/data/en/data/realtime.html

class satpy.readers.agri_l1.**HDF_AGRI_L1**(*filename*, *filename_info*, *filetype_info*)

Bases: *FY4Base*

AGRI l1 file handler.

Init filehandler.

adjust_attrs(*data*, *ds_info*)

Adjust the attrs of the data.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset.

satpy.readers.ahi_hsd module

Advanced Himawari Imager (AHI) standard format data reader.

References

- Himawari-8/9 Himawari Standard Data User's Guide
- http://www.data.jma.go.jp/mscweb/en/himawari89/space_segment/spsg_ahi.html

Time Information

AHI observations use the idea of a "nominal" time and an "observation" time. The "nominal" time or repeat cycle is the overall window when the instrument can record data, usually at a specific and consistent interval. The "observation" time is when the data was actually observed inside the nominal window. These two times are stored in a sub-dictionary in the metadata calls `time_parameters`. Nominal time can be accessed from the `nominal_start_time` and `nominal_end_time` metadata keys and observation time from the `observation_start_time` and `observation_end_time` keys. Observation time can also be accessed from the parent (`.attrs`) dictionary as the `start_time` and `end_time` keys.

Satellite Position

As discussed in the *Orbital Parameters* documentation, a satellite position can be described by a specific "actual" position, a "nominal" position, a "projection" position, or sometimes a "nadir" position. Not all readers are able to produce all of these positions. In the case of AHI HSD data we have an "actual" and "projection" position. For a lot of sensors/readers though, the "actual" position values do not change between bands or segments of the same time step (repeat cycle). AHI HSD files contain varying values for the actual position.

Other components in Satpy use this actual satellite position to generate other values (ex. sensor zenith angles). If these values are not consistent between bands then Satpy (dask) will not be able to share these calculations (generate one sensor zenith angle for band 1, another for band 2, etc) even though there is rarely a noticeable difference. To deal with this this reader has an option `round_actual_position` that defaults to `True` and will round the "actual" position (longitude, latitude, altitude) in a way to produce as consistent a position between bands as possible.

class satpy.readers.ahi_hsd.**AHIHSDFileHandler**(*filename*, *filename_info*, *filetype_info*, *mask_space=True*, *calib_mode='update'*, *user_calibration=None*, *round_actual_position=True*)

Bases: *BaseFileHandler*

AHI standard format reader.

The AHI sensor produces data for some pixels outside the Earth disk (i,e: atmospheric limb or deep space pixels). By default, these pixels are masked out as they contain data of limited or no value, but some applications do require these pixels. It is therefore possible to override the default behaviour and perform no masking of non-Earth pixels.

In order to change the default behaviour, use the 'mask_space' variable as part of `reader_kwargs` upon Scene creation:

```
import satpy
import glob

filenames = glob.glob('*FLDK*.dat')
scene = satpy.Scene(filenames,
                    reader='ahi_hsd',
                    reader_kwargs={'mask_space': False})
scene.load([0.6])
```

The AHI HSD data files contain multiple VIS channel calibration coefficients. By default, the updated coefficients in header block 6 are used. If the user prefers the default calibration coefficients from block 5 then they can pass calib_mode='nominal' when creating a scene:

```
import satpy
import glob

filenames = glob.glob('*FLDK*.dat')
scene = satpy.Scene(filenames,
                    reader='ahi_hsd',
                    reader_kwargs={'calib_mode': 'update'})
scene.load([0.6])
```

Alternative AHI calibrations are also available, such as GSICS coefficients. As such, you can supply custom per-channel correction by setting calib_mode='custom' and passing correction factors via:

```
user_calibration={'chan': ['slope': slope, 'offset': offset]}
```

Where slo and off are per-channel slope and offset coefficients defined by:

```
rad_leo = (rad_geo - off) / slo
```

If you do not have coefficients for a particular band, then by default the slope will be set to 1 .and the offset to 0.:

```
import satpy
import glob

# Load bands 7, 14 and 15, but we only have coefs for 7+14
calib_dict = {'B07': {'slope': 0.99, 'offset': 0.002},
              'B14': {'slope': 1.02, 'offset': -0.18}}
```

(continues on next page)

(continued from previous page)

```
filenames = glob.glob('*FLDK*.dat')
scene = satpy.Scene(filenames,
                    reader='ahi_hsd',
                    reader_kwargs={'user_calibration': calib_dict)
# B15 will not have custom radiance correction applied.
scene.load(['B07', 'B14', 'B15'])
```

By default, user-supplied calibrations / corrections are applied to the radiance data in accordance with the GSICS standard defined in the equation above. However, user-supplied gain and offset values for converting digital number into radiance via Rad = DN * gain + offset are also possible. To supply your own factors, supply a user calibration dict using *type: 'DN'* as follows:

```
calib_dict = {'B07': {'slope': 0.0037, 'offset': 18.5},
              'B14': {'slope': -0.002, 'offset': 22.8},
              'type': 'DN'}
```

You can also explicitly select radiance correction with *'type': 'RAD'* but this is not necessary as it is the default option if you supply your own correction coefficients.

Initialize the reader.

_check_fpos(*fp_*, *fpos*, *offset*, *block*)

Check file position matches blocksize.

_get_area_def()

_get_metadata(*key*, *ds_info*)

_get_user_calibration_correction_type()

_ir_calibrate(*data*)

IR calibration.

static _is_valid_timeline(*timeline*)

Check that the *observation_timeline* value is not a fill value.

_mask_invalid(*data*, *header*)

Mask invalid data.

_mask_space(*data*)

Mask space pixels.

_modify_observation_time_for_nominal(*observation_time*)

Round observation time to a nominal time based on known observation frequency.

AHI observations are split into different sectors including Full Disk (FLDK), Japan (JP) sectors, and smaller regional (R) sectors. Each sector is observed at different frequencies (ex. every 10 minutes, every 2.5 minutes, and every 30 seconds). This method will take the actual observation time and round it to the nearest interval for this sector. So if the observation time is 13:32:48 for the "JP02" sector which is the second Japan observation where every Japan observation is 2.5 minutes apart, then the result should be 13:32:30.

_read_data(*fp_*, *header*, *resolution*)

Read data block.

_read_header(*fp_*)

Read header.

_vis_calibrate(*data*)

Visible channel calibration only.

property area

Get AreaDefinition representing this file's data.

calibrate(*data*, *calibration*)

Calibrate the data.

convert_to_radiance(*data*)

Calibrate to radiance.

property end_time

Get the nominal end time.

get_area_def(*dsid*)

Get the area definition.

get_dataset(*key*, *info*)

Get the dataset.

property nominal_end_time

Get the nominal end time.

property nominal_start_time

Time this band was nominally to be recorded.

property observation_end_time

Get the observation end time.

property observation_start_time

Get the observation start time.

read_band(*key*, *ds_info*)

Read the data.

property start_time

Get the nominal start time.

satpy.readers.ahi_l1b_gridded_bin module

Advanced Himawari Imager (AHI) gridded format data reader.

This data comes in a flat binary format on a fixed grid, and needs to have calibration coefficients applied to it in order to retrieve reflectance or BT. LUTs can be downloaded at: ftp://hmwr829gr.cr.chiba-u.ac.jp/gridded/FD/support/

This data is gridded from the original Himawari geometry. To our knowledge, only full disk grids are available, not for the Meso or Japan rapid scans.

References

- **AHI gridded data website:**
  http://www.cr.chiba-u.jp/databases/GEO/H8_9/FD/index_jp.html

class satpy.readers.ahi_l1b_gridded_bin.**AHIGriddedFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

AHI gridded format reader.

This data is flat binary, big endian unsigned short. It covers the region 85E -> 205E, 60N -> 60S at variable resolution: - 0.005 degrees for Band 3 - 0.01 degrees for Bands 1, 2 and 4 - 0.02 degrees for all other bands. These are approximately equivalent to 0.5, 1 and 2km.

Files can either be zipped with bz2 compression (like the HSD format data), or can be uncompressed flat binary.

Initialize the reader.

_calibrate(*data*)

Load calibration from LUT and apply.

static _download_luts(*file_name*)

Download LUTs from remote server.

_get_luts()

Download the LUTs needed for count->Refl/BT conversion.

_load_lut()

Determine if LUT is available and, if not, download it.

_read_data(*fp_*)

Read raw binary data from file.

static _untar_luts(*tarred_file*, *outdir*)

Uncompress downloaded LUTs, which are a tarball.

calibrate(*data*, *calib*)

Calibrate the data.

get_area_def(*dsid*)

Get the area definition.

This is fixed, but not defined in the file. So we must generate it ourselves with some assumptions.

get_dataset(*key*, *info*)

Get the dataset.

read_band(*key*, *info*)

Read the data.

satpy.readers.ahi_l2_nc module

Reader for Himawari L2 cloud products from NOAA's big data programme.

For more information about the data, see: <https://registry.opendata.aws/noaa-himawari/>.

These products are generated by the NOAA enterprise cloud suite and have filenames like: AHI-CMSK_v1r1_h09_s202308240540213_e202308240549407_c202308240557548.nc

The second letter grouping (CMSK above) indicates the product type:

CMSK - Cloud mask

CHGT - Cloud height

CPHS - Cloud type and phase

These products are generated from the AHI sensor on Himawari-8 and Himawari-9, and are produced at the native instrument resolution for the IR channels (2km at nadir).

NOTE: This reader is currently only compatible with full disk scenes. Unlike level 1 himawari data, the netCDF files do not contain the required metadata to produce an appropriate area definition for the data contents, and hence the area definition is hardcoded into the reader.

A warning is displayed to the user highlighting this. The assumed area definition is a full disk image at the nominal subsatellite longitude of 140.7 degrees East.

All the simple data products are supported here, but multidimensional products are not yet supported. These include the CldHgtFlag and the CloudMaskPacked variables.

class satpy.readers.ahi_l2_nc.**HIML2NCFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for Himawari L2 NOAA enterprise data in netCDF format.

Initialize the reader.

_get_area_def()

property area

Get AreaDefinition representing this file's data.

property end_time

End timestamp of the dataset.

get_area_def(*dsid*)

Get the area definition.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Start timestamp of the dataset.

satpy.readers.ami_l1b module

Advanced Meteorological Imager reader for the Level 1b NetCDF4 format.

class satpy.readers.ami_l1b.**AMIL1bNetCDF**(*filename*, *filename_info*, *filetype_info*, *calib_mode='PYSPECTRAL'*, *allow_conditional_pixels=False*, *user_calibration=None*)

Bases: *BaseFileHandler*

Base reader for AMI L1B NetCDF4 files.

AMI data contains GSICS adjustment factors for the IR bands. By default, these are not applied. If you wish to apply them then you must set the calibration mode appropriately:

```
import satpy
import glob

filenames = glob.glob('*FLDK*.dat')
scene = satpy.Scene(filenames,
                    reader='ahi_hsd',
                    reader_kwargs={'calib_mode': 'gsics'})
scene.load(['B13'])
```

In addition, the GSICS website (and other sources) also supply radiance correction coefficients like so:

```
radiance_corr = (radiance_orig - corr_offset) / corr_slope
```

If you wish to supply such coefficients, pass 'user_calibration' and a dictionary containing per-channel slopes and offsets as a reader_kwarg:

```
user_calibration={'chan': {'slope': slope, 'offset': offset}}
```

If you do not have coefficients for a particular band, then by default the slope will be set to 1 .and the offset to 0.:

```
import satpy
import glob

# Load bands 7, 14 and 15, but we only have coefs for 7+14
calib_dict = {'WV063': {'slope': 0.99, 'offset': 0.002},
              'IR087': {'slope': 1.02, 'offset': -0.18}}

filenames = glob.glob('*.nc')
scene = satpy.Scene(filenames,
                    reader='ami_l1b',
                    reader_kwargs={'user_calibration': calib_dict,
                                   'calib_mode': 'file')
# IR133 will not have radiance correction applied.
scene.load(['WV063', 'IR087', 'IR133'])
```

By default these updated coefficients are not used. In most cases, setting *calib_mode* to *file* is required in order to use external coefficients.

Open the NetCDF file with xarray and prepare the Dataset for reading.

_apply_gsics_rad_correction(*data*)

Retrieve GSICS factors from L1 file and apply to radiance.

_apply_user_rad_correction(*data*)

Retrieve user-supplied radiance correction and apply.

_calibrate_ir(*dataset_id*, *data*)

Calibrate radiance data to BTs using either pyspectral or in-file coefficients.

property end_time

Get observation end time.

get_area_def(*dsid*)

Get area definition for this file.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset as a xarray DataArray.

get_orbital_parameters()

Collect orbital parameters for this file.

property start_time

Get observation start time.

satpy.readers.amsr2_l1b module

Reader for AMSR2 L1B files in HDF5 format.

class satpy.readers.amsr2_l1b.**AMSR2L1BFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

File handler for AMSR2 l1b.

Initialize file handler.

get_dataset(*ds_id*, *ds_info*)

Get output data and metadata of specified dataset.

get_metadata(*ds_id*, *ds_info*)

Get the metadata.

get_shape(*ds_id*, *ds_info*)

Get output shape of specified dataset.

satpy.readers.amsr2_l2 module

Reader for AMSR2 L2 files in HDF5 format.

class satpy.readers.amsr2_l2.**AMSR2L2FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *AMSR2L1BFileHandler*

AMSR2 level 2 file handler.

Initialize file handler.

get_dataset(*ds_id*, *ds_info*)

Get output data and metadata of specified dataset.

mask_dataset(*ds_info*, *data*)

Mask data with the fill value.

scale_dataset(*var_path*, *data*)

Scale data with the scale factor attribute.

satpy.readers.amsr2_l2_gaasp module

GCOM-W1 AMSR2 Level 2 files from the GAASP software.

GAASP output files are in the NetCDF4 format. Software is provided by NOAA and is also distributed by the CSPP group. More information on the products supported by this reader can be found here: https://www.star.nesdis.noaa.gov/jpss/gcom.php for more information.

GAASP includes both swath/granule products and gridded products. Swath products are provided in files with "MBT", "OCEAN", "SNOW", or "SOIL" in the filename. Gridded products are in files with "SEAICE-SH" or "SEAICE-NH" in the filename where SH stands for South Hemisphere and NH stands for North Hemisphere. These gridded products are on the EASE2 North pole and South pole grids. See https://nsidc.org/ease/ease-grid-projection-gt for more details.

Note that since SEAICE products can be on both the northern or southern hemisphere or both depending on what files are provided to Satpy, this reader appends a *_NH* and *_SH* suffix to all variable names that are dynamically discovered from the provided files.

class satpy.readers.amsr2_l2_gaasp.**GAASPFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Generic file handler for GAASP output files.

Initialize file handler.

_add_lonlat_coords(*data_arr*, *ds_info*)

_available_if_this_file_type(*configured_datasets*)

_available_new_datasets()

_fill_data(*data_arr*, *attrs*)

_get_ds_info_for_data_arr(*var_name*, *data_arr*)

_get_var_name_without_suffix(*var_name*)

_is_2d_yx_data_array(*data_arr*)

static _nan_for_dtype(*data_arr_dtype*)

_scale_data(*data_arr*, *attrs*)

available_datasets(*configured_datasets=None*)

Dynamically discover what variables can be loaded from this file.

See satpy.readers.file_handlers.BaseHandler.available_datasets() for more information.

dim_resolutions = {'Number_of_hi_rez_FOVs': 5000, 'Number_of_low_rez_FOVs': 10000}

property end_time

Get end time of observation.

get_dataset(*dataid*, *ds_info*)

Load, scale, and collect metadata for the specified DataID.

is_gridded = False

property nc

Get the xarray dataset for this file.

property platform_name

Name of the platform whose data is stored in this file.

property sensor_names

Sensors who have data in this file.

property start_time

Get start time of observation.

time_dims = ('Time_Dimension',)

x_dims: Tuple[str, ...] = ('Number_of_hi_rez_FOVs', 'Number_of_low_rez_FOVs')

y_dims: Tuple[str, ...] = ('Number_of_Scans',)

class satpy.readers.amsr2_l2_gaasp.**GAASPGriddedFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *GAASPFileHandler*

GAASP file handler for gridded products like SEAICE.

Initialize file handler.

static _get_extents(*data_shape*, *res*)

dim_resolutions = {'Number_of_X_Dimension': 10000}

get_area_def(*dataid*)

Create area definition for equirectangular projected data.

is_gridded = True

x_dims: Tuple[str, ...] = ('Number_of_X_Dimension',)

y_dims: Tuple[str, ...] = ('Number_of_Y_Dimension',)

class satpy.readers.amsr2_l2_gaasp.**GAASPLowResFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *GAASPFileHandler*

GAASP file handler for files that only have low resolution products.

Initialize file handler.

dim_resolutions = {'Number_of_low_rez_FOVs': 10000}

x_dims: Tuple[str, ...] = ('Number_of_low_rez_FOVs',)

satpy.readers.ascat_l2_soilmoisture_bufr module

ASCAT Soil moisture product reader for BUFR messages.

Based on the IASI L2 SO2 BUFR reader.

class satpy.readers.ascat_l2_soilmoisture_bufr.**AscatSoilMoistureBufr**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *BaseFileHandler*

File handler for the ASCAT Soil Moisture BUFR product.

Initialise the file handler for the ASCAT Soil Moisture BUFR data.

property end_time

Return the end time of data acquisition.

extract_msg_date_extremes(*bufr*, *date_min=None*, *date_max=None*)

Extract the minimum and maximum dates from a single bufr message.

get_bufr_data(*key*)

Get BUFR data by key.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using the BUFR key in dataset_info.

get_start_end_date()

Get the first and last date from the bufr file.

property platform_name

Return spacecraft name.

property start_time

Return the start time of data acqusition.

satpy.readers.atms_l1b_nc module

Advanced Technology Microwave Sounder (ATMS) Level 1B product reader.

The format is explained in the ATMS L1B Product User Guide

class satpy.readers.atms_l1b_nc.**AtmsL1bNCFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *NetCDF4FileHandler*

Reader class for ATMS L1B products in netCDF format.

Initialize file handler.

static _drop_coords(*dataset*)

Drop coords that are not in dims.

_merge_attributes(*dataset*, *dataset_info*)

Merge attributes of the dataset.

_select_dataset(*name*)

Select dataset.

static _standardize_dims(*dataset*)

Standardize dims to y, x.

property antenna_temperature

Get antenna temperature.

property attrs

Return attributes.

property end_time

Get observation end time.

get_dataset(*dataset_id*, *ds_info*)

Get dataset.

property platform_name

Get platform name.

property sensor

Get sensor.

property start_time

Get observation start time.

satpy.readers.atms_sdr_hdf5 module

Reader for the ATMS SDR format.

A reader for Advanced Technology Microwave Sounder (ATMS) SDR data as it e.g. comes out of the CSPP package for processing Direct Readout data.

The format is described in the JPSS COMMON DATA FORMAT CONTROL BOOK (CDFCB):

Joint Polar Satellite System (JPSS) Common Data Format Control Book - External (CDFCB-X) Volume III - SDR/TDR Formats

(474-00001-03_JPSS-CDFCB-X-Vol-III_0124C.pdf)

https://www.nesdis.noaa.gov/about/documents-reports/jpss-technical-documents/jpss-science-documents

class satpy.readers.atms_sdr_hdf5.**ATMS_SDR_FileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *JPSS_SDR_FileHandler*

ATMS SDR HDF5 File Reader.

Initialize file handler.

_get_atms_channel_index(*ch_name*)

Get the channels array index from name.

_get_scans_per_granule(*dataset_group*)

_get_variable(*var_path*, *channel_index=None*)

get_dataset(*dataset_id*, *ds_info*)

Get the dataset corresponding to *dataset_id*.

The size of the return DataArray will be dependent on the number of scans actually sensed of course.

satpy.readers.avhrr_l1b_gaclac module

Reading and calibrating GAC and LAC AVHRR data.

Uses Pygac under the hood. See the Pygac Documentation for supported data formats as well as calibration and navigation methods.

class satpy.readers.avhrr_l1b_gaclac.**GACLACFile**(*filename*, *filename_info*, *filetype_info*, *start_line=None*, *end_line=None*, *strip_invalid_coords=True*, *interpolate_coords=True*, ***reader_kwargs*)

Bases: `BaseFileHandler`

Reader for GAC and LAC data.

Init the file handler.

Parameters

- **start_line** – User defined start scanline
- **end_line** – User defined end scanline
- **strip_invalid_coords** – Strip scanlines with invalid coordinates in the beginning/end of the orbit
- **interpolate_coords** – Interpolate coordinates from every eighth pixel to all pixels.
- **reader_kwargs** – More keyword arguments to be passed to pygac.Reader. See the pygac documentation for available options.

_get_angle(*key*)

Get angles and buffer results.

_get_channel(*key*)

Get channel and buffer results.

_get_qual_flags()

Get quality flags and buffer results.

_slice(*data*)

Select user-defined scanlines and/or strip invalid coordinates.

Returns
Sliced data

_strip_invalid_lat()

Strip scanlines with invalid coordinates in the beginning/end of the orbit.

Returns
First and last scanline with valid latitudes.

_update_attrs(*res*)

Update dataset attributes.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Get the dataset.

read_raw_data()

Create a pygac reader and read raw data from the file.

slice(*data*, *times*)

Select user-defined scanlines and/or strip invalid coordinates.

Furthermore, update scanline timestamps.

Parameters

- **data** – Data to be sliced
- **times** – Scanline timestamps

Returns

Sliced data and timestamps

property start_time

Get the start time.

satpy.readers.clavrx module

Interface to CLAVR-X HDF4 products.

class satpy.readers.clavrx.**CLAVRXHDF4FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF4FileHandler*, *_CLAVRxHelper*

A file handler for CLAVRx files.

Init method.

_is_polar()

available_datasets(*configured_datasets=None*)

Automatically determine datasets provided by this file.

property end_time

Get the end time.

get_area_def(*key*)

Get the area definition of the data at hand.

get_dataset(*dataset_id*, *ds_info*)

Get a dataset.

get_nadir_resolution(*sensor*)

Get nadir resolution.

get_shape(*dataset_id*, *ds_info*)

Get the shape.

property start_time

Get the start time.

class satpy.readers.clavrx.**CLAVRXNetCDFFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *_CLAVRxHelper*, *BaseFileHandler*

File Handler for CLAVRX netcdf files.

Init method.

_available_new_datasets(*handled_vars*)

Metadata for available variables other than BT.

_get_ds_info_for_data_arr(*var_name*)

_is_2d_yx_data_array(*data_arr*)

_is_polar()

available_datasets(*configured_datasets=None*)

Dynamically discover what variables can be loaded from this file.

See `satpy.readers.file_handlers.BaseHandler.available_datasets()` for more information.

get_area_def(*key*)

Get the area definition of the data at hand.

get_dataset(*dataset_id*, *ds_info*)

Get a dataset.

class satpy.readers.clavrx.**_CLAVRxHelper**

Bases: object

A base class for the CLAVRx File Handlers.

static _area_extent(*x*, *y*, *h: float*)

static _find_input_nc(*filename: str*, *l1b_base: str*) → str

static _get_data(*data*, *dataset_id: dict*) → DataArray

Get a dataset.

static _read_axi_fixed_grid(*filename: str*, *l1b_attr*) → AreaDefinition

Read a fixed grid.

CLAVR-x does not transcribe fixed grid parameters to its output We have to recover that information from the original input file, which is partially named as L1B attribute

example attributes found in L2 CLAVR-x files: sensor = "AHI" ; platform = "HIM8" ; FILENAME = "clavrx_H08_20180719_1300.level2.hdf" ; L1B = "clavrx_H08_20180719_1300" ;

static _read_pug_fixed_grid(*projection_coordinates: netCDF4.Variable*, *distance_multiplier=1.0*) → dict

Read from recent PUG format, where axes are in meters.

static _remove_attributes(*attrs: dict*) → dict

Remove attributes that described data before scaling.

static _scale_data(*data_arr: DataArray | int*, *scale_factor: float*, *add_offset: float*) → DataArray

Scale data, if needed.

static get_metadata(*sensor: str*, *platform: str*, *attrs: dict*, *ds_info: dict*) → dict

Get metadata.

satpy.readers.clavrx.**_get_platform**(*platform: str*) → str

Get the platform.

satpy.readers.clavrx.**_get_rows_per_scan**(*sensor: str*) → int | None

Get number of rows per scan.

satpy.readers.clavrx.**_get_sensor**(*sensor: str*) → str

Get the sensor.

satpy.readers.cmsaf_claas2 module

Module containing CMSAF CLAAS v2 FileHandler.

class satpy.readers.cmsaf_claas2.**CLAAS2**(**args*, ***kwargs*)

Bases: *NetCDF4FileHandler*

Handle CMSAF CLAAS-2 files.

Initialise class.

_get_dsinfo(*var*)

Get metadata for variable.

Return metadata dictionary for variable `var`.

_get_full_disk()

_get_subset_of_full_disk()

Get subset of the full disk.

CLAAS products are provided on a grid that is slightly smaller than the full disk (excludes most of the space pixels).

available_datasets(*configured_datasets=None*)

Yield a collection of available datasets.

Return a generator that will yield the datasets available in the loaded files. See docstring in parent class for specification details.

property end_time

Get end time from file.

get_area_def(*dataset_id*)

Get the area definition.

get_dataset(*dataset_id*, *info*)

Get the dataset.

grid_size = 3636

property start_time

Get start time from file.

satpy.readers.cmsaf_claas2.**_adjust_area_to_match_shifted_data**(*area*)

satpy.readers.cmsaf_claas2.**_is_georef_offset_present**(*date*)

satpy.readers.electrol_hrit module

HRIT format reader.

References

ELECTRO-L GROUND SEGMENT MSU-GS INSTRUMENT,
LRIT/HRIT Mission Specific Implementation, February 2012

class satpy.readers.electrol_hrit.**HRITGOMSEpilogueFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HRITFileHandler*

GOMS HRIT format reader.

Initialize the reader.

read_epilogue()

Read the prologue metadata.

class satpy.readers.electrol_hrit.**HRITGOMSFileHandler**(*filename*, *filename_info*, *filetype_info*, *prologue*, *epilogue*)

Bases: *HRITFileHandler*

GOMS HRIT format reader.

Initialize the reader.

_calibrate(*data*)

Visible/IR channel calibration.

static _getitem(*block*, *lut*)

calibrate(*data*, *calibration*)

Calibrate the data.

get_area_def(*dsid*)

Get the area definition of the band.

get_dataset(*key*, *info*)

Get the data from the files.

class satpy.readers.electrol_hrit.**HRITGOMSPrologueFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HRITFileHandler*

GOMS HRIT format reader.

Initialize the reader.

process_prologue()

Reprocess prologue to correct types.

read_prologue()

Read the prologue metadata.

satpy.readers.electrol_hrit.**recarray2dict**(*arr*)

Change record array to a dictionary.

satpy.readers.epic_l1b_h5 module

File handler for DSCOVR EPIC L1B data in hdf5 format.

The `epic_l1b_h5` reader reads and calibrates EPIC L1B image data in hdf5 format.

This reader supports all image and most ancillary datasets. Once the reader is initialised:

`` scn = Scene([epic_filename], reader='epic_l1b_h5')``

Channels can be loaded with the 'B' prefix and their wavelength in nanometers:

```
scn.load(['B317', 'B688'])
```

while ancillary data can be loaded by its name:

```
scn.load(['solar_zenith_angle'])
```

Note that ancillary dataset names use common standards and not the dataset names in the file. By default, channel data is loaded as calibrated reflectances, but counts data is also available.

class satpy.readers.epic_l1b_h5.**DscovrEpicL1BH5FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

File handler for DSCOVR EPIC L1b data.

Init filehandler.

static _mask_infinite(*band*)

_update_metadata(*band*)

static calibrate(*data*, *ds_name*, *calibration=None*)

Convert counts input reflectance.

property end_time

Get the end time.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset.

property start_time

Get the start time.

satpy.readers.eps_l1b module

Reader for eps level 1b data. Uses xml files as a format description.

class satpy.readers.eps_l1b.**EPSAVHRRFile**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Eps level 1b reader for AVHRR data.

Initialize FileHandler.

_get_angle_dataarray(*key*)

Get an angle dataarray.

_get_calibrated_dataarray(*key*)

Get a calibrated dataarray.

_get_data_array(*key*)

_get_full_angles(*solar_zenith*, *sat_zenith*, *solar_azimuth*, *sat_azimuth*)

_get_full_angles_uncached()

Get the interpolated angles.

_get_full_lonlats_uncached()

Get the interpolated longitudes and latitudes.

_interpolate(*lons_like*, *lats_like*)

_interpolate_20km_to_1km

_read_all()

property end_time

Get end time.

get_bounding_box()

Get bounding box.

get_dataset(*key*, *info*)

Get calibrated channel data.

get_lonlats()

Get lonlats.

keys()

List of reader's keys.

property platform_name

Get platform name.

property sensor_name

Get sensor name.

sensors = {'AVHR': 'avhrr-3'}

spacecrafts = {'M01': 'Metop-B', 'M02': 'Metop-A', 'M03': 'Metop-C'}

property start_time

Get start time.

property three_a_mask

Mask for 3A.

property three_b_mask

Mask for 3B.

units = {'brightness_temperature': 'K', 'reflectance': '%'}

satpy.readers.eps_l1b.**create_xarray**(*arr*)

Create xarray with correct dimensions.

satpy.readers.eps_l1b.**radiance_to_bt**(*arr*, *wc_*, *a__*, *b__*)

Convert to BT in K.

satpy.readers.eps_l1b.**radiance_to_refl**(*arr*, *solar_flux*)

Convert to reflectances in %.

satpy.readers.eps_l1b.**read_records**(*filename*)

Read *filename* without scaling it afterwards.

satpy.readers.eum_base module

Utilities for EUMETSAT satellite data.

satpy.readers.eum_base.**get_service_mode**(*instrument_name*, *ssp_lon*)

Get information about service mode for a given instrument and subsatellite longitude.

satpy.readers.eum_base.**recarray2dict**(*arr*)

Convert numpy record array to a dictionary.

satpy.readers.eum_base.**timecds2datetime**(*tcds*)

Convert time_cds-variables to datetime-object.

Works both with a dictionary and a numpy record_array.

satpy.readers.fci_l1c_nc module

Interface to MTG-FCI L1c NetCDF files.

This module defines the `FCIL1cNCFileHandler` file handler, to be used for reading Meteosat Third Generation (MTG) Flexible Combined Imager (FCI) Level-1c data. FCI will fly on the MTG Imager (MTG-I) series of satellites, with the first satellite (MTG-I1) scheduled to be launched on the 13th of December 2022. For more information about FCI, see EUMETSAT.

For simulated test data to be used with this reader, see test data releases. For the Product User Guide (PUG) of the FCI L1c data, see PUG.

Note: This reader currently supports Full Disk High Spectral Resolution Imagery (FDHSI) and High Spatial Resolution Fast Imagery (HRFI) data in full-disc ("FD") scanning mode. If the user provides a list of both FDHSI and HRFI files from the same repeat cycle to the Satpy `Scene`, Satpy will automatically read the channels from the source with the finest resolution, i.e. from the HRFI files for the vis_06, nir_22, ir_38, and ir_105 channels. If needed, the desired resolution can be explicitly requested using e.g.: `scn.load(['vis_06'], resolution=1000)`.

Note that RSS data is not supported yet.

Geolocation is based on information from the data files. It uses:

- From the shape of the data variable `data/<channel>/measured/effective_radiance`, start and end line columns of current swath.
- From the data variable `data/<channel>/measured/x`, the x-coordinates for the grid, in radians (azimuth angle positive towards West).
- From the data variable `data/<channel>/measured/y`, the y-coordinates for the grid, in radians (elevation angle positive towards North).
- From the attribute `semi_major_axis` on the data variable `data/mtg_geos_projection`, the Earth equatorial radius
- From the attribute `inverse_flattening` on the same data variable, the (inverse) flattening of the ellipsoid

- From the attribute `perspective_point_height` on the same data variable, the geostationary altitude in the normalised geostationary projection
- From the attribute `longitude_of_projection_origin` on the same data variable, the longitude of the projection origin
- From the attribute `sweep_angle_axis` on the same, the sweep angle axis, see https://proj.org/operations/projections/geos.html

From the pixel centre angles in radians and the geostationary altitude, the extremities of the lower left and upper right corners are calculated in units of arc length in m. This extent along with the number of columns and rows, the sweep angle axis, and a dictionary with equatorial radius, polar radius, geostationary altitude, and longitude of projection origin, are passed on to `pyresample.geometry.AreaDefinition`, which then uses proj4 for the actual geolocation calculations.

The reading routine supports channel data in counts, radiances, and (depending on channel) brightness temperatures or reflectances. The brightness temperature and reflectance calculation is based on the formulas indicated in PUG. Radiance datasets are returned in units of radiance per unit wavenumber (mW m-2 sr-1 (cm-1)-1). Radiances can be converted to units of radiance per unit wavelength (W m-2 um-1 sr-1) by multiplying with the *radiance_unit_conversion_coefficient* dataset attribute.

For each channel, it also supports a number of auxiliary datasets, such as the pixel quality, the index map and the related geometric and acquisition parameters: time, subsatellite latitude, subsatellite longitude, platform altitude, subsolar latitude, subsolar longitude, earth-sun distance, sun-satellite distance, swath number, and swath direction.

All auxiliary data can be obtained by prepending the channel name such as `"vis_04_pixel_quality"`.

Warning: The API for the direct reading of pixel quality is temporary and likely to change. Currently, for each channel, the pixel quality is available by `<chan>_pixel_quality`. In the future, they will likely all be called `pixel_quality` and disambiguated by a to-be-decided property in the *DataID*.

Note: For reading compressed data, a decompression library is needed. Either install the FCIDECOMP library (see PUG), or the `hdf5plugin` package with:

```
pip install hdf5plugin
```

or:

```
conda install hdf5plugin -c conda-forge
```

If you use `hdf5plugin`, make sure to add the line `import hdf5plugin` at the top of your script.

class satpy.readers.fci_l1c_nc.**FCIL1cNCFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *NetCDF4FsspecFileHandler*

Class implementing the MTG FCI L1c Filehandler.

This class implements the Meteosat Third Generation (MTG) Flexible Combined Imager (FCI) Level-1c NetCDF reader. It is designed to be used through the `Scene` class using the `load` method with the reader `"fci_l1c_nc"`.

Initialize file handler.

_get_aux_data_lut_vector(*aux_data_name*)

Load the lut vector of an auxiliary variable.

_get_dataset_aux_data(*dsname*)

Get the auxiliary data arrays using the index map.

_get_dataset_index_map(*dsname*)

Load the index map for an FCI channel.

_get_dataset_measurand(*key*, *info=None*)

Load dataset corresponding to channel measurement.

Load a dataset when the key refers to a measurand, whether uncalibrated (counts) or calibrated in terms of brightness temperature, radiance, or reflectance.

_get_dataset_quality(*dsname*)

Load a quality field for an FCI channel.

static _getitem(*block*, *lut*)

_platform_name_translate = {'MTI1': 'MTG-I1', 'MTI2': 'MTG-I2', 'MTI3': 'MTG-I3', 'MTI4': 'MTG-I4'}

calc_area_extent(*key*)

Calculate area extent for a dataset.

calibrate(*data*, *key*)

Calibrate data.

calibrate_counts_to_physical_quantity(*data*, *key*)

Calibrate counts to radiances, brightness temperatures, or reflectances.

calibrate_counts_to_rad(*data*, *key*)

Calibrate counts to radiances.

calibrate_rad_to_bt(*radiance*, *key*)

IR channel calibration.

calibrate_rad_to_refl(*radiance*, *key*)

VIS channel calibration.

property end_time

Get end time.

get_area_def(*key*)

Calculate on-fly area definition for a dataset in geos-projection.

get_channel_measured_group_path(*channel*)

Get the channel's measured group path.

get_dataset(*key*, *info=None*)

Load a dataset.

get_segment_position_info()

Get information about the size and the position of the segment inside the final image array.

As the final array is composed by stacking segments vertically, the position of a segment inside the array is defined by the numbers of the start (lowest) and end (highest) row of the segment. The row numbering is assumed to start with 1. This info is used in the GEOVariableSegmentYAMLReader to compute optimal segment sizes for missing segments.

Note: in the FCI terminology, a segment is actually called "chunk". To avoid confusion with the dask concept of chunk, and to be consistent with SEVIRI, we opt to use the word segment.

property nominal_end_time

Get nominal end time.

property nominal_start_time

Get nominal start time.

property observation_end_time

Get observation end time.

property observation_start_time

Get observation start time.

property orbital_param

Compute the orbital parameters for the current segment.

property rc_period_min

Get nominal repeat cycle duration.

As RSS is not yet implemeted and error will be raised if RSS are to be read

property start_time

Get start time.

satpy.readers.fci_l1c_nc.**_ensure_dataarray**(*arr*)

satpy.readers.fci_l1c_nc.**_get_aux_data_name_from_dsname**(*dsname*)

satpy.readers.fci_l1c_nc.**_get_channel_name_from_dsname**(*dsname*)

satpy.readers.fci_l2_nc module

Reader for the FCI L2 products in NetCDF4 format.

class satpy.readers.fci_l2_nc.**FciL2CommonFunctions**

Bases: object

Shared operations for file handlers.

_get_global_attributes()

Create a dictionary of global attributes to be added to all datasets.

Returns

A dictionary of global attributes.
filename: name of the product file spacecraft_name: name of the spacecraft ssp_lon: longitude of subsatellite point sensor: name of sensor platform_name: name of the platform

Return type
dict

static _mask_data(*variable*, *fill_value*)

Set fill_values, as defined in yaml-file, to NaN.

Set data points in variable to NaN if they are equal to fill_value or any of the values in fill_value if fill_value is a list.

_set_attributes(*variable*, *dataset_info*, *segmented=False*)

Set dataset attributes.

_slice_dataset(*variable*, *dataset_info*, *dimensions*)

Slice data if dimension layers have been provided in yaml-file.

property sensor_name

Return instrument name.

property spacecraft_name

Return spacecraft name.

property ssp_lon

Return longitude at subsatellite point.

class satpy.readers.fci_l2_nc.**FciL2NCFileHandler**(*filename*, *filename_info*, *filetype_info*, *with_area_definition=True*)

Bases: *FciL2CommonFunctions*, *BaseFileHandler*

Reader class for FCI L2 products in NetCDF4 format.

Open the NetCDF file with xarray and prepare for dataset reading.

_compute_area_def(*dataset_id*)

Compute the area definition.

Returns
A pyresample AreaDefinition object containing the area definition.

Return type
AreaDefinition

static _decode_clm_test_data(*variable*, *dataset_info*)

_get_area_extent()

Calculate area extent of dataset.

_get_proj_area(*dataset_id*)

Extract projection and area information.

get_area_def(*key*)

Return the area definition.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using the file_key in dataset_info.

static get_total_cot(*variable*)

Sum the cloud optical thickness from the two OCA layers.

The optical thickness has to be transformed to linear space before adding the values from the two layers. The combined/total optical thickness is then transformed back to logarithmic space.

class satpy.readers.fci_l2_nc.**FciL2NCSegmentFileHandler**(*filename*, *filename_info*, *filetype_info*, *with_area_definition=False*)

Bases: *FciL2CommonFunctions*, *BaseFileHandler*

Reader class for FCI L2 Segmented products in NetCDF4 format.

Open the NetCDF file with xarray and prepare for dataset reading.

_construct_area_def(*dataset_id*)

Construct the area definition.

Returns
A pyresample AreaDefinition object containing the area definition.

Return type
AreaDefinition

static _modify_area_extent(*stand_area_extent*)

Modify area extent to macth satellite projection.

Area extent has to be modified since the L2 products are stored with the south-east in the upper-right corner (as opposed to north-east in the standardized area definitions).

get_area_def(*key*)

Return the area definition.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using the file_key in dataset_info.

satpy.readers.file_handlers module

Interface for BaseFileHandlers.

class satpy.readers.file_handlers.**BaseFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: `object`

Base file handler.

Initialize file handler.

static _combine(*infos*, *func*, **keys*)

_combine_orbital_parameters(*all_infos*)

_combine_time_parameters(*all_infos*)

available_datasets(*configured_datasets=None*)

Get information of available datasets in this file.

This is used for dynamically specifying what datasets are available from a file in addition to what's configured in a YAML configuration file. Note that this method is called for each file handler for each file type; care should be taken when possible to reduce the amount of redundant datasets produced.

This method should **not** update values of the dataset information dictionary **unless** this file handler has a matching file type (the data could be loaded from this object in the future) and at least **one** `satpy.dataset.DataID` key is also modified. Otherwise, this file type may override the information provided by a more preferred file type (as specified in the YAML file). It is recommended that any non-ID metadata be updated during the `BaseFileHandler.get_dataset()` part of loading. This method is not guaranteed that it will be called before any other file type's handler. The availability "boolean" not being `None` does not mean that a file handler called later can't provide an additional dataset, but it must provide more identifying (DataID) information to do so and should yield its new dataset in addition to the previous one.

Parameters
configured_datasets (*list*) – Series of (bool or None, dict) in the same way as is returned by this method (see below). The bool is whether the dataset is available from at least one of the current file handlers. It can also be `None` if no file handler before us knows how to handle it. The dictionary is existing dataset metadata. The dictionaries are typically provided from a YAML configuration file and may be modified, updated, or used as a "template" for additional available datasets. This argument could be the result of a previous file handler's implementation of this method.

Returns

Iterator of (bool or None, dict) pairs where dict is the dataset's metadata. If the dataset is available in the current file type then the boolean value should be `True`, `False` if we **know** about the dataset but it is unavailable, or `None` if this file object is not responsible for it.

Example 1 - Supplement existing configured information:

```
def available_datasets(self, configured_datasets=None):
    "Add information to configured datasets."
    # we know the actual resolution
    res = self.resolution

    # update previously configured datasets
    for is_avail, ds_info in (configured_datasets or []):
        # some other file handler knows how to load this
        # don't override what they've done
        if is_avail is not None:
            yield is_avail, ds_info

        matches = self.file_type_matches(ds_info['file_type'])
        if matches and ds_info.get('resolution') != res:
            # we are meant to handle this dataset (file type matches)
            # and the information we can provide isn't available yet
            new_info = ds_info.copy()
            new_info['resolution'] = res
            yield True, new_info
        elif is_avail is None:
            # we don't know what to do with this
            # see if another future file handler does
            yield is_avail, ds_info
```

Example 2 - Add dynamic datasets from the file:

```
def available_datasets(self, configured_datasets=None):
    "Add information to configured datasets."
    # pass along existing datasets
    for is_avail, ds_info in (configured_datasets or []):
        yield is_avail, ds_info

    # get dynamic variables known to this file (that we created)
    for var_name, val in self.dynamic_variables.items():
        ds_info = {
            'file_type': self.filetype_info['file_type'],
            'resolution': 1000,
            'name': var_name,
        }
        yield True, ds_info
```

combine_info(*all_infos*)

Combine metadata for multiple datasets.

When loading data from multiple files it can be non-trivial to combine things like start_time, end_time, start_orbit, end_orbit, etc.

By default this method will produce a dictionary containing all values that were equal across **all** provided info dictionaries.

Additionally it performs the logical comparisons to produce the following if they exist:

- start_time
- end_time
- start_orbit
- end_orbit
- orbital_parameters
- time_parameters

Also, concatenate the areas.

property end_time

Get end time.

file_type_matches(*ds_ftype*)

Match file handler's type to this dataset's file type.

Parameters

ds_ftype (*str or list*) – File type or list of file types that a dataset is configured to be loaded from.

Returns

`True` if this file handler object's type matches the dataset's file type(s), `None` otherwise. `None` is returned instead of `False` to follow the convention of the *available_datasets()* method.

get_area_def(*dsid*)

Get area definition.

get_bounding_box()

Get the bounding box of the files, as a (lons, lats) tuple.

The tuple return should a lons and lats list of coordinates traveling clockwise around the points available in the file.

get_dataset(*dataset_id*, *ds_info*)

Get dataset.

property sensor_names

List of sensors represented in this file.

property start_time

Get start time.

satpy.readers.file_handlers.**open_dataset**(*filename*, **args*, ***kwargs*)

Open a file with xarray.

Parameters

filename (*Union[str, FSFile]*) – The path to the file to open. Can be a *string* or *FSFile* object which allows using *fsspec* or *s3fs* like files.

Return type

xarray.Dataset

Notes

This can be used to enable readers to open remote files.

satpy.readers.fy4_base module

Base reader for the L1 HDF data from the AGRI and GHI instruments aboard the FengYun-4A/B satellites.

The files read by this reader are described in the official Real Time Data Service:

http://fy4.nsmc.org.cn/data/en/data/realtime.html

class satpy.readers.fy4_base.**FY4Base**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

The base class for the FengYun4 AGRI and GHI readers.

Init filehandler.

static _getitem(*block*, *lut*)

apply_lut(*data*, *lut*)

Calibrate digital number (DN) by applying a LUT.

Parameters

- **data** – Raw detector digital number
- **lut** – the look up table

Returns
Calibrated quantity

calibrate(*data*, *ds_info*, *ds_name*, *file_key*)

Calibrate the data.

calibrate_to_bt(*data*, *ds_info*, *ds_name*)

Calibrate to Brightness Temperatures [K].

calibrate_to_reflectance(*data*, *channel_index*, *ds_info*)

Calibrate to reflectance [%].

property end_time

Get the end time.

get_area_def(*key*)

Get the area definition.

property reflectance_coeffs

Retrieve the reflectance calibration coefficients from the HDF file.

static scale(*dn*, *slope*, *offset*)

Convert digital number (DN) to calibrated quantity through scaling.

Parameters

- **dn** – Raw detector digital number
- **slope** – Slope
- **offset** – Offset

Returns
Scaled data

property start_time

Get the start time.

satpy.readers.generic_image module

Reader for generic image (e.g. gif, png, jpg, tif, geotiff, ...).

Returns a dataset without calibration. Includes coordinates if available in the file (eg. geotiff). If nodata values are present (and rasterio is able to read them), it will be preserved as attribute `_FillValue` in the returned dataset. In case that nodata values should be used to mask pixels (that have equal values) with np.nan, it has to be enabled in the reader yaml file (key `nodata_handling` per dataset with value `"nan_mask"`).

class satpy.readers.generic_image.**GenericImageFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Handle reading of generic image files.

Initialize filehandler.

property end_time

Return end time.

get_area_def(*dsid*)

Get area definition of the image.

get_dataset(*key*, *info*)

Get a dataset from the file.

read()

Read the image.

property start_time

Return start time.

satpy.readers.generic_image.**_handle_nodatavals**(*data*, *nodata_handling*)

Mask data with np.nan or only set 'attr_FillValue'.

satpy.readers.generic_image.**_mask_image_data**(*data*, *info*)

Mask image data if necessary.

Masking is done if alpha channel is present or dataset 'nodata_handling' is set to 'nan_mask'. In the latter case even integer data is converted to float32 and masked with np.nan.

satpy.readers.geocat module

Interface to GEOCAT HDF4 or NetCDF4 products.

Note: GEOCAT files do not currently have projection information or precise pixel resolution information. Additionally the longitude and latitude arrays are stored as 16-bit integers which causes loss of precision. For this reason the lon/lats can't be used as a reliable coordinate system to calculate the projection X/Y coordinates.

Until GEOCAT adds projection information and X/Y coordinate arrays, this reader will estimate the geostationary area the best it can. It currently takes a single lon/lat point as reference and uses hardcoded resolution and projection information to calculate the area extents.

class satpy.readers.geocat.**GEOCATFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *NetCDF4FileHandler*

GEOCAT netCDF4 file handler.

Loading data with decode_times=True

By default, this reader will use `xarray_kwargs={"engine": "netcdf4", "decode_times": False}`. to match behavior of xarray when the geocat reader was first written. To use different options use reader_kwargs when loading the Scene:

```
scene = satpy.Scene(filenames,
                    reader='geocat',
                    reader_kwargs={'xarray_kwargs': {'engine': 'netcdf4', 'decode_
↪times': True}})
```

Open and perform initial investigation of NetCDF file.

_calc_area_resolution(*ds_res*)

_first_good_nav(*lon_arr*, *lat_arr*)

_get_extents(*proj*, *res*, *lon_arr*, *lat_arr*)

_get_proj(*platform*, *ref_lon*)

_load_nav(*name*)

available_datasets(*configured_datasets=None*)

Update information for or add datasets provided by this file.

If this file handler can load a dataset then it will supplement the dataset info with the resolution and possibly coordinate datasets needed to load it. Otherwise it will continue passing the dataset information down the chain.

See *satpy.readers.file_handlers.BaseFileHandler.available_datasets()* for details.

property end_time

Get end time.

get_area_def(*dsid*)

Get area definition.

get_dataset(*dataset_id*, *ds_info*)

Get dataset.

get_metadata(*dataset_id*, *ds_info*)

Get metadata.

get_platform(*platform*)

Get platform.

get_sensor(*sensor*)

Get sensor.

get_shape(*dataset_id*, *ds_info*)

Get shape.

property is_geo

Check platform.

platforms: dict[str, str] = {}

property resolution

Get resolution.

resolutions = {'abi': {1: 1002.0086577437705, 2: 2004.017315487541}, 'ahi': {1: 999.9999820317674, 2: 1999.999964063535, 4: 3999.99992812707}}

property sensor_names

Get sensor names.

sensors = {'goes': 'goes_imager', 'goes16': 'abi', 'goesr': 'abi', 'himawari8': 'ahi'}

property start_time

Get start time.

satpy.readers.gerb_l2_hr_h5 module

GERB L2 HR HDF5 reader.

A reader for the Top of Atmosphere outgoing fluxes from the Geostationary Earth Radiation Budget instrument aboard the Meteosat Second Generation satellites.

class satpy.readers.gerb_l2_hr_h5.**GERB_HR_FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

File handler for GERB L2 High Resolution H5 files.

Initialize file handler.

property end_time

Get end time.

get_area_def(*dsid*)

Area definition for the GERB product.

get_dataset(*ds_id*, *ds_info*)

Read a HDF5 file into an xarray DataArray.

property start_time

Get start time.

satpy.readers.gerb_l2_hr_h5.**gerb_get_dataset**(*ds*, *ds_info*)

Load a GERB dataset in memory from a HDF5 file or HDF5FileHandler.

The routine takes into account the quantisation factor and fill values.

satpy.readers.ghi_l1 module

Geostationary High-speed Imager reader for the Level_1 HDF format.

This instrument is aboard the Fengyun-4B satellite. No document is available to describe this format is available, but it's broadly similar to the co-flying AGRI instrument.

class satpy.readers.ghi_l1.**HDF_GHI_L1**(*filename*, *filename_info*, *filetype_info*)

Bases: *FY4Base*

GHI l1 file handler.

Init filehandler.

adjust_attrs(*data*, *ds_info*)

Adjust the attrs of the data.

get_area_def(*key*)

Get the area definition.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset.

satpy.readers.ghrsst_l2 module

Reader for the GHRSST level-2 formatted data.

class satpy.readers.ghrsst_l2.**GHRSSTL2FileHandler**(*filename*, *filename_info*, *filetype_info*, *engine=None*)

Bases: *BaseFileHandler*

File handler for GHRSST L2 netCDF files.

Initialize the file handler for GHRSST L2 netCDF data.

static _is_sst_file(*name*)

Check if file in the tar archive is a valid SST file.

_open_tarfile()

property end_time

Get end time.

get_dataset(*key*, *info*)

Get any available dataset.

property nc

Get the xarray Dataset for the filename.

property sensor

Get the sensor name.

property start_time

Get start time.

satpy.readers.glm_l2 module

Geostationary Lightning Mapper reader for the Level 2 format from glmtools.

More information about *glmtools* and the files it produces can be found on the project's GitHub repository:

https://github.com/deeplycloudy/glmtools

class satpy.readers.glm_l2.**NCGriddedGLML2**(*filename*, *filename_info*, *filetype_info*)

Bases: *NC_ABI_BASE*

File reader for individual GLM L2 NetCDF4 files.

Open the NetCDF file with xarray and prepare the Dataset for reading.

_is_2d_xy_var(*data_arr*)

_is_category_product(*data_arr*)

available_datasets(*configured_datasets=None*)

Discover new datasets and add information from file.

property end_time

End time of the current file's observations.

get_dataset(*key*, *info*)

Load a dataset.

property sensor

Get sensor name for current file handler.

property start_time

Start time of the current file's observations.

satpy.readers.goes_imager_hrit module

GOES HRIT format reader.

References

LRIT/HRIT Mission Specific Implementation, February 2012 GVARRDL98.pdf 05057_SPE_MSG_LRIT_HRI

exception satpy.readers.goes_imager_hrit.**CalibrationError**

Bases: Exception

Dummy error-class.

class satpy.readers.goes_imager_hrit.**HRITGOESFileHandler**(*filename*, *filename_info*, *filetype_info*, *prologue*)

Bases: *HRITFileHandler*

GOES HRIT format reader.

Initialize the reader.

_calibrate(*data*)

Calibrate *data*.

_get_calibration_params()

Get the calibration parameters from the metadata.

_get_proj_dict(*dataset_id*)

calibrate(*data*, *calibration*)

Calibrate the data.

get_area_def(*dataset_id*)

Get the area definition of the band.

get_dataset(*key*, *info*)

Get the data from the files.

class satpy.readers.goes_imager_hrit.**HRITGOESPrologueFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HRITFileHandler*

GOES HRIT format reader.

Initialize the reader.

process_prologue()

Reprocess prologue to correct types.

read_prologue()

Read the prologue metadata.

satpy.readers.goes_imager_hrit.**_epoch_doy_offset_from_sgs_time**(*sgs_time_array: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → timedelta

satpy.readers.goes_imager_hrit.**_epoch_year_from_sgs_time**(*sgs_time_array: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → datetime

satpy.readers.goes_imager_hrit.**make_gvar_float**(*float_val*)

Make gvar float.

satpy.readers.goes_imager_hrit.**make_sgs_time**(*sgs_time_array: _SupportsArray[dtype[Any]] | _NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex | str | bytes]*) → datetime

Make sgs time.

satpy.readers.goes_imager_nc module

Reader for GOES 8-15 imager data in netCDF format.

Supports netCDF files from both NOAA-CLASS and EUMETSAT.

NOAA-CLASS

GOES-Imager netCDF files from NOAA-CLASS contain detector counts alongside latitude and longitude coordinates.

Note: If ordering files via NOAA CLASS, select 16 bits/pixel.

Note: Some essential information are missing in the netCDF files:

1. Subsatellite point
2. Calibration coefficients
3. Detector-scanline assignment, i.e. information about which scanline was recorded by which detector

Items 1. and 2. are not critical because the images are geo-located and NOAA provides static calibration coefficients ([VIS], [IR]). The detector-scanline assignment however cannot be reconstructed properly. This is where an approximation has to be applied (see below).

Oversampling

GOES-Imager oversamples the viewed scene in E-W direction by a factor of 1.75: IR/VIS pixels are 112/28 urad on a side, but the instrument samples every 64/16 urad in E-W direction (see [BOOK-I] and [BOOK-N]). That means pixels are actually overlapping on the ground. This cannot be represented by a pyresample area definition.

For full disk images it is possible to estimate an area definition with uniform sampling where pixels don't overlap. This can be used for resampling and is available via `scene[dataset].attrs["area_def_uni"]`. The pixel size is derived from altitude and N-S sampling angle. The area extent is based on the maximum scanning angles at the earth's limb.

Calibration

Calibration is performed according to [VIS] and [IR], but with an average calibration coefficient applied to all detectors in a certain channel. The reason for and impact of this approximation is described below.

The GOES imager simultaneously records multiple scanlines per sweep using multiple detectors per channel. The VIS channel has 8 detectors, the IR channels have 1-2 detectors (see e.g. Figures 3-5a/b, 3-6a/b and 3-7/a-b in [BOOK-N]). Each detector has its own calibration coefficients, so in order to perform an accurate calibration, the detector-scanline assignment is needed.

In theory it is known which scanline was recorded by which detector (VIS: 5,6,7,8,1,2,3,4; IR: 1,2). However, the plate on which the detectors are mounted flexes due to thermal gradients in the instrument which leads to a N-S shift of +/- 8 visible or +/- 2 IR pixels. This shift is compensated in the GVAR scan formation process, but in a way which is hard to reconstruct properly afterwards. See [GVAR], section 3.2.1. for details.

Since the calibration coefficients of the detectors in a certain channel only differ slightly, a workaround is to calibrate each scanline with the average calibration coefficients. A worst case estimate of the introduced error can be obtained

by calibrating all possible counts with both the minimum and the maximum calibration coefficients and computing the difference. The maximum differences are:

GOES-8		
Channel	Diff	Unit
00_7	0.0	% # Counts are normalized
03_9	0.187	K
06_8	0.0	K # only one detector
10_7	0.106	K
12_0	0.036	K

GOES-9		
Channel	Diff	Unit
00_7	0.0	% # Counts are normalized
03_9	0.0	K # coefs identical
06_8	0.0	K # only one detector
10_7	0.021	K
12_0	0.006	K

GOES-10		
Channel	Diff	Unit
00_7	1.05	%
03_9	0.0	K # coefs identical
06_8	0.0	K # only one detector
10_7	0.013	K
12_0	0.004	K

GOES-11		
Channel	Diff	Unit
00_7	1.25	%
03_9	0.0	K # coefs identical
06_8	0.0	K # only one detector
10_7	0.0	K # coefs identical
12_0	0.065	K

GOES-12		
Channel	Diff	Unit
00_7	0.8	%
03_9	0.0	K # coefs identical
06_5	0.044	K
10_7	0.0	K # coefs identical
13_3	0.0	K # only one detector

GOES-13		
Channel	Diff	Unit
00_7	1.31	%
03_9	0.0	K # coefs identical
06_5	0.085	K
10_7	0.008	K
13_3	0.0	K # only one detector

GOES-14		
Channel	Diff	Unit
00_7	0.66	%
03_9	0.0	K # coefs identical
06_5	0.043	K
10_7	0.006	K
13_3	0.003	K

GOES-15		
Channel	Diff	Unit
00_7	0.86	%
03_9	0.0	K # coefs identical
06_5	0.02	K
10_7	0.009	K
13_3	0.008	K

EUMETSAT

During tandem operations of GOES-15 and GOES-17, EUMETSAT distributed a variant of this dataset with the following differences:

1. The geolocation is in a separate file, used for all bands
2. VIS data is calibrated to Albedo (or reflectance)
3. IR data is calibrated to radiance.
4. VIS data is downsampled to IR resolution (4km)
5. File name differs also slightly
6. Data is received via EumetCast

References:

- [GVAR] GVAR transmission format
- [BOOK-N] GOES-N databook
- [BOOK-I] GOES-I databook (broken)
- [IR] Conversion of GVAR Infrared Data to Scene Radiance or Temperature
- [VIS] Calibration of the Visible Channels of the GOES Imagers and Sounders
- [GLOSSARY] GVAR_IMG Glossary
- [SCHED-W] GOES-15 Routine Imager Schedule
- [SCHED-E] Optimized GOES-East Routine Imager Schedule

class satpy.readers.goes_imager_nc.**AreaDefEstimator**(*platform_name*, *channel*)

Bases: object

Estimate area definition for GOES-Imager.

Create the instance.

_create_area_def(*projection*, *area_extent*, *shape*)

_get_area_description()

_get_area_extent_at_max_scan_angle(*proj_dict*)

_get_max_scan_angle(*proj_dict*)

_get_projection(*projection_longitude*)

_get_shape_with_uniform_pixel_size(*area_extent*)

_get_uniform_pixel_size()

get_area_def_with_uniform_sampling(*projection_longitude*)

Get area definition with uniform sampling.

The area definition is based on geometry and instrument properties: Pixel size is derived from altitude and N-S sampling angle. Area extent is based on the maximum scanning angles at the limb of the earth.

class satpy.readers.goes_imager_nc.**GOESCoefficientReader**(*ir_url*, *vis_url*)

Bases: object

Read GOES Imager calibration coefficients from NOAA reference HTMLs.

Init the coef reader.

_denoise(*string*)

_float(*string*)

Convert string to float.

Take care of numbers in exponential format

_get_ir_coefs(*platform*, *channel*)

_get_table(*root*, *heading*, *heading_type*)

_get_vis_coefs(*platform*)

_load_url_or_file(*url*)

get_coefs(*platform*, *channel*)

Get the coefs.

gvar_channels = {'GOES-10': {'00_7': 1, '03_9': 2, '06_8': 3, '10_7': 4, '12_0': 5}, 'GOES-11': {'00_7': 1, '03_9': 2, '06_8': 3, '10_7': 4, '12_0': 5}, 'GOES-12': {'00_7': 1, '03_9': 2, '06_5': 3, '10_7': 4, '13_3': 6}, 'GOES-13': {'00_7': 1, '03_9': 2, '06_5': 3, '10_7': 4, '13_3': 6}, 'GOES-14': {'00_7': 1, '03_9': 2, '06_5': 3, '10_7': 4, '13_3': 6}, 'GOES-15': {'00_7': 1, '03_9': 2, '06_5': 3, '10_7': 4, '13_3': 6}, 'GOES-8': {'00_7': 1, '03_9': 2, '06_8': 3, '10_7': 4, '12_0': 5}, 'GOES-9': {'00_7': 1, '03_9': 2, '06_8': 3, '10_7': 4, '12_0': 5}}

ir_tables = {'GOES-10': '2-3', 'GOES-11': '2-4', 'GOES-12': '2-5a', 'GOES-13': '2-6', 'GOES-14': '2-7c', 'GOES-15': '2-8b', 'GOES-8': '2-1', 'GOES-9': '2-2'}

vis_tables = {'GOES-10': 'Table 2.', 'GOES-11': 'Table 3.', 'GOES-12': 'Table 4.', 'GOES-13': 'Table 5.', 'GOES-14': 'Table 6.', 'GOES-15': 'Table 7.', 'GOES-8': 'Table 1.', 'GOES-9': 'Table 1.'}

class satpy.readers.goes_imager_nc.**GOESEUMGEONCFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for GOES Geolocation data in EUM netCDF format.

Initialize the reader.

get_dataset(*key*, *info*)

Load dataset designated by the given key from file.

property resolution

Specify the spatial resolution of the dataset.

In the EUMETSAT format VIS data is downsampled to IR resolution (4km).

class satpy.readers.goes_imager_nc.**GOESEUMNCFileHandler**(*filename*, *filename_info*, *filetype_info*, *geo_data*)

Bases: *GOESNCBaseFileHandler*

File handler for GOES Imager data in EUM netCDF format.

TODO: Remove datasets which are not available in the file (counts, VIS radiance) via available_datasets() -> See #434

Initialize the reader.

calibrate(*data*, *calibration*, *channel*)

Perform calibration.

get_dataset(*key*, *info*)

Load dataset designated by the given key from file.

ir_sectors = {(566, 3464): 'Southern Hemisphere (GOES-East)', (1062, 2760): 'Southern Hemisphere (GOES-West)', (1354, 3312): 'Northern Hemisphere (GOES-West)', (1826, 3464): 'Northern Hemisphere (GOES-East)', (2704, 5208): 'Full Disc'}

```
vis_sectors = {(566, 3464): 'Southern Hemisphere (GOES-East)', (1062, 2760):
'Southern Hemisphere (GOES-West)', (1354, 3312): 'Northern Hemisphere (GOES-West)',
(1826, 3464): 'Northern Hemisphere (GOES-East)', (2704, 5208): 'Full Disc'}
```

class satpy.readers.goes_imager_nc.**GOESNCBaseFileHandler**(*filename*, *filename_info*, *filetype_info*, *geo_data=None*)

Bases: *BaseFileHandler*

File handler for GOES Imager data in netCDF format.

Initialize the reader.

_calibrate(*radiance*, *coefs*, *channel*, *calibration*)

Convert radiance to reflectance or brightness temperature.

static _calibrate_ir(*radiance*, *coefs*)

Convert IR radiance to brightness temperature.

Reference: [IR]

Parameters

- **radiance** – Radiance [mW m-2 cm-1 sr-1]
- **coefs** – Dictionary of calibration coefficients. Keys: n: The channel's central wavenumber [cm-1] a: Offset [K] b: Slope [1] btmin: Minimum brightness temperature threshold [K] btmax: Maximum brightness temperature threshold [K]

Returns
Brightness temperature [K]

static _calibrate_vis(*radiance*, *k*)

Convert VIS radiance to reflectance.

Note: Angle of incident radiation and annual variation of the earth-sun distance is not taken into account. A value of 100% corresponds to the radiance of a perfectly reflecting diffuse surface illuminated at normal incidence when the sun is at its annual-average distance from the Earth.

TODO: Take angle of incident radiation (cos sza) and annual variation of the earth-sun distance into account.

Reference: [VIS]

Parameters

- **radiance** – Radiance [mW m-2 cm-1 sr-1]
- **k** – pi / H, where H is the solar spectral irradiance at annual-average sun-earth distance, averaged over the spectral response function of the detector). Units of k: [m2 um sr W-1]

Returns
Reflectance [%]

_counts2radiance(*counts*, *coefs*, *channel*)

Convert raw detector counts to radiance.

_get_area_def_uniform_sampling(*lon0*, *channel*)

Get area definition with uniform sampling.

static _get_earth_mask(*lat*)

Identify earth/space pixels.

Returns
Mask (1=earth, 0=space)

`static _get_nadir_pixel(earth_mask, sector)`

Find the nadir pixel.

Parameters

- **earth_mask** – Mask identifying earth and space pixels
- **sector** – Specifies the scanned sector

Returns
nadir row, nadir column

`static _get_platform_name(ncattr)`

Determine name of the platform.

`_get_sector(channel, nlines, ncols)`

Determine which sector was scanned.

`static _ircounts2radiance(counts, scale, offset)`

Convert IR counts to radiance.

Reference: [IR].

Parameters

- **counts** – Raw detector counts
- **scale** – Scale [mW-1 m2 cm sr]
- **offset** – Offset [1]

Returns
Radiance [mW m-2 cm-1 sr-1]

`_is_yaw_flip(lat)`

Determine whether the satellite is yaw-flipped ('upside down').

`_update_metadata(data, ds_info)`

Update metadata of the given DataArray.

`static _viscounts2radiance(counts, slope, offset)`

Convert VIS counts to radiance.

References: [VIS]

Parameters

- **counts** – Raw detector counts
- **slope** – Slope [W m-2 um-1 sr-1]
- **offset** – Offset [W m-2 um-1 sr-1]

Returns
Radiance [W m-2 um-1 sr-1]

`available_datasets(configured_datasets=None)`

Update information for or add datasets provided by this file.

If this file handler can load a dataset then it will supplement the dataset info with the resolution and possibly coordinate datasets needed to load it. Otherwise it will continue passing the dataset information down the chain.

See `satpy.readers.file_handlers.BaseFileHandler.available_datasets()` for details.

abstract calibrate(*data*, *calibration*, *channel*)

Perform calibration.

property end_time

End timestamp of the dataset.

abstract get_dataset(*key*, *info*)

Load dataset designated by the given key from file.

get_shape(*key*, *info*)

Get the shape of the data.

Returns
Number of lines, number of columns

abstract property ir_sectors

Get the ir sectors.

property meta

Derive metadata from the coordinates.

property resolution

Specify the spatial resolution of the dataset.

Channel 13_3's spatial resolution changes from one platform to another while the wavelength and file format remain the same. In order to avoid multiple YAML reader definitions for the same file format, read the channel's resolution from the file instead of defining it in the YAML dataset. This information will then be used by the YAML reader to complement the YAML definition of the dataset.

Returns
Spatial resolution in kilometers

property start_time

Start timestamp of the dataset.

abstract property vis_sectors

Get the vis sectors.

yaw_flip_sampling_distance = 10

class satpy.readers.goes_imager_nc.**GOESNCFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: `GOESNCBaseFileHandler`

File handler for GOES Imager data in netCDF format.

Initialize the reader.

calibrate(*counts*, *calibration*, *channel*)

Perform calibration.

get_dataset(*key*, *info*)

Load dataset designated by the given key from file.

ir_sectors = {(566, 3464): 'Southern Hemisphere (GOES-East)', (1062, 2760): 'Southern Hemisphere (GOES-West)', (1354, 3312): 'Northern Hemisphere (GOES-West)', (1826, 3464): 'Northern Hemisphere (GOES-East)', (2704, 5208): 'Full Disc'}

```
vis_sectors = {(2267, 13852):  'Southern Hemisphere (GOES-East)', (4251, 11044):
'Southern Hemisphere (GOES-West)', (5419, 13244):  'Northern Hemisphere
(GOES-West)', (7307, 13852):  'Northern Hemisphere (GOES-East)', (10819, 20800):
'Full Disc'}
```

satpy.readers.goes_imager_nc.**is_vis_channel**(*channel*)

Determine whether the given channel is a visible channel.

satpy.readers.goes_imager_nc.**test_coefs**(*ir_url*, *vis_url*)

Test calibration coefficients against NOAA reference pages.

Currently the reference pages are:

ir_url = https://www.ospo.noaa.gov/Operations/GOES/calibration/gvar-conversion.html vis_url = https://www.ospo.noaa.gov/Operations/GOES/calibration/goes-vis-ch-calibration.html

Parameters

- **ir_url** – Path or URL to HTML page with IR coefficients
- **vis_url** – Path or URL to HTML page with VIS coefficients

Raises

ValueError if coefficients don't match the reference –

satpy.readers.gpm_imerg module

Reader for GPM imerg data on half-hourly timesteps.

References

- The NASA IMERG ATBD: https://pmm.nasa.gov/sites/default/files/document_files/IMERG_ATBD_V06.pdf

class satpy.readers.gpm_imerg.**Hdf5IMERG**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

IMERG hdf5 reader.

Init method.

property end_time

Find the end time from filename info.

get_area_def(*dsid*)

Create area definition from the gridded lat/lon values.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset.

property start_time

Find the start time from filename info.

satpy.readers.grib module

Generic Reader for GRIB2 files.

Currently this reader depends on the *pygrib* python package. The *eccodes* package from ECMWF is preferred, but does not support python 3 at the time of writing.

class satpy.readers.grib.**GRIBFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: BaseFileHandler

Generic GRIB file handler.

Open grib file and do initial message parsing.

_analyze_messages(*grib_file*)

_area_def_from_msg(*msg*)

static _convert_datetime(*msg*, *date_key*, *time_key*, *date_format='%Y%m%d%H%M'*)

static _correct_cyl_minmax_xy(*proj_params*, *min_lon*, *min_lat*, *max_lon*, *max_lat*)

static _correct_proj_params_over_prime_meridian(*proj_params*)

_create_dataset_ids(*keys*)

_get_area_info(*msg*, *proj_params*)

static _get_corner_lonlat(*proj_params*, *lons*, *lats*)

static _get_corner_xy(*proj_params*, *lons*, *lats*, *scans_positively*)

_get_cyl_area_info(*msg*, *proj_params*)

static _get_cyl_minmax_lonlat(*lons*, *lats*)

static _get_extents(*min_x*, *min_y*, *max_x*, *max_y*, *shape*)

_get_message(*ds_info*)

available_datasets(*configured_datasets=None*)

Automatically determine datasets provided by this file.

property end_time

Get end time of this entire file.

Assumes the last message is the latest message.

get_area_def(*dsid*)

Get area definition for message.

If latlong grid then convert to valid eqc grid.

get_dataset(*dataset_id*, *ds_info*)

Read a GRIB message into an xarray DataArray.

get_metadata(*msg*, *ds_info*)

Get metadata.

property start_time

Get start time of this entire file.

Assumes the first message is the earliest message.

satpy.readers.hdf4_utils module

Helpers for reading hdf4-based files.

class satpy.readers.hdf4_utils.**HDF4FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Base class for common HDF4 operations.

Open file and collect information.

_collect_attrs(*name*, *attrs*)

_open_xarray_dataset(*val*, *chunks=4096*)

Read the band in blocks.

collect_metadata(*name*, *obj*)

Collect all metadata about file content.

get(*item*, *default=None*)

Get variable as DataArray or return the default.

satpy.readers.hdf4_utils.**from_sds**(*var*, **args*, ***kwargs*)

Create a dask array from a SD dataset.

satpy.readers.hdf5_utils module

Helpers for reading hdf5-based files.

class satpy.readers.hdf5_utils.**HDF5FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Small class for inspecting a HDF5 file and retrieve its metadata/header data.

Initialize file handler.

_collect_attrs(*name*, *attrs*)

_get_reference(*hf*, *ref*)

collect_metadata(*name*, *obj*)

Collect metadata.

get(*item*, *default=None*)

Get item.

get_reference(*name*, *key*)

Get reference.

satpy.readers.hdfeos_base module

Base HDF-EOS reader.

class satpy.readers.hdfeos_base.**HDFEOSBaseFileReader**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *BaseFileHandler*

Base file handler for HDF EOS data for both L1b and L2 products.

Initialize the base reader.

_add_satpy_metadata(*data_id:* DataID, *data_arr:* *DataArray*)

Add metadata that is specific to Satpy.

_chunks_for_variable(*hdf_dataset*)

_get_good_data_mask(*data_arr*, *is_category=False*)

static _get_res_multiplier(*var_shape*)

_load_all_metadata_attributes()

_platform_name_from_filename()

_read_dataset_in_file(*dataset_name*)

classmethod _read_mda(*lines*, *element=None*)

_resolution_to_rows_per_scan(*resolution:* *int*) → int

_scale_and_mask_data_array(*data*, *is_category=False*)

Unscale byte data and mask invalid/fill values.

MODIS requires unscaling the in-file bytes in an unexpected way:

```
data = (byte_value - add_offset) * scale_factor
```

See the below L1B User's Guide Appendix C for more information:

https://mcst.gsfc.nasa.gov/sites/default/files/file_attachments/M1054E_PUG_2017_0901_V6.2.2_Terra_V6.2.1_Aqua.pdf

classmethod _split_line(*line*, *lines*)

_start_time_from_filename()

property end_time

Get the end time of the dataset.

load_dataset(*dataset_name*, *is_category=False*)

Load the dataset from HDF EOS file.

property metadata_platform_name

Platform name from the internal file metadata.

classmethod read_mda(*attribute*)

Read the EOS metadata.

property start_time

Get the start time of the dataset.

class satpy.readers.hdfeos_base.**HDFEOSGeoReader**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDFEOSBaseFileReader*

Handler for the geographical datasets.

Initialize the geographical reader.

DATASET_NAMES = {'latitude': 'Latitude', 'longitude': 'Longitude', 'satellite_azimuth_angle': ('SensorAzimuth', 'Sensor_Azimuth'), 'satellite_zenith_angle': ('SensorZenith', 'Sensor_Zenith'), 'solar_azimuth_angle': ('SolarAzimuth', 'SolarAzimuth'), 'solar_zenith_angle': ('SolarZenith', 'Solar_Zenith')}

static _geo_resolution_for_l1b(*metadata*)

static _geo_resolution_for_l2_l1b(*metadata*)

_load_ds_by_name(*ds_name*)

Attempt loading using multiple common names.

property geo_resolution

Resolution of the geographical data retrieved in the metadata.

get_dataset(*dataset_id:* DataID, *dataset_info:* *dict*) → DataArray

Get the geolocation dataset.

get_interpolated_dataset(*name1*, *name2*, *resolution*, *offset=0*)

Load and interpolate datasets.

static is_geo_loadable_dataset(*dataset_name:* *str*) → bool

Determine if this dataset should be loaded as a Geo dataset.

static read_geo_resolution(*metadata*)

Parse metadata to find the geolocation resolution.

satpy.readers.hdfeos_base.**_find_and_run_interpolation**(*interpolation_functions*, *src_resolution*, *dst_resolution*, *args*)

satpy.readers.hdfeos_base.**_interpolate_no_angles**(*clons*, *clats*, *src_resolution*, *dst_resolution*)

satpy.readers.hdfeos_base.**_interpolate_with_angles**(*clons*, *clats*, *csatz*, *src_resolution*, *dst_resolution*)

satpy.readers.hdfeos_base.**interpolate**(*clons*, *clats*, *csatz*, *src_resolution*, *dst_resolution*)

Interpolate two parallel datasets jointly.

satpy.readers.hrit_base module

HRIT/LRIT format reader.

This module is the base module for all HRIT-based formats. Here, you will find the common building blocks for hrit reading.

One of the features here is the on-the-fly decompression of hrit files. It needs a path to the xRITDecompress binary to be provided through the environment variable called XRIT_DECOMPRESS_PATH. When compressed hrit files are then encountered (files finishing with *.C_*), they are decompressed to the system's temporary directory for reading.

class satpy.readers.hrit_base.**HRITFileHandler**(*filename*, *filename_info*, *filetype_info*, *hdr_info*)

Bases: BaseFileHandler

HRIT standard format reader.

Initialize the reader.

_get_hd(*hdr_info*)

Open the file, read and get the basic file header info and set the mda dictionary.

_get_output_info()

property end_time

Get end time.

get_area_def(*dsid*)

Get the area definition of the band.

get_area_extent(*size*, *offsets*, *factors*, *platform_height*)

Get the area extent of the file.

get_dataset(*key*, *info*)

Load a dataset.

get_xy_from_linecol(*line*, *col*, *offsets*, *factors*)

Get the intermediate coordinates from line & col.

Intermediate coordinates are actually the instruments scanning angles.

property observation_end_time

Get observation end time.

property observation_start_time

Get observation start time.

read_band(*key*, *info*)

Read the data.

property start_time

Get start time.

class satpy.readers.hrit_base.**HRITSegment**(*filename*, *mda*)

Bases: object

An HRIT segment with data.

Set up the segment.

_get_input_info()

_is_file_like()

_read_data_from_disk()

_read_data_from_file()

_read_file_like()

read_data()

Read the data.

satpy.readers.hrit_base.**decompress**(*infile*, *outdir='.'*)

Decompress an XRIT data file and return the path to the decompressed file.

It expect to find Eumetsat's xRITDecompress through the environment variable XRIT_DECOMPRESS_PATH.

satpy.readers.hrit_base.**decompressed**(*filename*)

Decompress context manager.

satpy.readers.hrit_base.**get_header_content**(*fp*, *header_dtype*, *count=1*)

Return the content of the HRIT header.

satpy.readers.hrit_base.**get_header_id**(*fp*)

Return the HRIT header common data.

satpy.readers.hrit_base.**get_xritdecompress_cmd**()

Find a valid binary for the xRITDecompress command.

satpy.readers.hrit_base.**get_xritdecompress_outfile**(*stdout*)

Analyse the output of the xRITDecompress command call and return the file.

satpy.readers.hrit_jma module

HRIT format reader for JMA data.

Introduction

The JMA HRIT format is described in the JMA HRIT - Mission Specific Implementation. There are three readers for this format in Satpy:

- `jami_hrit`: For data from the *JAMI* instrument on MTSAT-1R
- `mtsat2-imager_hrit`: For data from the *Imager* instrument on MTSAT-2
- `ahi_hrit`: For data from the *AHI* instrument on Himawari-8/9

Although the data format is identical, the instruments have different characteristics, which is why there is a dedicated reader for each of them. Sample data is available here:

- JAMI/Imager sample data
- AHI sample data

Example:

Here is an example how to read Himwari-8 HRIT data with Satpy:

```
from satpy import Scene
import glob

filenames = glob.glob('data/IMG_DK01B14_2018011109*')
scn = Scene(filenames=filenames, reader='ahi_hrit')
scn.load(['B14'])
print(scn['B14'])
```

Output:

```
<xarray.DataArray (y: 5500, x: 5500)>
dask.array<concatenate, shape=(5500, 5500), dtype=float64, chunksize=(550, 4096), ...
Coordinates:
    acq_time  (y) datetime64[ns] 2018-01-11T09:00:20.995200 ... 2018-01-11T09:09:40.
↪348800
    crs       object +proj=geos +lon_0=140.7 +h=35785831 +x_0=0 +y_0=0 +a=6378169 ...
  * y         (y) float64 5.5e+06 5.498e+06 5.496e+06 ... -5.496e+06 -5.498e+06
  * x         (x) float64 -5.498e+06 -5.496e+06 -5.494e+06 ... 5.498e+06 5.5e+06
Attributes:
    orbital_parameters:   {'projection_longitude': 140.7, 'projection_latitud...
    standard_name:        toa_brightness_temperature
    level:                None
    wavelength:           (11.0, 11.2, 11.4)
    units:                K
    calibration:          brightness_temperature
    file_type:            ['hrit_b14_seg', 'hrit_b14_fd']
    modifiers:            ()
    polarization:         None
    sensor:               ahi
    name:                 B14
    platform_name:        Himawari-8
    resolution:           4000
    start_time:           2018-01-11 09:00:20.995200
    end_time:             2018-01-11 09:09:40.348800
    area:                 Area ID: FLDK, Description: Full Disk, Projection I...
    ancillary_variables:  []
```

JMA HRIT data contain the scanline acquisition time for only a subset of scanlines. Timestamps of the remaining scanlines are computed using linear interpolation. This is what you'll find in the `acq_time` coordinate of the dataset.

Compression

Gzip-compressed MTSAT files can be decompressed on the fly using *FSFile*:

```
import fsspec
from satpy import Scene
from satpy.readers import FSFile

filename = "/data/HRIT_MTSAT1_20090101_0630_DK01IR1.gz"
open_file = fsspec.open(filename, compression="gzip")
fs_file = FSFile(open_file)
scn = Scene([fs_file], reader="jami_hrit")
scn.load(["IR1"])
```

class satpy.readers.hrit_jma.**HRITJMAFileHandler**(*filename*, *filename_info*, *filetype_info*, *use_acquisition_time_as_start_time=False*)

Bases: *HRITFileHandler*

JMA HRIT format reader.

By default, the reader uses the start time parsed from the filename. To use exact time, computed from the metadata, the user can define a keyword argument:

```
scene = Scene(filenames=filenames,
              reader='ahi_hrit',
              reader_kwargs={'use_acquisition_time_as_start_time': True})
```

As this time is different for every channel, time-dependent calculations like SZA correction can be pretty slow when multiple channels are used.

The exact scanline times are always available as coordinates of an individual channels:

```
scene.load(["B03"])
print(scene["B03].coords["acq_time"].data)
```

would print something similar to:

```
array(['2021-12-08T06:00:20.131200000', '2021-12-08T06:00:20.191948000',
       '2021-12-08T06:00:20.252695000', ...,
       '2021-12-08T06:09:39.449390000', '2021-12-08T06:09:39.510295000',
       '2021-12-08T06:09:39.571200000'], dtype='datetime64[ns]')
```

The first value represents the exact start time, and the last one the exact end time of the data acquisition.

Initialize the reader.

_check_sensor_platform_consistency(*sensor*)

Make sure sensor and platform are consistent.

Parameters

sensor (*str*) – Sensor name from YAML dataset definition

Raises

ValueError if they don't match –

_get_acq_time()

Get the acquisition times from the file.

Acquisition times for a subset of scanlines are stored in the header as follows:

b'LINE:=1rTIME:=54365.022558rLINE:=21rTIME:=54365.022664r…'

Missing timestamps in between are computed using linear interpolation.

_get_area_def()

Get the area definition of the band.

_get_line_offset()

Get line offset for the current segment.

Read line offset from the file and adapt it to the current segment or half disk scan so that

y(l) ~ l - loff

because this is what get_geostationary_area_extent() expects.

_get_platform()

Get the platform name.

The platform is not specified explicitly in JMA HRIT files. For segmented data it is not even specified in the filename. But it can be derived indirectly from the projection name:

GEOS(140.00): MTSAT-1R GEOS(140.25): MTSAT-1R # TODO: Check if there is more… GEOS(140.70): Himawari-8 GEOS(145.00): MTSAT-2

See [MTSAT], section 3.1. Unfortunately Himawari-8 and 9 are not distinguishable using that method at the moment. From [HIMAWARI]:

"HRIT/LRIT files have the same file naming convention in the same format in Himawari-8 and Himawari-9, so there is no particular difference."

TODO: Find another way to distinguish Himawari-8 and 9.

References: [MTSAT] http://www.data.jma.go.jp/mscweb/notice/Himawari7_e.html [HIMAWARI] http://www.data.jma.go.jp/mscweb/en/himawari89/space_segment/sample_hrit.html

static _interp(*arr*, *cal*)

_mask_space(*data*)

Mask space pixels.

calibrate(*data*, *calibration*)

Calibrate the data.

property end_time

Get end time of the scan.

get_area_def(*dsid*)

Get the area definition of the band.

get_dataset(*key*, *info*)

Get the dataset designated by *key*.

property start_time

Get start time of the scan.

satpy.readers.hrit_jma.**mjd2datetime64**(*mjd*)

Convert Modified Julian Day (MJD) to datetime64.

satpy.readers.hrpt module

Reading and calibrating hrpt avhrr data.

Todo: - AMSU - Compare output with AAPP

Reading: http://www.ncdc.noaa.gov/oa/pod-guide/ncdc/docs/klm/html/c4/sec4-1.htm#t413-1

Calibration: http://www.ncdc.noaa.gov/oa/pod-guide/ncdc/docs/klm/html/c7/sec7-1.htm

class satpy.readers.hrpt.**HRPTFile**(*filename*, *filename_info*, *filetype_info*)

Bases: `BaseFileHandler`

Reader for HRPT Minor Frame, 10 bits data expanded to 16 bits.

Init the file handler.

property _chunks

Get the best chunks for this data.

property _data

Get the data.

_get_avhrr_tiepoints(*scan_points*, *scanline_nb*)

_get_ch3_mask_or_true(*key*)

_get_channel_data(*key*)

Get channel data.

_get_navigation_data(*key*)

Get navigation data.

property _is3b

calibrate_solar_channel(*data*, *key*)

Calibrate a solar channel.

calibrate_thermal_channel(*data*, *key*)

Calibrate a thermal channel.

property calibrator

Create a calibrator for the data.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Get the dataset.

property lons_lats

Get the lons and lats.

property platform_name

Get the platform name.

read()

Read the file.

property start_time

Get the start time.

property telemetry

Get the telemetry.

property times

Get the timestamps for each line.

satpy.readers.hrpt.**_get_channel_index**(*key*)

Get the avhrr channel index.

satpy.readers.hrpt.**bfield**(*array*, *bit*)

Return the bit array.

satpy.readers.hrpt.**geo_interpolate**(*lons32km*, *lats32km*)

Interpolate geo data.

satpy.readers.hrpt.**time_seconds**(*tc_array*, *year*)

Return the time object from the timecodes.

satpy.readers.hsaf_grib module

A reader for files produced by the Hydrology SAF.

Currently this reader depends on the *pygrib* python package. The *eccodes* package from ECMWF is preferred, but does not support python 3 at the time of writing.

class satpy.readers.hsaf_grib.**HSAFFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for HSAF grib files.

Init the file handler.

_get_area_def(*msg*)

Get the area definition of the datasets in the file.

static _get_datetime(*msg*)

_get_message(*idx*)

property analysis_time

Get validity time of this file.

get_area_def(*dsid*)

Get area definition for message.

get_dataset(*ds_id*, *ds_info*)

Read a GRIB message into an xarray DataArray.

get_metadata(*msg*)

Get the metadata.

satpy.readers.hsaf_h5 module

A reader for HDF5 Snow Cover (SC) file produced by the Hydrology SAF.

class satpy.readers.hsaf_h5.**HSAFFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for HSAF H5 files.

Init the file handler.

_get_area_def()

Area definition for h10 - hardcoded.

Area definition not available in the HDF5 message, so using hardcoded one (it's known).

```
hsaf_h10:
  description: H SAF H10 area definition
  projection:
    proj: geos
    lon_0: 0
    h: 35785831
    x_0: 0
    y_0: 0
    a: 6378169
```

(continues on next page)

(continued from previous page)

```
    rf: 295.488065897001
    no_defs: null
    type: crs
  shape:
    height: 916
    width: 1902
  area_extent:
    lower_left_xy: [-1936760.3163240477, 2635854.280233425]
    upper_right_xy: [3770006.7195370505, 5384223.683413638]
    units: m
```

_get_dataset(*ds_name*)

_prepare_variable_for_palette(*dset*, *ds_info*)

property end_time

Get end time.

get_area_def(*dsid*)

Area definition for h10 SC dataset.

Since it is not available in the HDF5 message, using hardcoded one (it's known).

get_dataset(*ds_id*, *ds_info*)

Read a HDF5 file into an xarray DataArray.

get_metadata(*dset*, *name*)

Get the metadata.

property start_time

Get start time.

satpy.readers.hy2_scat_l2b_h5 module

HY-2B L2B Reader.

Distributed by Eumetsat in HDF5 format. Also handle the HDF5 files from NSOAS, based on a file example.

class satpy.readers.hy2_scat_l2b_h5.**HY2SCATL2BH5FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

File handler for HY2 scat.

Initialize file handler.

_mask_data(*data*)

_scale_data(*data*)

property end_time

Time for final observation.

get_dataset(*key*, *info*)

Get the dataset.

get_metadata()

Get the metadata.

get_variable_metadata()

Get the variable metadata.

property platform_name

Get the Platform ShortName.

property start_time

Time for first observation.

satpy.readers.iasi_l2 module

IASI L2 files.

class satpy.readers.iasi_l2.**IASIL2CDRNC**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FsspecFileHandler*

Reader for IASI L2 CDR in NetCDF format.

Reader for IASI All Sky Temperature and Humidity Profiles - Climate Data Record Release 1.1 - Metop-A and -B. Data and documentation are available from http://doi.org/10.15770/EUM_SEC_CLM_0063. Data are also available from the EUMETSAT Data Store under ID EO:EUM:DAT:0576.

Initialize object.

available_datasets(*configured_datasets=None*)

Get available datasets based on what's in the file.

Returns all datasets in the root group.

get_dataset(*data_id*, *ds_info*)

Obtain dataset.

class satpy.readers.iasi_l2.**IASIL2HDF5**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for IASI L2 HDF5 files.

Init the file handler.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Get the start time.

satpy.readers.iasi_l2.**_form_datetimes**(*days*, *msecs*)

Calculate seconds since EPOCH from days and milliseconds for each of IASI scan.

satpy.readers.iasi_l2.**read_dataset**(*fid*, *key*)

Read dataset.

satpy.readers.iasi_l2.**read_geo**(*fid*, *key*)

Read geolocation and related datasets.

satpy.readers.iasi_l2_so2_bufr module

IASI L2 SO2 BUFR format reader.

Introduction

The `iasi_l2_so2_bufr` reader reads IASI level2 SO2 data in BUFR format. The algorithm is described in the Theoretical Basis Document, linked below.

Each BUFR file consists of a number of messages, one for each scan, each of which contains SO2 column amounts in Dobson units for retrievals performed with plume heights of 7, 10, 13, 16 and 25 km.

Reader Arguments

A list of retrieval files, fnames, can be opened as follows:

```
Scene(reader="iasi_l2_so2_bufr", filenames=fnames)
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene
import glob

filenames = glob.glob(
    '/test_data/W_XX-EUMETSAT-Darmstadt,SOUNDING+SATELLITE,METOPA+IASI_C_EUMC_
↪20200204091455_68984_eps_o_so2_l2.bin')
scn = Scene(filenames=filenames, reader='iasi_l2_so2_bufr')
scn.load(['so2_height_3', 'so2_height_4'])
print(scn['so2_height_3'])
```

Output:

```
<xarray.DataArray 'so2_height_3' (y: 23, x: 120)>
dask.array<where, shape=(23, 120), dtype=float64, chunksize=(1, 120), chunktype=numpy.
↪ndarray>
Coordinates:
    crs      object +proj=latlong +datum=WGS84 +ellps=WGS84 +type=crs
Dimensions without coordinates: y, x
Attributes:
    sensor:               IASI
    units:                dobson
    file_type:            iasi_l2_so2_bufr
    wavelength:           None
    modifiers:            ()
    platform_name:        METOP-2
```

(continues on next page)

(continued from previous page)

```
    resolution:           12000
    fill_value:           -1e+100
    level:                None
    polarization:         None
    coordinates:          ('longitude', 'latitude')
    calibration:          None
    key:                  #3#sulphurDioxide
    name:                 so2_height_3
    start_time:           2020-02-04 09:14:55
    end_time:             2020-02-04 09:17:51
    area:                 Shape: (23, 120)\nLons: <xarray.DataArray 'longitud...
    ancillary_variables:  []
```

References: Algorithm Theoretical Basis Document: https://acsaf.org/docs/atbd/Algorithm_Theoretical_Basis_Document_IASI_SO2_Jul_2016.pdf

class satpy.readers.iasi_l2_so2_bufr.**IASIL2SO2BUFR**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *BaseFileHandler*

File handler for the IASI L2 SO2 BUFR product.

Initialise the file handler for the IASI L2 SO2 BUFR data.

property end_time

Return the end time of data acquisition.

get_array(*key*)

Get all data from file for the given BUFR key.

get_attribute(*key*)

Get BUFR attributes.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using the BUFR key in dataset_info.

get_start_end_date()

Get the first and last date from the bufr file.

property platform_name

Return spacecraft name.

property start_time

Return the start time of data acqusition.

satpy.readers.ici_l1b_nc module

EUMETSAT EPS-SG Ice Cloud Imager (ICI) Level 1B products reader.

The format is explained in the EPS-SG ICI Level 1B Product Format Specification V3A.

This version is applicable for the ici test data released in Jan 2021.

class satpy.readers.ici_l1b_nc.**IciL1bNCFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *NetCDF4FileHandler*

Reader class for ICI L1B products in netCDF format.

Read the calibration data and prepare the class for dataset reading.

_calibrate(*variable*, *dataset_info*)

Perform the calibration.

Parameters

- **variable** – xarray DataArray containing the dataset to calibrate.
- **dataset_info** – dictionary of information about the dataset.

Returns

array containing the calibrated values and all the original metadata.

Return type
DataArray

static _calibrate_bt(*radiance*, *cw*, *a*, *b*)

Perform the calibration to brightness temperature.

Parameters

- **radiance** – xarray DataArray or numpy ndarray containing the radiance values.
- **cw** – center wavenumber [cm-1].
- **a** – temperature coefficient [-].
- **b** – temperature coefficient [K].

Returns

array containing the calibrated brightness temperature values.

Return type
DataArray

static _drop_coords(*variable*)

Drop coords that are not in dims.

_fetch_variable(*var_key*)

Fetch variable.

_filter_variable(*variable*, *dataset_info*)

Filter variable in the third dimension.

_get_global_attributes()

Create a dictionary of global attributes.

_get_quality_attributes()

Get quality attributes.

static _get_third_dimension_name(*variable*)

Get name of the third dimension of the variable.

_interpolate(*interpolation_type*)

Interpolate from tie points to pixel points.

static _interpolate_geo(*longitude*, *latitude*, *n_samples*)

Perform the interpolation of geographic coordinates from tie points to pixel points.

Parameters

- **longitude** – xarray DataArray containing the longitude dataset to interpolate.
- **latitude** – xarray DataArray containing the longitude dataset to interpolate.
- **n_samples** – int describing number of samples per scan to interpolate onto.

Returns

tuple of arrays containing the interpolate values, all the original
metadata and the updated dimension names.

`_interpolate_viewing_angle`(*azimuth*, *zenith*, *n_samples*)

Perform the interpolation of angular coordinates from tie points to pixel points.

Parameters

- **azimuth** – xarray DataArray containing the azimuth angle dataset to interpolate.
- **zenith** – xarray DataArray containing the zenith angle dataset to interpolate.
- **n_samples** – int describing number of samples per scan to interpolate onto.

Returns

tuple of arrays containing the interpolate values, all the original
metadata and the updated dimension names.

`_manage_attributes`(*variable*, *dataset_info*)

Manage attributes of the dataset.

`_orthorectify`(*variable*, *orthorect_data_name*)

Perform the orthorectification.

Parameters

- **variable** – xarray DataArray containing the dataset to correct for orthorectification.
- **orthorect_data_name** – name of the orthorectification correction data in the product.

Returns

array containing the corrected values and all the
original metadata.

Return type
DataArray

`static _standardize_dims`(*variable*)

Standardize dims to y, x.

`property end_time`

Get observation end time.

`get_dataset`(*dataset_id*, *dataset_info*)

Get dataset using file_key in dataset_info.

`property latitude`

Get latitude coordinates.

`property longitude`

Get longitude coordinates.

`property longitude_and_latitude`

Get longitude and latitude coordinates.

property observation_azimuth

Get observation azimuth angles.

property observation_azimuth_and_zenith

Get observation azimuth and zenith angles.

property observation_zenith

Get observation zenith angles.

property platform_name

Return platform name.

property sensor

Return sensor.

property solar_azimuth

Get solar azimuth angles.

property solar_azimuth_and_zenith

Get solar azimuth and zenith angles.

property solar_zenith

Get solar zenith angles.

property ssp_lon

Return subsatellite point longitude.

property start_time

Get observation start time.

class satpy.readers.ici_l1b_nc.**InterpolationType**(*value*)

Bases: Enum

Enum for interpolation types.

LONLAT = 0

OBSERVATION_ANGLES = 2

SOLAR_ANGLES = 1

satpy.readers.insat3d_img_l1b_h5 module

File handler for Insat 3D L1B data in hdf5 format.

class satpy.readers.insat3d_img_l1b_h5.**Insat3DIMGL1BH5FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for insat 3d imager data.

Initialize file handler.

property datatree

Create the datatree.

property end_time

Get the end time.

get_area_def(*ds_id*)

Get the area definition.

get_dataset(*ds_id*, *ds_info*)

Get a data array.

property start_time

Get the start time.

satpy.readers.insat3d_img_l1b_h5.**_rename_dims**(*ds*)

Rename dimensions to satpy standards.

satpy.readers.insat3d_img_l1b_h5.**apply_lut**(*data*, *lut*)

Apply a lookup table.

satpy.readers.insat3d_img_l1b_h5.**decode_lut_arr**(*arr*, *lut*)

Decode an array using a lookup table.

satpy.readers.insat3d_img_l1b_h5.**get_lonlat_suffix**(*resolution*)

Get the lonlat variable suffix from the resolution.

satpy.readers.insat3d_img_l1b_h5.**open_dataset**(*filename*, *resolution=1000*)

Open a dataset for a given resolution.

satpy.readers.insat3d_img_l1b_h5.**open_datatree**(*filename*)

Open a datatree.

satpy.readers.li_base_nc module

Base class used for the MTG Lighting Imager netCDF4 readers.

The base LI reader class supports generating the available datasets programmatically: to achieve this, each LI product type should provide a "file description" which is itself retrieved directly from the YAML configuration file for the reader of interest, as a custom `file_desc` entry inside the 'file_type' section corresponding to that product type.

Each of the `file_desc` entry describes what are the variables that are available into that product that should be used to register the available satpy datasets.

Each of those description entries may contain the following elements:

- **product_type** [required]:

  Indicate the processing_level / product_type name to use internally for that type of product file. This should correspond to the `{processing_level}-{product_type}` part of the full file_pattern.

- **search_paths** [optional]:

  A list of the possible paths that should be prefixed to a given variable name when searching for that variable in the NetCDF file to register a dataset on it. The list is given in priority order. If no search path is provided (or an empty array is provided) then the variables will only be searched directly in the root group of the NetCDF structure.

- **swath_coordinates** [required]:

  The LI reader will use a `SwathDefinition` object to define the area/coordinates of each of the provided datasets depending on the content of this entry. The user can either:

  – Specify a `swath_coordinates` entry directly with `latitude` and `longitude` entries, in which case, the datasets that will match one of the `'variable_patterns'` provided will use those lat/lon variables as coordinate providers.

  - Specify a `swath_coordinates` entry directly with `projection`, `azimuth` and `elevation` entries instead, in which case, the reader will first use the variables pointed by those 3 entries compute the corresponding latitude/longitude data from the scan angles contained in the product file. And then, continue with assigned those lat/lon datasets as coordinates for datasets that will match one of the `variable_patterns` provided.

  Note: It is acceptable to specify an empty array for the list of `variable_patterns`, in this case, the swath coordinates will not be assigned to any dataset.

- **sectors** [optional]:

  The custom dataset description mechanism makes a distinction between "ordinary" variables which should be used to create a "single dataset" and "sectored variables" which will be found per sector and will thus be used to generate as many datasets as there are sectors (see below). So this entry is used to specify the list of sector names there should be available in the NetCDF structure.

- **sector_variables** [optional]:

  This entry is used to provide a list of the variables that are available **per sector** in the NetCDF file. Thus, assuming the `sectors` entry is set to the standard list `['north', 'east', 'south', 'west']`, 4 separated datasets will be registered for each variable listed here (using the conventional suffix `"{sector_name}_sector"`)

- **variables** [optional]:

  This entry is used to provide a list of "ordinary variables" (ie. variables that are not available **per sector**). Each of those variables will be used to register one dataset.

  Note: A single product may provide both the "variables" and the "sector_variables" at the same time (as this is the case for LI LEF for instance)

- **variable_transforms** [optional]:

  This entry is may be used to provide specific additional entries **per variable name** (ie. will apply to both in sector or out of sector variables) that should be added to the dataset infos when registering a dataset with that variable. While any kind of info could be added this way to the final dataset infos, we are currently using the entry mainly to provide our LI reader with the following traits which will then be used to "transform" the data of the dataset as requested on loading:

  - `broadcast_to`: if this extra info is found in a dataset_info on dataset loading, then the initial data array will be broadcast to the shape of the variable found under the variable path specified as value for that entry. Note that, if the pattern `{sector_name}` if found in this entry value, then the reader will assume that we are writing a dataset from an in sector variable, and use the current sector name to find the appropriate alternate variable that will be used as reference to broadcast the current variable data.

  - `seconds_to_datetime`: This transformation is used to internally convert variables provided as float values to the `np.datetime64` data type. The value specified for this entry should be the reference epoch time used as offsets for the elapsed seconds when converting the data.

  - `seconds_to_timedelta`: This transformation is used to internally convert variables (assumed to use a "second" unit) provided as float values to the `np.timedelta64` data type. This entry should be set to `true` to activate this transform. During the conversion, we internally use a nanosecond resolution on the input floating point second values.

  - `milliseconds_to_timedelta`: Same kind of transformation as `seconds_to_timedelta` except that the source data is assumed to contain millisecond float values.

  - `accumulate_index_offset`: if this extra info is found in a `dataset_info` on dataset loading, then we will consider that the dataset currently being generated is an array of indices inside the variable pointed by the path provided as value for that entry. Note that the same usage of the pattern `{sector_name}` mentioned for the entry "broadcast_to" will also apply here. This behavior is useful when multiple input files are loaded

together in a single satpy scene, in which case, the variables from each files will be concatenated to produce a single dataset for each variable, and thus the need to correct the reported indices accordingly.

An example of usage of this entry is as follows:

```
variable_transforms:
  integration_frame_index:
    accumulate_index_offset: "{sector_name}/exposure_time"
```

In the example above the integration_frame_index from each sector (i.e. optical channel) provides a list of indices in the corresponding exposure_time array from that same sector. The final indices will thus correctly take into account that the final exposure_time array contains all the values concatenated from all the input files in the scene.

- `use_rescaling`: By default, we currently apply variable rescaling as soon as we find one (or more) of the attributes named `'scale_factor'`, `'scaling_factor'` or `'add_offset'` in the source netcdf variable. This automatic transformation can be disabled for a given variable specifying a value of false for this extra info element, for instance:

```
variable_transforms:
  latitude:
    use_rescaling: false
```

Note: We are currently not disabling rescaling for any dataset, so that entry is not used in the current version of the YAML config files for the LI readers.

class satpy.readers.li_base_nc.**LINCFileHandler**(*filename*, *filename_info*, *filetype_info*, *cache_handle=True*)

Bases: *NetCDF4FileHandler*

Base class used as parent for the concrete LI reader classes.

Initialize LINCFileHandler.

add_provided_dataset(*ds_infos*)

Add a provided dataset to our internal list.

apply_accumulate_index_offset(*data_array*, *ds_info*)

Apply the accumulate_index_offset transform on a given array.

apply_broadcast_to(*data_array*, *ds_info*)

Apply the broadcast_to transform on a given array.

apply_fill_value(*arr*, *fill_value*)

Apply fill values, unless it is None.

apply_milliseconds_to_timedelta(*data_array*, *_ds_info*)

Apply the milliseconds_to_timedelta transform on a given array.

apply_seconds_to_datetime(*data_array*, *ds_info*)

Apply the seconds_to_datetime transform on a given array.

apply_seconds_to_timedelta(*data_array*, *_ds_info*)

Apply the seconds_to_timedelta transform on a given array.

apply_transforms(*data_array*, *ds_info*)

Apply all transformations requested in the ds_info on the provided data array.

apply_use_rescaling(*data_array*, *ds_info=None*)

Apply the use_rescaling transform on a given array.

available_datasets(*configured_datasets=None*)

Determine automatically the datasets provided by this file.

Uses a per product type dataset registration mechanism using the dataset descriptions declared in the reader construction above.

check_variable_extra_info(*ds_infos*, *vname*)

Check if we have extra infos for that variable.

combine_info(*all_infos*)

Re-implement combine_info.

This is to be able to reset our __index_offset attribute in the shared ds_info currently being updated.

property end_time

Get the end time.

generate_coords_from_scan_angles()

Generate the latitude/longitude coordinates from the scan azimuth and elevation angles.

get_coordinate_names(*ds_infos*)

Get the target coordinate names, applying the sector name as needed.

get_daskified_lon_lat(*proj_dict*)

Get daskified lon and lat array using map_blocks.

get_dataset(*dataset_id*, *ds_info=None*)

Get a dataset.

get_dataset_infos(*dname*)

Retrieve the dataset infos corresponding to one of the registered datasets.

get_first_valid_variable(*var_paths*)

Select the first valid path for a variable from the given input list and returns the data.

get_latlon_names()

Retrieve the user specified names for latitude/longitude coordinates.

Use default 'latitude' / 'longitude' if not specified.

get_measured_variable(*var_paths*, *fill_value=nan*)

Retrieve a measured variable path taking into account the potential old data formatting schema.

And also replace the missing values with the provided fill_value (except if this is explicitly set to None). Also, if a slice index is provided, only that slice of the array (on the axis=0) is retrieved (before filling the missing values).

get_projection_config()

Retrieve the projection configuration details.

get_transform_reference(*transform_name*, *ds_info*)

Retrieve a variable that should be used as reference during a transform.

get_transformed_dataset(*ds_info*)

Retrieve a dataset with all transformations applied on it.

get_variable_search_paths(*var_paths*)

Get the search paths from the dataset descriptions.

inverse_projection(*azimuth*, *elevation*, *proj_dict*)

Compute inverse projection.

is_prod_in_accumulation_grid()

Check if the current product is an accumulated product in geos grid.

register_available_datasets()

Register all the available dataset that should be made available from this file handler.

register_coords_from_scan_angles()

Register lat lon datasets in this reader.

register_dataset(*var_name*, *oc_name=None*)

Register a simple dataset given name elements.

register_sector_datasets()

Register all the available sector datasets.

register_variable_datasets()

Register all the available raw (i.e. not in sectors).

property sensor_names

List of sensors represented in this file.

property start_time

Get the start time.

update_array_attributes(*data_array*, *ds_info*)

Inject the attributes from the ds_info structure into the final data array, ignoring the internal entries.

validate_array_dimensions(*data_array*, *ds_info=None*)

Ensure that the dimensions of the provided data_array are valid.

variable_path_exists(*var_path*)

Check if a given variable path is available in the underlying netCDF file.

All we really need to do here is to access the file_content dictionary and check if we have a variable under that var_path key.

satpy.readers.li_l2_nc module

MTG Lighting Imager (LI) L2 unified reader.

This reader supports reading all the products from the LI L2 processing level:

- L2-LE
- L2-LGR
- L2-AFA
- L2-LEF
- L2-LFL
- L2-AF
- L2-AFR

class satpy.readers.li_l2_nc.**LIL2NCFileHandler**(*filename*, *filename_info*, *filetype_info*, *with_area_definition=False*)

Bases: *LINCFileHandler*

Implementation class for the unified LI L2 satpy reader.

Initialize LIL2NCFileHandler.

get_area_def(*dsid*)

Compute area definition for a dataset, only supported for accumulated products.

get_array_on_fci_grid(*data_array: DataArray*)

Obtain the accumulated products as a (sparse) 2-d array.

The array has the shape of the FCI 2 km grid (5568x5568px), and will have an AreaDefinition attached.

get_dataset(*dataset_id*, *ds_info=None*)

Get the dataset and apply gridding if requested.

is_var_with_swath_coord(*dsid*)

Check if the variable corresponding to this dataset is listed as variable with swath coordinates.

satpy.readers.maia module

Reader for NWPSAF AAPP MAIA Cloud product.

https://nwpsaf.eu/site/software/aapp/

Documentation reference:

[NWPSAF-MF-UD-003] DATA Formats [NWPSAF-MF-UD-009] MAIA version 4 Scientific User Manual

class satpy.readers.maia.**MAIAFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for Maia files.

Init the file handler.

property end_time

Get the end time.

get_dataset(*key*, *info*, *out=None*)

Get a dataset from the file.

get_platform(*platform*)

Get the platform.

read(*filename*)

Read the file.

property start_time

Get the start time.

satpy.readers.meris_nc_sen3 module

ENVISAT MERIS reader.

Sentinel 3 like format: https://earth.esa.int/eogateway/documents/20142/37627/MERIS-Sentinel-3-Like-L1-andL2-PFS.pdf

Default:
scn = Scene(filenames=my_files, reader='meris_nc_sen3')

References

- `xarray.open_dataset()`

class `satpy.readers.meris_nc_sen3.`**`NCMERIS2`**(*filename*, *filename_info*, *filetype_info*)

Bases: *NCOLCI2*

File handler for MERIS l2.

Init the file handler.

`getbitmask`(*wqsf*, *items=None*)

Get the bitmask. Experimental default mask.

class `satpy.readers.meris_nc_sen3.`**`NCMERISAngles`**(*filename*, *filename_info*, *filetype_info*)

Bases: *NCOLCIAngles*

File handler for the MERIS angles.

Init the file handler.

class `satpy.readers.meris_nc_sen3.`**`NCMERISCal`**(*filename*, *filename_info*, *filetype_info*)

Bases: *NCOLCIBase*

Dummy class for calibration.

Init the meris reader base.

class `satpy.readers.meris_nc_sen3.`**`NCMERISGeo`**(*filename*, *filename_info*, *filetype_info*)

Bases: *NCOLCIBase*

Dummy class for navigation.

Init the meris reader base.

class `satpy.readers.meris_nc_sen3.`**`NCMERISMeteo`**(*filename*, *filename_info*, *filetype_info*)

Bases: *NCOLCIMeteo*

File handler for the MERIS meteo data.

Init the file handler.

satpy.readers.mersi_l1b module

Reader for the FY-3D MERSI-2 L1B file format.

The files for this reader are HDF5 and come in four varieties; band data and geolocation data, both at 250m and 1000m resolution.

This reader was tested on FY-3D MERSI-2 data, but should work on future platforms as well assuming no file format changes.

class satpy.readers.mersi_l1b.**MERSIL1B**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

MERSI-2/MERSI-LL L1B file reader.

Initialize file handler.

_get_bt_dataset(*data*, *calibration_index*, *wave_number*)

Get the dataset as brightness temperature.

Apparently we don't use these calibration factors for Rad -> BT:

```
coeffs = self._get_coefficients(ds_info['calibration_key'], calibration_index)
# coefficients are per-scan, we need to repeat the values for a
# clean alignment
coeffs = np.repeat(coeffs, data.shape[0] // coeffs.shape[1], axis=1)
coeffs = coeffs.rename({
    coeffs.dims[0]: 'coefficients', coeffs.dims[1]: 'y'
})  # match data dims
data = coeffs[0] + coeffs[1] * data + coeffs[2] * data**2 + coeffs[3] * data**3
```

_get_coefficients(*cal_key*, *cal_index*)

_get_single_slope_intercept(*slope*, *intercept*, *cal_index*)

_mask_data(*data*, *dataset_id*, *attrs*)

Mask the data using fill_value and valid_range attributes.

_strptime(*date_attr*, *time_attr*)

Parse date/time strings.

property end_time

Time for final observation.

get_dataset(*dataset_id*, *ds_info*)

Load data variable and metadata and calibrate if needed.

property sensor_name

Map sensor name to Satpy 'standard' sensor names.

property start_time

Time for first observation.

satpy.readers.mimic_TPW2_nc module

Reader for Mimic TPW data in netCDF format from SSEC.

This module implements reader for MIMIC_TPW2 netcdf files. MIMIC-TPW2 is an experimental global product of total precipitable water (TPW), using morphological compositing of the MIRS retrieval from several available operational microwave-frequency sensors. Originally described in a 2010 paper by Wimmers and Velden. This Version 2 is developed from an older method that uses simpler, but more limited TPW retrievals and advection calculations.

More information, data and credits at http://tropic.ssec.wisc.edu/real-time/mtpw2/credits.html

class satpy.readers.mimic_TPW2_nc.**MimicTPW2FileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *NetCDF4FileHandler*

NetCDF4 reader for MIMC TPW.

Initialize the reader.

available_datasets(*configured_datasets=None*)

Get datasets in file matching gelocation shape (lat/lon).

property end_time

End timestamp of the dataset same as start_time.

get_area_def(*dsid*)

Flip data up/down and define equirectangular AreaDefintion.

get_dataset(*ds_id*, *info*)

Load dataset designated by the given key from file.

get_metadata(*data*, *info*)

Get general metadata for file.

property sensor_name

Sensor name.

property start_time

Start timestamp of the dataset determined from yaml.

satpy.readers.mirs module

Interface to MiRS product.

class satpy.readers.mirs.**MiRSL2ncHandler**(*filename*, *filename_info*, *filetype_info*, *limb_correction=True*)

Bases: *BaseFileHandler*

MiRS handler for NetCDF4 files using xarray.

The MiRS retrieval algorithm runs on multiple sensors. For the ATMS sensors, a limb correction is applied by default. In order to change that behavior, use the keyword argument `limb_correction=False`:

```
from satpy import Scene, find_files_and_readers

filenames = find_files_and_readers(base_dir, reader="mirs")
scene = Scene(filenames, reader_kwargs={'limb_correction': False})
```

Init method.

_apply_valid_range(*data_arr*, *valid_range*, *scale_factor*, *add_offset*)
Get and apply valid_range.

_available_btemp_datasets(*yaml_info*)
Create metadata for channel BTs.

_available_new_datasets(*handled_vars*)
Metadata for available variables other than BT.

_count_channel_repeat_number()
Count channel/polarization pair repetition.

_fill_data(*data_arr*, *fill_value*, *scale_factor*, *add_offset*)
Fill missing data with NaN.

property _get_coeff_filenames
Retrieve necessary files for coefficients if needed.

_get_ds_info_for_data_arr(*var_name*)

property _get_platform_name
Get platform name.

property _get_sensor
Get sensor.

_is_2d_yx_data_array(*data_arr*)

static _nan_for_dtype(*data_arr_dtype*)

static _scale_data(*data_arr*, *scale_factor*, *add_offset*)
Scale data, if needed.

apply_attributes(*data*, *ds_info*)
Combine attributes from file and yaml and apply.

File attributes should take precedence over yaml if both are present

available_datasets(*configured_datasets=None*)
Dynamically discover what variables can be loaded from this file.

See `satpy.readers.file_handlers.BaseHandler.available_datasets()` for more information.

property end_time
Get end time.

force_date(*key*)
Force datetime.date for combine.

force_time(*key*)
Force datetime.time for combine.

get_dataset(*ds_id*, *ds_info*)
Get datasets.

property platform_shortname
Get platform shortname.

property sensor_names
Return standard sensor names for the file's data.

property start_time

Get start time.

update_metadata(*ds_info*)

Get metadata.

satpy.readers.mirs.**apply_atms_limb_correction**(*datasets*, *channel_idx*, *dmean*, *coeffs*, *amean*, *nchx*, *nchanx*)

Calculate the correction for each channel.

satpy.readers.mirs.**get_coeff_by_sfc**(*coeff_fn*, *bt_data*, *idx*)

Read coefficients for specific filename (land or sea).

satpy.readers.mirs.**limb_correct_atms_bt**(*bt_data*, *surf_type_mask*, *coeff_fns*, *ds_info*)

Gather data needed for limb correction.

satpy.readers.mirs.**read_atms_coeff_to_string**(*fn*)

Read the coefficients into a string.

satpy.readers.mirs.**read_atms_limb_correction_coefficients**(*fn*)

Read the limb correction files.

satpy.readers.modis_l1b module

Modis level 1b hdf-eos format reader.

Introduction

The `modis_l1b` reader reads and calibrates Modis L1 image data in hdf-eos format. Files often have a pattern similar to the following one:

```
M[O/Y]D02[1/H/Q]KM.A[date].[time].[collection].[processing_time].hdf
```

Other patterns where "collection" and/or "proccessing_time" are missing might also work (see the readers yaml file for details). Geolocation files (MOD03) are also supported. The IMAPP direct broadcast naming format is also supported with names like: `a1.12226.1846.1000m.hdf`.

Saturation Handling

Band 2 of the MODIS sensor is available in 250m, 500m, and 1km resolutions. The band data may include a special fill value to indicate when the detector was saturated in the 250m version of the data. When the data is aggregated to coarser resolutions this saturation fill value is converted to a "can't aggregate" fill value. By default, Satpy will replace these fill values with NaN to indicate they are invalid. This is typically undesired when generating images for the data as they appear as "holes" in bright clouds. To control this the keyword argument `mask_saturated` can be passed and set to `False` to set these two fill values to the maximum valid value.

```
scene = satpy.Scene(filenames=filenames,
                    reader='modis_l1b',
                    reader_kwargs={'mask_saturated': False})
scene.load(['2'])
```

Note that the saturation fill value can appear in other bands (ex. bands 7-19) in addition to band 2. Also, the "can't aggregate" fill value is a generic "catch all" for any problems encountered when aggregating high resolution bands to lower resolutions. Filling this with the max valid value could replace non-saturated invalid pixels with valid values.

Geolocation files

For the 1km data (mod021km) geolocation files (mod03) are optional. If not given to the reader 1km geolocations will be interpolated from the 5km geolocation contained within the file.

For the 500m and 250m data geolocation files are needed.

References

- Modis gelocation description: http://www.icare.univ-lille1.fr/wiki/index.php/MODIS_geolocation

class satpy.readers.modis_l1b.**HDFEOSBandReader**(*filename*, *filename_info*, *filetype_info*, *mask_saturated=True*, ***kwargs*)

Bases: *HDFEOSBaseFileReader*

Handler for the regular band channels.

Init the file handler.

_calibrate_data(*key*, *info*, *array*, *var_attrs*, *index*)

_fill_saturated(*array*, *valid_max*)

Replace saturation-related values with max reflectance.

If the file handler was created with `mask_saturated` set to `True` then all invalid/fill values are set to NaN. If `False` then the fill values 65528 and 65533 are set to the maximum valid value. These values correspond to "can't aggregate" and "saturation".

Fill values:

- 65535 Fill Value (includes reflective band data at night mode and completely missing L1A scans)
- 65534 L1A DN is missing within a scan
- 65533 Detector is saturated
- 65532 Cannot compute zero point DN, e.g., SV is saturated
- 65531 Detector is dead (see comments below)
- 65530 RSB dn** below the minimum of the scaling range
- 65529 TEB radiance or RSB dn exceeds the maximum of the scaling range
- 65528 Aggregation algorithm failure
- 65527 Rotation of Earth view Sector from nominal science collection position
- 65526 Calibration coefficient b1 could not be computed
- 65525 Subframe is dead
- 65524 Both sides of the PCLW electronics on simultaneously
- 65501 - 65523 (reserved for future use)
- 65500 NAD closed upper limit

_get_band_index(*var_attrs*, *band_name*)

Get the relative indices of the desired channel.

_get_band_variable_name_and_index(*band_name*)

_mask_invalid(*array*, *valid_min*, *valid_max*)

Replace fill values with NaN.

_mask_uncertain_pixels(*array*, *uncertainty*, *band_index*)

get_dataset(*key*, *info*)

Read data from file and return the corresponding projectables.

res = {'1': 1000, 'H': 500, 'Q': 250}

res_to_possible_variable_names = {250: ['EV_250_RefSB'], 500: ['EV_250_Aggr500_RefSB', 'EV_500_RefSB'], 1000: ['EV_250_Aggr1km_RefSB', 'EV_500_Aggr1km_RefSB', 'EV_1KM_RefSB', 'EV_1KM_Emissive']}

class satpy.readers.modis_l1b.**MixedHDFEOSReader**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDFEOSGeoReader*, *HDFEOSBandReader*

A file handler for the files that have both regular bands and geographical information in them.

Init the file handler.

get_dataset(*key*, *info*)

Get the dataset.

satpy.readers.modis_l1b.**calibrate_bt**(*array*, *attributes*, *index*, *band_name*)

Calibration for the emissive channels.

satpy.readers.modis_l1b.**calibrate_counts**(*array*, *attributes*, *index*)

Calibration for counts channels.

satpy.readers.modis_l1b.**calibrate_radiance**(*array*, *attributes*, *index*)

Calibration for radiance channels.

satpy.readers.modis_l1b.**calibrate_refl**(*array*, *attributes*, *index*)

Calibration for reflective channels.

satpy.readers.modis_l2 module

Modis level 2 hdf-eos format reader.

Introduction

The `modis_l2` reader reads and calibrates Modis L2 image data in hdf-eos format. Since there are a multitude of different level 2 datasets not all of theses are implemented (yet).

Currently the reader supports:

- m[o/y]d35_l2: cloud_mask dataset
- some datasets in m[o/y]d06 files

To get a list of the available datasets for a given file refer to the "Load data" section in *Reading*.

Geolocation files

Similar to the `modis_l1b` reader the geolocation files (mod03) for the 1km data are optional and if not given 1km geolocations will be interpolated from the 5km geolocation contained within the file.

For the 500m and 250m data geolocation files are needed.

References

- Documentation about the format: https://modis-atmos.gsfc.nasa.gov/products

class satpy.readers.modis_l2.**ModisL2HDFFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDFEOSGeoReader*

File handler for MODIS HDF-EOS Level 2 files.

Includes error handling for files produced by IMAPP produced files.

Initialize the geographical reader.

_extract_and_mask_category_dataset(*dataset_id*, *dataset_info*, *var_name*)

_load_all_metadata_attributes()

_mask_with_quality_assurance_if_needed(*dataset*, *dataset_info*, *dataset_id*)

_select_hdf_dataset(*hdf_dataset_name*, *byte_dimension*)

Load a dataset from HDF-EOS level 2 file.

property end_time

Get the end time of the dataset.

get_dataset(*dataset_id*, *dataset_info*)

Get DataArray for specified dataset.

property is_imapp_mask_byte1

Get if this file is the IMAPP 'mask_byte1' file type.

static read_geo_resolution(*metadata*)

Parse metadata to find the geolocation resolution.

It is implemented as a staticmethod to match read_mda pattern.

property start_time

Get the start time of the dataset.

satpy.readers.modis_l2.**_bits_strip**(*bit_start*, *bit_count*, *value*)

Extract specified bit from bit representation of integer value.

Parameters

- **bit_start** (*int*) – Starting index of the bits to extract (first bit has index 0)
- **bit_count** (*int*) – Number of bits starting from bit_start to extract
- **value** (*int*) – Number from which to extract the bits
- **Returns** –
- **-------** – int Value of the extracted bits

satpy.readers.modis_l2.**_extract_byte_mask**(*dataset*, *byte_information*, *bit_start*, *bit_count*)

satpy.readers.modis_l2.**_extract_two_byte_mask**(*data_a: ndarray*, *data_b: ndarray*, *bit_start: int*, *bit_count: int*) → ndarray

satpy.readers.modis_l3 module

Modis level 3 hdf-eos format reader.

Introduction

The `modis_l3` reader reads MODIS L3 products in HDF-EOS format.

There are multiple level 3 products, including some on sinusoidal grids and some on the climate modeling grid (CMG). This reader supports the CMG products at present, and the sinusoidal products will be added if there is demand.

The reader has been tested with:

- MCD43c*: BRDF/Albedo data, such as parameters, albedo and nbar
- MOD09CMG: Surface Reflectance on climate monitoring grid.

To get a list of the available datasets for a given file refer to the "Load data" section in *Reading*.

class satpy.readers.modis_l3.**ModisL3GriddedHDFFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDFEOSGeoReader*

File handler for MODIS HDF-EOS Level 3 CMG gridded files.

Initialize the geographical reader.

_dynamic_variables_from_file(*handled_var_names: set*) → Iterable[tuple[bool, dict]]

_get_area_extent()

Get the grid properties.

_get_res()

Compute the resolution from the file metadata.

available_datasets(*configured_datasets=None*)

Automatically determine datasets provided by this file.

get_area_def(*dsid*)

Get the area definition.

This is fixed, but not defined in the file. So we must generate it ourselves with some assumptions.

get_dataset(*dataset_id*, *dataset_info*)

Get DataArray for specified dataset.

satpy.readers.msi_safe module

SAFE MSI L1C reader.

The MSI data has a special value for saturated pixels. By default, these pixels are set to np.inf, but for some applications it might be desirable to have these pixels left untouched. For this case, the *mask_saturated* flag is available in the reader, and can be toggled with `reader_kwargs` upon Scene creation:

```
scene = satpy.Scene(filenames,
                    reader='msi_safe',
                    reader_kwargs={'mask_saturated': False})
scene.load(['B01'])
```

L1B format description for the files read here:

https://sentinels.copernicus.eu/documents/247904/0/Sentinel-2-product-specifications-document-V14-9.pdf/

class satpy.readers.msi_safe.**SAFEMSIL1C**(*filename*, *filename_info*, *filetype_info*, *mda*, *tile_mda*, *mask_saturated=True*)

Bases: *BaseFileHandler*

File handler for SAFE MSI files (jp2).

Initialize the reader.

_read_from_file(*key*)

property end_time

Get the end time.

get_area_def(*dsid*)

Get the area def.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Get the start time.

class satpy.readers.msi_safe.**SAFEMSIMDXML**(*filename*, *filename_info*, *filetype_info*, *mask_saturated=True*)

Bases: *SAFEMSIXMLMetadata*

File handle for sentinel 2 safe XML generic metadata.

Init the reader.

_band_index(*band*)

_sanitize_data(*data*)

property band_indices

Get the band indices from the metadata.

band_offset(*band*)

Get the band offset for *band*.

property band_offsets

Get the band offsets from the metadata.

calibrate_to_radiances(*data*, *band_name*)

Calibrate *data* to radiance using the radiometric information for the metadata.

calibrate_to_reflectances(*data*, *band_name*)

Calibrate *data* using the radiometric information for the metadata.

property no_data

Get the nodata value from the metadata.

physical_gain(*band_name*)

Get the physical gain for a given *band_name*.

property physical_gains

Get the physical gains dictionary.

property saturated

Get the saturated value from the metadata.

property special_values

Get the special values from the metadata.

class satpy.readers.msi_safe.**SAFEMSITileMDXML**(*filename*, *filename_info*, *filetype_info*, *mask_saturated=True*)

Bases: *SAFEMSIXMLMetadata*

File handle for sentinel 2 safe XML tile metadata.

Init the reader.

_area_extent(*resolution*)

static _do_interp(*minterp*, *xcoord*, *ycoord*)

_get_coarse_dataset(*key*, *info*)

Get the coarse dataset refered to by *key* from the XML data.

_get_satellite_angles(*angles*, *info*)

_get_solar_angles(*angles*, *info*)

static _get_values_from_tag(*xml_tree*, *xml_tag*)

_shape(*resolution*)

get_area_def(*dsid*)

Get the area definition of the dataset.

get_dataset(*key*, *info*)

Get the dataset referred to by *key*.

interpolate_angles(*angles*, *resolution*)

Interpolate the angles.

property projection

Get the geographic projection.

class satpy.readers.msi_safe.**SAFEMSIXMLMetadata**(*filename*, *filename_info*, *filetype_info*, *mask_saturated=True*)

Bases: *BaseFileHandler*

Base class for SAFE MSI XML metadata filehandlers.

Init the reader.

property end_time

Get end time.

property start_time

Get start time.

satpy.readers.msi_safe.**_fill_swath_edges**(*angles*)

Fill gaps at edges of swath.

satpy.readers.msu_gsa_l1b module

Reader for the Arctica-M1 MSU-GS/A data.

The files for this reader are HDF5 and contain channel data at 1km resolution for the VIS channels and 4km resolution for the IR channels. Geolocation data is available at both resolutions, as is sun and satellite geometry.

This reader was tested on sample data provided by EUMETSAT.

class satpy.readers.msu_gsa_l1b.**MSUGSAFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

MSU-GS/A L1B file reader.

Initialize file handler.

static _apply_scale_offset(*in_data*)

Apply the scale and offset to data.

get_dataset(*dataset_id*, *ds_info*)

Load data variable and metadata and calibrate if needed.

property platform_name

Platform name is also hardcoded.

property satellite_altitude

Satellite altitude at time of scan.

There is no documentation but this appears to be height above surface in meters.

property satellite_latitude

Satellite latitude at time of scan.

property satellite_longitude

Satellite longitude at time of scan.

property sensor_name

Sensor name is hardcoded.

property start_time

Time for timeslot scan start.

satpy.readers.mviri_l1b_fiduceo_nc module

FIDUCEO MVIRI FCDR Reader.

Introduction

The FIDUCEO MVIRI FCDR is a Fundamental Climate Data Record (FCDR) of re-calibrated Level 1.5 Infrared, Water Vapour, and Visible radiances from the Meteosat Visible Infra-Red Imager (MVIRI) instrument onboard the Meteosat First Generation satellites. There are two variants of the dataset: The *full FCDR* and a simplified version called *easy FCDR*. Some datasets are only available in one of the two variants, see the corresponding YAML definition in `satpy/etc/readers/`.

Dataset Names

The FIDUCEO MVIRI readers use names `VIS`, `WV` and `IR` for the visible, water vapor and infrared channels, respectively. These are different from the original netCDF variable names for the following reasons:

- VIS channel is named differently in full FCDR (`counts_vis`) and easy FCDR (`toa_bidirectional_reflectance_vis`)
- netCDF variable names contain the calibration level (e.g. `counts_...`), which might be confusing for satpy users if a different calibration level is chosen.

Remaining datasets (such as quality flags and uncertainties) have the same name in the reader as in the netCDF file.

Example:

This is how to read FIDUCEO MVIRI FCDR data in satpy:

```
from satpy import Scene

scn = Scene(filenames=['FIDUCEO_FCDR_L15_MVIRI_MET7-57.0...'],
            reader='mviri_l1b_fiduceo_nc')
scn.load(['VIS', 'WV', 'IR'])
```

Global netCDF attributes are available in the `raw_metadata` attribute of each loaded dataset.

Image Orientation

The images are stored in MVIRI scanning direction, that means South is up and East is right. This can be changed as follows:

```
scn.load(['VIS'], upper_right_corner='NE')
```

Geolocation

In addition to the image data, FIDUCEO also provides so called *static FCDRs* containing latitude and longitude coordinates. In order to simplify their usage, the FIDUCEO MVIRI readers do not make use of these static files, but instead provide an area definition that can be used to compute longitude and latitude coordinates on demand.

```
area = scn['VIS'].attrs['area']
lons, lats = area.get_lonlats()
```

Those were compared to the static FCDR and they agree very well, however there are small differences. The mean difference is < 1E3 degrees for all channels and projection longitudes.

Huge VIS Reflectances

You might encounter huge VIS reflectances (10^8 percent and greater) in situations where both radiance and solar zenith angle are small. The reader certainly needs some improvement in this regard. Maybe the corresponding uncertainties can be used to filter these cases before calculating reflectances.

VIS Channel Quality Flags

Quality flags are available for the VIS channel only. A simple approach for masking bad quality pixels is to set the `mask_bad_quality` keyword argument to `True`:

```
scn = Scene(filenames=['FIDUCEO_FCDR_L15_MVIRI_MET7-57.0...'],
            reader='mviri_l1b_fiduceo_nc',
            reader_kwargs={'mask_bad_quality': True})
```

See *`FiduceoMviriBase`* for an argument description. In some situations however the entire image can be flagged (look out for warnings). In that case check out the `quality_pixel_bitmask` and `data_quality_bitmask` datasets to find out why.

Angles

The FIDUCEO MVIRI FCDR provides satellite and solar angles on a coarse tiepoint grid. By default these datasets will be interpolated to the higher VIS resolution. This can be changed as follows:

```
scn.load(['solar_zenith_angle'], resolution=4500)
```

If you need the angles in both resolutions, use data queries:

```
from satpy import DataQuery

query_vis = DataQuery(
    name='solar_zenith_angle',
    resolution=2250
)
query_ir = DataQuery(
    name='solar_zenith_angle',
    resolution=4500
)
scn.load([query_vis, query_ir])
```

(continues on next page)

(continued from previous page)

```
# Use the query objects to access the datasets as follows
sza_vis = scn[query_vis]
```

References:

- [Handbook] MFG User Handbook
- [PUG] FIDUCEO MVIRI FCDR Product User Guide

`satpy.readers.mviri_l1b_fiduceo_nc.ALTITUDE = 35785860.0`

[Handbook] section 5.2.1.

class `satpy.readers.mviri_l1b_fiduceo_nc.DatasetWrapper(nc)`

Bases: `object`

Helper class for accessing the dataset.

Wrap the given dataset.

`_cleanup_attrs(ds)`

Cleanup dataset attributes.

`_coordinates_not_assigned(ds)`

`_reassign_coords(ds)`

Re-assign coordinates.

For some reason xarray doesn't assign coordinates to all high resolution data variables.

`_rename_dims(ds)`

Rename dataset dimensions to match satpy's expectations.

`_should_dims_be_renamed(ds)`

Determine whether dataset dimensions need to be renamed.

property attrs

Exposes dataset attributes.

`get_image_size(resolution)`

Get image size for the given resolution.

`get_time()`

Get time coordinate.

Variable is sometimes named "time" and sometimes "time_ir_wv".

`get_xy_coords(resolution)`

Get x and y coordinates for the given resolution.

class `satpy.readers.mviri_l1b_fiduceo_nc.FiduceoMviriBase(filename, filename_info, filetype_info, mask_bad_quality=False)`

Bases: `BaseFileHandler`

Baseclass for FIDUCEO MVIRI file handlers.

Initialize the file handler.

Parameters

mask_bad_quality – Mask VIS pixels with bad quality, that means any quality flag except "ok". If you need more control, use the `quality_pixel_bitmask` and `data_quality_bitmask` datasets.

_calibrate(*ds*, *channel*, *calibration*)

Calibrate the given dataset.

abstract _calibrate_vis(*ds*, *channel*, *calibration*)

Calibrate VIS channel. To be implemented by subclasses.

_cleanup_coords(*ds*)

Cleanup dataset coordinates.

Y/x coordinates have been useful for interpolation so far, but they only contain row/column numbers. Drop these coordinates so that Satpy can assign projection coordinates upstream (based on the area definition).

_get_acq_time_uncached(*resolution*)

Get scanline acquisition time for the given resolution.

Note that the acquisition time does not increase monotonically with the scanline number due to the scan pattern and rectification.

_get_angles_uncached(*name*, *resolution*)

Get angle dataset.

Files provide angles (solar/satellite zenith & azimuth) at a coarser resolution. Interpolate them to the desired resolution.

_get_calib_coefs()

Get calibration coefficients for all channels.

Note: Only coefficients present in both file types.

_get_channel(*name*, *resolution*, *calibration*)

Get and calibrate channel data.

_get_orbital_parameters()

Get the orbital parameters.

_get_other_dataset(*name*)

Get other datasets such as uncertainties.

_get_ssp(*coord*)

_get_ssp_lonlat()

Get longitude and latitude at the subsatellite point.

Easy FCDR files provide satellite position at the beginning and end of the scan. This method computes the mean of those two values. In the full FCDR the information seems to be missing.

Returns

Subsatellite longitude and latitude

_update_attrs(*ds*, *info*)

Update dataset attributes.

get_area_def(*dataset_id*)

Get area definition of the given dataset.

get_dataset(*dataset_id*, *dataset_info*)

Get the dataset.

nc_keys = {'IR': 'count_ir', 'WV': 'count_wv'}

class satpy.readers.mviri_l1b_fiduceo_nc.**FiduceoMviriEasyFcdrFileHandler**(*filename*, *filename_info*, *filetype_info*, *mask_bad_quality=False*)

Bases: *FiduceoMviriBase*

File handler for FIDUCEO MVIRI Easy FCDR.

Initialize the file handler.

Parameters

mask_bad_quality – Mask VIS pixels with bad quality, that means any quality flag except "ok". If you need more control, use the `quality_pixel_bitmask` and `data_quality_bitmask` datasets.

_calibrate_vis(*ds*, *channel*, *calibration*)

Calibrate VIS channel.

Easy FCDR provides reflectance only, no counts or radiance.

nc_keys = {'IR': 'count_ir', 'VIS': 'toa_bidirectional_reflectance_vis', 'WV': 'count_wv'}

class satpy.readers.mviri_l1b_fiduceo_nc.**FiduceoMviriFullFcdrFileHandler**(*filename*, *filename_info*, *filetype_info*, *mask_bad_quality=False*)

Bases: *FiduceoMviriBase*

File handler for FIDUCEO MVIRI Full FCDR.

Initialize the file handler.

Parameters

mask_bad_quality – Mask VIS pixels with bad quality, that means any quality flag except "ok". If you need more control, use the `quality_pixel_bitmask` and `data_quality_bitmask` datasets.

_calibrate_vis(*ds*, *channel*, *calibration*)

Calibrate VIS channel.

_get_calib_coefs()

Add additional VIS coefficients only present in full FCDR.

nc_keys = {'IR': 'count_ir', 'VIS': 'count_vis', 'WV': 'count_wv'}

class satpy.readers.mviri_l1b_fiduceo_nc.**IRWVCalibrator**(*coefs*)

Bases: `object`

Calibrate IR & WV channels.

Initialize the calibrator.

Parameters

coefs – Calibration coefficients.

_calibrate_rad_bt(*counts*, *calibration*)

Calibrate counts to radiance or brightness temperature.

_counts_to_radiance(*counts*)

Convert IR/WV counts to radiance.

Reference: [PUG], equations (4.1) and (4.2).

_radiance_to_brightness_temperature(*rad*)

Convert IR/WV radiance to brightness temperature.

Reference: [PUG], equations (5.1) and (5.2).

calibrate(*counts*, *calibration*)

Calibrate IR/WV counts to the given calibration.

class satpy.readers.mviri_l1b_fiduceo_nc.**Interpolator**

Bases: object

Interpolate datasets to another resolution.

static interp_acq_time(*time2d*, *target_y*)

Interpolate scanline acquisition time to the given coordinates.

The files provide timestamps per pixel for the low resolution channels (IR/WV) only.

1) Average values in each line to obtain one timestamp per line.
2) For the VIS channel duplicate values in y-direction (as advised by [PUG]).

Note that the timestamps do not increase monotonically with the line number in some cases.

Returns
Mean scanline acquisition timestamps

static interp_tiepoints(*ds*, *target_x*, *target_y*)

Interpolate dataset between tiepoints.

Uses linear interpolation.

FUTURE: [PUG] recommends cubic spline interpolation.

Parameters

- **ds** – Dataset to be interpolated
- **target_x** – Target x coordinates
- **target_y** – Target y coordinates

satpy.readers.mviri_l1b_fiduceo_nc.**MVIRI_FIELD_OF_VIEW = 18.0**

[Handbook] section 5.3.2.1.

class satpy.readers.mviri_l1b_fiduceo_nc.**Navigator**

Bases: object

Navigate MVIRI images.

_get_factors_offsets(*im_size*)

Determine line/column offsets and scaling factors.

_get_proj_params(*im_size*, *projection_longitude*)

Get projection parameters for the given settings.

get_area_def(*im_size*, *projection_longitude*)

Create MVIRI area definition.

class satpy.readers.mviri_l1b_fiduceo_nc.**VISCalibrator**(*coefs*, *solar_zenith_angle=None*)

Bases: object

Calibrate VIS channel.

Initialize the calibrator.

Parameters

- **coefs** – Calibration coefficients.
- **solar_zenith_angle** (*optional*) – Solar zenith angle. Only required for calibration to reflectance.

_calibrate_rad_refl(*counts*, *calibration*)

Calibrate counts to radiance or reflectance.

_counts_to_radiance(*counts*)

Convert VIS counts to radiance.

Reference: [PUG], equations (7) and (8).

_radiance_to_reflectance(*rad*)

Convert VIS radiance to reflectance factor.

Note: Produces huge reflectances in situations where both radiance and solar zenith angle are small. Maybe the corresponding uncertainties can be used to filter these cases before calculating reflectances.

Reference: [PUG], equation (6).

calibrate(*counts*, *calibration*)

Calibrate VIS counts.

static refl_factor_to_percent(*refl*)

Convert reflectance factor to percent.

update_refl_attrs(*refl*)

Update attributes of reflectance datasets.

class satpy.readers.mviri_l1b_fiduceo_nc.**VisQualityControl**(*mask*)

Bases: object

Simple quality control for VIS channel.

Initialize the quality control.

check()

Check VIS channel quality and issue a warning if it's bad.

mask(*ds*)

Mask VIS pixels with bad quality.

Pixels are considered bad quality if the "quality_pixel_bitmask" is everything else than 0 (no flag set).

satpy.readers.mviri_l1b_fiduceo_nc.**is_high_resol**(*resolution*)

Identify high resolution channel.

satpy.readers.mws_l1b module

Reader for the EPS-SG Microwave Sounder (MWS) level-1b data.

Documentation: https://www.eumetsat.int/media/44139

class satpy.readers.mws_l1b.**MWSL1BFile**(*filename*, *filename_info*, *filetype_info*)

Bases: *NetCDF4FileHandler*

Class implementing the EPS-SG-A1 MWS L1b Filehandler.

This class implements the European Polar System Second Generation (EPS-SG) Microwave Sounder (MWS) Level-1b NetCDF reader. It is designed to be used through the `Scene` class using the `load` method with the reader `"mws_l1b_nc"`.

Initialize file handler.

static _drop_coords(*variable*)

Drop coords that are not in dims.

_get_dataset_aux_data(*dsname*)

Get the auxiliary data arrays using the index map.

_get_dataset_channel(*key*, *dataset_info*)

Load dataset corresponding to channel measurement.

Load a dataset when the key refers to a measurand, whether uncalibrated (counts) or calibrated in terms of brightness temperature or radiance.

_get_global_attributes()

Create a dictionary of global attributes.

_get_quality_attributes()

Get quality attributes.

_manage_attributes(*variable*, *dataset_info*)

Manage attributes of the dataset.

_platform_name_translate = {'SGA1': 'Metop-SG-A1', 'SGA2': 'Metop-SG-A2', 'SGA3': 'Metop-SG-A3'}

static _standardize_dims(*variable*)

Standardize dims to y, x.

property end_time

Get end time.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using file_key in dataset_info.

property platform_name

Get the platform name.

property sensor

Get the sensor name.

property start_time

Get start time.

property sub_satellite_latitude_end

Get the latitude of sub-satellite point at end of the product.

property sub_satellite_latitude_start

Get the latitude of sub-satellite point at start of the product.

property sub_satellite_longitude_end

Get the longitude of sub-satellite point at end of the product.

property sub_satellite_longitude_start

Get the longitude of sub-satellite point at start of the product.

satpy.readers.mws_l1b.**_get_aux_data_name_from_dsname**(*dsname*)

satpy.readers.mws_l1b.**get_channel_index_from_name**(*chname*)

Get the MWS channel index from the channel name.

satpy.readers.netcdf_utils module

Helpers for reading netcdf-based files.

class satpy.readers.netcdf_utils.**NetCDF4FileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: `BaseFileHandler`

Small class for inspecting a NetCDF4 file and retrieving its metadata/header data.

File information can be accessed using bracket notation. Variables are accessed by using:

wrapper["var_name"]

Or:

wrapper["group/subgroup/var_name"]

Attributes can be accessed by appending "/attr/attr_name" to the item string:

wrapper["group/subgroup/var_name/attr/units"]

Or for global attributes:

wrapper["/attr/platform_short_name"]

Or for all of global attributes:

wrapper["/attrs"]

Note that loading datasets requires reopening the original file (unless those datasets are cached, see below), but to get just the shape of the dataset append "/shape" to the item string:

wrapper["group/subgroup/var_name/shape"]

If your file has many small data variables that are frequently accessed, you may choose to cache some of them. You can do this by passing a number, any variable smaller than this number in bytes will be read into RAM. Warning, this part of the API is provisional and subject to change.

You may get an additional speedup by passing `cache_handle=True`. This will keep the netCDF4 dataset handles open throughout the lifetime of the object, and instead of using *xarray.open_dataset* to open every data variable, a dask array will be created "manually". This may be useful if you have a dataset distributed over many files,

such as for FCI. Note that the coordinates will be missing in this case. If you use this option, `xarray_kwargs` will have no effect.

Parameters

- **filename** (*str*) – File to read
- **filename_info** (*dict*) – Dictionary with filename information
- **filetype_info** (*dict*) – Dictionary with filetype information
- **auto_maskandscale** (*bool*) – Apply mask and scale factors
- **xarray_kwargs** (*dict*) – Addition arguments to *xarray.open_dataset*
- **cache_var_size** (*int*) – Cache variables smaller than this size.
- **cache_handle** (*bool*) – Keep files open for lifetime of filehandler.

Initialize object.

_collect_attrs(*name*, *obj*)

Collect all the attributes for the provided file object.

_collect_cache_var_names(*cache_var_size*)

_collect_global_attrs(*obj*)

Collect all the global attributes for the provided file object.

_collect_groups_info(*base_name*, *obj*)

_collect_listed_variables(*file_handle*, *listed_variables*)

_collect_variable_info(*var_name*, *var_obj*)

_collect_variables_info(*base_name*, *obj*)

_get_attr(*obj*, *key*)

_get_attr_value(*obj*, *key*)

_get_file_handle()

_get_group(*key*, *val*)

Get a group from the netcdf file.

_get_object_attrs(*obj*)

static _get_required_variable_names(*listed_variables*, *variable_name_replacements*)

_get_var_from_filehandle(*group*, *key*)

_get_var_from_xr(*group*, *key*)

_get_variable(*key*, *val*)

Get a variable from the netcdf file.

static _set_file_handle_auto_maskandscale(*file_handle*, *auto_maskandscale*)

_set_xarray_kwargs(*xarray_kwargs*, *auto_maskandscale*)

collect_cache_vars(*cache_var_size*)

Collect data variables for caching.

This method will collect some data variables and store them in RAM. This may be useful if some small variables are frequently accessed, to prevent needlessly frequently opening and closing the file, which in case of xarray is associated with some overhead.

Should be called later than *collect_metadata*.

Parameters
cache_var_size (*int*) – Maximum size of the collected variables in bytes

collect_dimensions(*name*, *obj*)

Collect dimensions.

collect_metadata(*name*, *obj*)

Collect all file variables and attributes for the provided file object.

This method also iterates through subgroups of the provided object.

file_handle = None

get(*item*, *default=None*)

Get item.

get_and_cache_npxr(*var_name*)

Get and cache variable as DataArray[numpy].

class satpy.readers.netcdf_utils.**NetCDF4FsspecFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

NetCDF4 file handler using fsspec to read files remotely.

Initialize object.

_collect_cache_var_names(*cache_var_size*)

_collect_cache_var_names_h5netcdf(*cache_var_size*)

_get_attr(*obj*, *key*)

_get_file_handle()

_get_object_attrs(*obj*)

_getitem_h5netcdf(*key*)

satpy.readers.netcdf_utils.**_compose_replacement_names**(*variable_name_replacements*, *var*, *variable_names*)

satpy.readers.nucaps module

Interface to NUCAPS Retrieval NetCDF files.

NUCAPS stands for NOAA Unique Combined Atmospheric Processing System. NUCAPS retrievals include temperature, moisture, trace gas, and cloud-cleared radiance profiles. Product details can be found at:

https://www.ospo.noaa.gov/Products/atmosphere/soundings/nucaps/

This reader supports both standard NOAA NUCAPS EDRs, and Science EDRs, which are essentially a subset of the standard EDRs with some additional parameters such as relative humidity and boundary layer temperature.

NUCAPS data is derived from Cross-track Infrared Sounder (CrIS) data, and from Advanced Technology Microwave Sounder (ATMS) data, instruments onboard Joint Polar Satellite System spacecraft.

class satpy.readers.nucaps.**NUCAPSFileHandler**(**args*, ***kwargs*)

Bases: *NetCDF4FileHandler*

File handler for NUCAPS netCDF4 format.

Initialize file handler.

_parse_datetime(*datestr*)

Parse NUCAPS datetime string.

property end_orbit_number

Return orbit number for the end of the swath.

property end_time

Get end time.

get_dataset(*dataset_id*, *ds_info*)

Load data array and metadata for specified dataset.

get_metadata(*dataset_id*, *ds_info*)

Get metadata.

get_shape(*ds_id*, *ds_info*)

Return data array shape for item specified.

property platform_name

Return standard platform name for the file's data.

property sensor_names

Return standard sensor or instrument name for the file's data.

property start_orbit_number

Return orbit number for the beginning of the swath.

property start_time

Get start time.

class satpy.readers.nucaps.**NUCAPSReader**(*config_files*, *mask_surface=True*, *mask_quality=True*, ***kwargs*)

Bases: *FileYAMLReader*

Reader for NUCAPS NetCDF4 files.

Configure reader behavior.

Parameters

- **mask_surface** (*boolean*) – mask anything below the surface pressure
- **mask_quality** (*boolean*) – mask anything where the *Quality_Flag* metadata is `!= 1`.

_abc_impl = <_abc._abc_data object>

_filter_dataset_keys_outside_pressure_levels(*dataset_keys*, *pressure_levels*)

load(*dataset_keys*, *previous_datasets=None*, *pressure_levels=None*)

Load data from one or more set of files.

Parameters

pressure_levels – mask out certain pressure levels: True for all levels (min, max) for a range of pressure levels [...] list of levels to include

load_ds_ids_from_config()

Convert config dataset entries to DataIDs.

Special handling is done to provide level specific datasets for any pressured based datasets. For example, a dataset is added for each pressure level of 'Temperature' with each new dataset being named 'Temperature_Xmb' where X is the pressure level.

satpy.readers.nucaps.**_get_pressure_level_condition**(*plevels_ds*, *pressure_levels*)

satpy.readers.nucaps.**_mask_data_below_surface_pressure**(*datasets_loaded*, *dataset_keys*)

satpy.readers.nucaps.**_mask_data_with_quality_flag**(*datasets_loaded*, *dataset_keys*)

satpy.readers.nucaps.**_remove_data_at_pressure_levels**(*datasets_loaded*, *plevels_ds*, *pressure_levels*)

satpy.readers.nwcsaf_msg2013_hdf5 module

Reader for the old NWCSAF/Geo (v2013 and earlier) cloud product format.

References

- The NWCSAF GEO 2013 products documentation: http://www.nwcsaf.org/web/guest/archive - Search for Code "ICD/3"; Type "MSG" and the box to the right should say 'Status' (which means any status). Version 7.0 seems to be for v2013

  http://www.nwcsaf.org/aemetRest/downloadAttachment/2623

class satpy.readers.nwcsaf_msg2013_hdf5.**Hdf5NWCSAF**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

NWCSAF MSG hdf5 reader.

Init method.

get_area_def(*dsid*)

Get the area definition of the datasets in the file.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset.

property start_time

Return the start time of the object.

satpy.readers.nwcsaf_msg2013_hdf5.**get_area_extent**(*cfac*, *lfac*, *coff*, *loff*, *numcols*, *numlines*)

Get the area extent from msg parameters.

satpy.readers.nwcsaf_nc module

Nowcasting SAF common PPS&MSG NetCDF/CF format reader.

References

- The NWCSAF GEO 2018 products documentation: http://www.nwcsaf.org/web/guest/archive

class satpy.readers.nwcsaf_nc.**NcNWCSAF**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

NWCSAF PPS&MSG NetCDF reader.

Init method.

_adjust_variable_for_legacy_software(*variable*)

static _ensure_crs_extents_in_meters(*crs*, *area_extent*)

Fix units in Earth shape, satellite altitude and 'units' attribute.

_get_filekeys(*dsid_name*, *info*)

_get_projection()

Get projection from the NetCDF4 attributes.

_get_varname_in_file(*info*, *info_type='file_key'*)

static _mask_variable(*variable*)

_prepare_variable_for_palette(*variable*, *info*)

_upsample_geolocation_uncached()

Upsample the geolocation (lon,lat) from the tiepoint grid.

drop_xycoords(*variable*)

Drop x, y coords when y is scan line number.

property end_time

Return the end time of the object.

get_area_def(*dsid*)

Get the area definition of the datasets in the file.

Only applicable for MSG products!

get_dataset(*dsid*, *info*)

Load a dataset.

get_orbital_parameters(*variable*)

Get the orbital parameters from the file if possible (geo).

remove_timedim(*var*)

Remove time dimension from dataset.

scale_dataset(*variable*, *info*)

Scale the data set, applying the attributes from the netCDF file.

The scale and offset attributes will then be removed from the resulting variable.

property sensor_names

List of sensors represented in this file.

set_platform_and_sensor(***kwargs*)

Set some metadata: platform_name, sensors, and pps (identifying PPS or Geo).

property start_time

Return the start time of the object.

satpy.readers.nwcsaf_nc.**read_nwcsaf_time**(*time_value*)

Read the time, nwcsaf-style.

satpy.readers.nwcsaf_nc.**remove_empties**(*variable*)

Remove empty objects from the *variable*'s attrs.

satpy.readers.oceancolorcci_l3_nc module

Reader for files produced by ESA's Ocean Color CCI project.

This reader currently supports the lat/lon gridded products and does not yet support the products on a sinusoidal grid. The products on each of the composite periods (1, 5 and 8 day plus monthly) are supported and both the merged product files (OC_PRODUCTS) and single product (RRS, CHLOR_A, IOP, K_490) are supported.

class satpy.readers.oceancolorcci_l3_nc.**OCCCIFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

File handler for Ocean Color CCI netCDF files.

Initialize object.

static _parse_datetime(*datestr*)

Parse datetime.

_update_attrs(*dataset*, *dataset_info*)

Update dataset attributes.

property composite_period

Determine composite period from filename information.

property end_time

Get the end time.

get_area_def(*dsid*)

Get the area definition based on information in file.

There is no area definition in the file itself, so we have to compute it from the metadata, which specifies the area extent and pixel resolution.

get_dataset(*dataset_id*, *ds_info*)

Get dataset.

property start_time

Get the start time.

satpy.readers.olci_nc module

Sentinel-3 OLCI reader.

This reader supports an optional argument to choose the 'engine' for reading OLCI netCDF4 files. By default, this reader uses the default xarray choice of engine, as defined in the `xarray.open_dataset()` documentation`.

As an alternative, the user may wish to use the 'h5netcdf' engine, but that is not default as it typically prints many non-fatal but confusing error messages to the terminal. To choose between engines the user can do as follows for the default:

```
scn = Scene(filenames=my_files, reader='olci_l1b')
```

or as follows for the h5netcdf engine:

```
scn = Scene(filenames=my_files,
            reader='olci_l1b', reader_kwargs={'engine': 'h5netcdf'})
```

References

- `xarray.open_dataset()`

class satpy.readers.olci_nc.**BitFlags**(*value*, *flag_list=None*)

Bases: `object`

Manipulate flags stored bitwise.

Init the flags.

class satpy.readers.olci_nc.**NCOLCI1B**(*filename*, *filename_info*, *filetype_info*, *cal*, *engine=None*)

Bases: *NCOLCIChannelBase*

File handler for OLCI l1b.

Init the file handler.

static _get_items(*idx*, *solar_flux*)

Get items.

_get_solar_flux(*band*)

Get the solar flux for the band.

get_dataset(*key*, *info*)

Load a dataset.

class satpy.readers.olci_nc.**NCOLCI2**(*filename*, *filename_info*, *filetype_info*, *engine=None*, *unlog=False*, *mask_items=None*)

Bases: *NCOLCIChannelBase*

File handler for OLCI l2.

Init the file handler.

delog(*data_array*)

Remove log10 from the units and values.

get_dataset(*key*, *info*)

Load a dataset.

getbitmask(*wqsf*, *items=None*)

Get the bitmask.

class satpy.readers.olci_nc.**NCOLCIAngles**(*filename*, *filename_info*, *filetype_info*, *engine=None*, ***kwargs*)

Bases: *NCOLCILowResData*

File handler for the OLCI angles.

Init the file handler.

_interpolate_angles(*azi*, *zen*)

datasets = {'satellite_azimuth_angle': 'OAA', 'satellite_zenith_angle': 'OZA', 'solar_azimuth_angle': 'SAA', 'solar_zenith_angle': 'SZA'}

get_dataset(*key*, *info*)

Load a dataset.

property satellite_angles

Return the satellite angles.

property sun_angles

Return the sun angles.

class satpy.readers.olci_nc.**NCOLCIBase**(*filename*, *filename_info*, *filetype_info*, *engine=None*, ***kwargs*)

Bases: *BaseFileHandler*

The OLCI reader base.

Init the olci reader base.

cols_name = 'columns'

property end_time

End time property.

get_dataset(*key*, *info*)

Load a dataset.

property nc

Get the nc xr dataset.

rows_name = 'rows'

property start_time

Start time property.

class satpy.readers.olci_nc.**NCOLCICal**(*filename*, *filename_info*, *filetype_info*, *engine=None*, ***kwargs*)

Bases: *NCOLCIBase*

Dummy class for calibration.

Init the olci reader base.

class satpy.readers.olci_nc.**NCOLCIChannelBase**(*filename*, *filename_info*, *filetype_info*, *engine=None*)

Bases: *NCOLCIBase*

Base class for channel reading.

Init the file handler.

class satpy.readers.olci_nc.**NCOLCIGeo**(*filename*, *filename_info*, *filetype_info*, *engine=None*, ***kwargs*)

Bases: *NCOLCIBase*

Dummy class for navigation.

Init the olci reader base.

class satpy.readers.olci_nc.**NCOLCILowResData**(*filename*, *filename_info*, *filetype_info*, *engine=None*, ***kwargs*)

Bases: *NCOLCIBase*

Handler for low resolution data.

Init the file handler.

_do_interpolate(*data*)

property _need_interpolation

cols_name = 'tie_columns'

rows_name = 'tie_rows'

class satpy.readers.olci_nc.**NCOLCIMeteo**(*filename*, *filename_info*, *filetype_info*, *engine=None*)

Bases: *NCOLCILowResData*

File handler for the OLCI meteo data.

Init the file handler.

datasets = ['humidity', 'sea_level_pressure', 'total_columnar_water_vapour', 'total_ozone']

get_dataset(*key*, *info*)

Load a dataset.

satpy.readers.omps_edr module

Interface to OMPS EDR format.

class satpy.readers.omps_edr.**EDREOSFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *EDRFileHandler*

EDR EOS file handler.

Initialize file handler.

_fill_name = 'MissingValue'

class satpy.readers.omps_edr.**EDRFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

EDR file handler.

Initialize file handler.

_fill_name = '_FillValue'

adjust_scaling_factors(*factors*, *file_units*, *output_units*)

Adjust scaling factors.

property end_orbit_number

Get the end orbit number.

get_dataset(*dataset_id*, *ds_info*)

Get the dataset.

get_metadata(*dataset_id*, *ds_info*)

Get the metadata.

get_shape(*ds_id*, *ds_info*)

Get the shape.

property platform_name

Get the platform name.

property sensor_name

Get the sensor name.

property start_orbit_number

Get the start orbit number.

satpy.readers.osisaf_l3_nc module

A reader for OSI-SAF level 3 products in netCDF format.

class satpy.readers.osisaf_l3_nc.**OSISAFL3NCFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

Reader for the OSISAF l3 netCDF format.

Initialize object.

_get_ds_units(*ds_info*, *var_path*)

Find the units of the datasets.

_get_ease_grid()

Set up the EASE grid.

_get_finfo_grid()

Get grid in case of filename info being used.

_get_ftype_grid()

Get grid in case of filetype info being used.

_get_geographic_grid()

Set up the EASE grid.

_get_instname()

Get instrument name.

_get_platname()

Get platform name.

_get_polar_stereographic_grid()

Set up the polar stereographic grid.

static _parse_datetime(*datestr*)

property end_time

Get the end time.

get_area_def(*area_id*)

Get the area definition, which varies depending on file type and structure.

get_dataset(*dataset_id*, *ds_info*)

Load a dataset.

property start_time

Get the start time.

satpy.readers.pmw_channels_definitions module

Passive Microwave instrument and channel specific features.

class satpy.readers.pmw_channels_definitions.**FrequencyBandBaseArithmetics**

Bases: object

Mixin class with basic frequency comparison operations.

classmethod convert(*frq*)

Convert *frq* to this type if possible.

class satpy.readers.pmw_channels_definitions.**FrequencyDoubleSideBand**(*central: float*, *side: float*, *bandwidth: float*, *unit: str = 'GHz'*)

Bases: *FrequencyBandBaseArithmetics*, *FrequencyDoubleSideBandBase*

The frequency double side band class.

The elements of the double-side-band type frequency band are the central frquency, the relative side band frequency (relative to the center - left and right) and their bandwidths, and optionally a unit (defaults to GHz). No clever unit conversion is done here, it's just used for checking that two ranges are comparable.

Frequency Double Side Band is supposed to describe the special type of bands commonly used in humidty sounding from Passive Microwave Sensors. When the absorption band being observed is symmetrical it is advantageous (giving better NeDT) to sense in a band both right and left of the central absorption frequency.

Create new instance of FrequencyDoubleSideBandBase(central, side, bandwidth, unit)

static _check_band_contains_other(*band*, *other_band*)

Check that a band contains another band.

A band is here defined as a tuple of a central frequency and a bandwidth.

distance(*value*)

Get the distance to the double side band.

Determining the distance in frequency space between two double side bands can be quite ambiguous, as such bands are in effect a set of 2 narrow bands, one on each side of the absorption line. To keep it as simple

as possible we have until further decided to set the distance between such two bands to infitiy if neither of them are contained in the other.

If the frequency entered is a single value and this frequency falls inside one of the side bands, the distance will be the minimum of the distances to the two outermost sides of the double side band. However, is such a single frequency value falls outside one of the two side bands, the distance will be set to infitiy.

If the frequency entered is a tuple the distance will either be 0 (if one is containde in the other) or infinity.

class satpy.readers.pmw_channels_definitions.**FrequencyDoubleSideBandBase**(*central: float, side: float, bandwidth: float, unit: str = 'GHz'*)

Bases: NamedTuple

Base class for a frequency double side band.

Frequency Double Side Band is supposed to describe the special type of bands commonly used in humidty sounding from Passive Microwave Sensors. When the absorption band being observed is symmetrical it is advantageous (giving better NeDT) to sense in a band both right and left of the central absorption frequency.

This is needed because of this bug: https://bugs.python.org/issue41629

Create new instance of FrequencyDoubleSideBandBase(central, side, bandwidth, unit)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {'unit': 'GHz'}

_fields = ('central', 'side', 'bandwidth', 'unit')

classmethod _make(*iterable*)

Make a new FrequencyDoubleSideBandBase object from a sequence or iterable

_replace(***kwds*)

Return a new FrequencyDoubleSideBandBase object replacing specified fields with new values

bandwidth: float

Alias for field number 2

central: float

Alias for field number 0

side: float

Alias for field number 1

unit: str

Alias for field number 3

class satpy.readers.pmw_channels_definitions.**FrequencyQuadrupleSideBand**(*central: float, side: float, sideside: float, bandwidth: float, unit: str = 'GHz'*)

Bases: *FrequencyBandBaseArithmetics*, *FrequencyQuadrupleSideBandBase*

The frequency quadruple side band class.

The elements of the quadruple-side-band type frequency band are the central frquency, the relative (main) side band frequency (relative to the center - left and right), the sub-side band frequency (relative to the offset side-band(s)) and their bandwidths. Optionally a unit (defaults to GHz) may be specified. No clever unit conversion is done here, it's just used for checking that two ranges are comparable.

Frequency Quadruple Side Band is supposed to describe the special type of bands commonly used in temperature sounding from Passive Microwave Sensors. When the absorption band being observed is symmetrical it is advantageous (giving better NeDT) to sense in a band both right and left of the central absorption frequency. But to avoid (CO2) absorption lines symmetrically positioned on each side of the main absorption band it is common to split the side bands in two 'side-side' bands.

Create new instance of FrequencyQuadrupleSideBandBase(central, side, sideside, bandwidth, unit)

distance(*value*)

Get the distance to the quadruple side band.

Determining the distance in frequency space between two quadruple side bands can be quite ambiguous, as such bands are in effect a set of 4 narrow bands, two on each side of the main absorption band, and on each side, one on each side of the secondary absorption lines. To keep it as simple as possible we have until further decided to define the distance between such two bands to infinity if they are determined to be equal.

If the frequency entered is a single value, the distance will be the minimum of the distances to the two outermost sides of the quadruple side band.

If the frequency entered is a tuple or list and the two quadruple frequency bands are contained in each other (equal) the distance will always be zero.

class satpy.readers.pmw_channels_definitions.**FrequencyQuadrupleSideBandBase**(*central: float, side: float, sideside: float, bandwidth: float, unit: str = 'GHz'*)

Bases: NamedTuple

Base class for a frequency quadruple side band.

Frequency Quadruple Side Band is supposed to describe the special type of bands commonly used in temperature sounding from Passive Microwave Sensors. When the absorption band being observed is symmetrical it is advantageous (giving better NeDT) to sense in a band both right and left of the central absorption frequency. But to avoid (CO2) absorption lines symmetrically positioned on each side of the main absorption band it is common to split the side bands in two 'side-side' bands.

This is needed because of this bug: https://bugs.python.org/issue41629

Create new instance of FrequencyQuadrupleSideBandBase(central, side, sideside, bandwidth, unit)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {'unit': 'GHz'}

_fields = ('central', 'side', 'sideside', 'bandwidth', 'unit')

classmethod _make(*iterable*)

Make a new FrequencyQuadrupleSideBandBase object from a sequence or iterable

_replace(***kwds*)

Return a new FrequencyQuadrupleSideBandBase object replacing specified fields with new values

bandwidth: *float*

Alias for field number 3

central: *float*

Alias for field number 0

side: *float*

Alias for field number 1

sideside: *float*

Alias for field number 2

unit: *str*

Alias for field number 4

class satpy.readers.pmw_channels_definitions.**FrequencyRange**(*central: float, bandwidth: float, unit: str = 'GHz'*)

Bases: *FrequencyBandBaseArithmetics*, *FrequencyRangeBase*

The Frequency range class.

The elements of the range are central and bandwidth values, and optionally a unit (defaults to GHz). No clever unit conversion is done here, it's just used for checking that two ranges are comparable.

This type is used for passive microwave sensors.

Create new instance of FrequencyRangeBase(central, bandwidth, unit)

distance(*value*)

Get the distance from value.

class satpy.readers.pmw_channels_definitions.**FrequencyRangeBase**(*central: float, bandwidth: float, unit: str = 'GHz'*)

Bases: NamedTuple

Base class for frequency ranges.

This is needed because of this bug: https://bugs.python.org/issue41629

Create new instance of FrequencyRangeBase(central, bandwidth, unit)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {'unit': 'GHz'}

_fields = ('central', 'bandwidth', 'unit')

classmethod _make(*iterable*)

Make a new FrequencyRangeBase object from a sequence or iterable

_replace(***kwds*)

Return a new FrequencyRangeBase object replacing specified fields with new values

bandwidth: *float*

Alias for field number 1

central: *float*

Alias for field number 0

unit: str

Alias for field number 2

satpy.readers.pmw_channels_definitions.**_is_inside_interval**(*value*, *central*, *width*)

satpy.readers.safe_sar_l2_ocn module

SAFE SAR L2 OCN format reader.

The OCN data contains various parameters, but mainly the wind speed and direction calculated from SAR data and input model data from ECMWF

Implemented in this reader is the OWI, Ocean Wind field.

See more at ESA webpage https://sentinel.esa.int/web/sentinel/ocean-wind-field-component

class satpy.readers.safe_sar_l2_ocn.**SAFENC**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Measurement file reader.

Init the file reader.

_get_data_channels(*key*, *info*)

property end_time

Product end_time, parsed from the measurement file name.

property fend_time

Product fend_time meaning the end time parsed from the SAFE directory.

property fstart_time

Product fstart_time meaning the start time parsed from the SAFE directory.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Product start_time, parsed from the measurement file name.

satpy.readers.sar_c_safe module

SAFE SAR-C reader.

This module implements a reader for Sentinel 1 SAR-C GRD (level1) SAFE format as provided by ESA. The format is comprised of a directory containing multiple files, most notably two measurement files in geotiff and a few xml files for calibration, noise and metadata.

References

- *Level 1 Product Formatting* https://sentinel.esa.int/web/sentinel/technical-guides/sentinel-1-sar/products-algorithms/level-1-product-formatting
- J. Park, A. A. Korosov, M. Babiker, S. Sandven and J. Won, *"Efficient Thermal Noise Removal for Sentinel-1 TOPSAR Cross-Polarization Channel,"* in IEEE Transactions on Geoscience and Remote Sensing, vol. 56, no. 3, pp. 1555-1565, March 2018. doi: 10.1109/TGRS.2017.2765248

class satpy.readers.sar_c_safe.**AzimuthNoiseReader**(*root*, *shape*)

Bases: `object`

Class to parse and read azimuth-noise data.

The azimuth noise vector is provided as a series of blocks, each comprised of a column of data to fill the block and a start and finish column number, and a start and finish line. For example, we can see here a (fake) azimuth noise array:

```
[[ 1.  1.  1. nan nan nan nan nan nan nan]
 [ 1.  1.  1. nan nan nan nan nan nan nan]
 [ 2.  2.  3.  3.  3.  4.  4.  4.  4. nan]
 [ 2.  2.  3.  3.  3.  4.  4.  4.  4. nan]
 [ 2.  2.  3.  3.  3.  4.  4.  4.  4. nan]
 [ 2.  2.  5.  5.  5.  5.  6.  6.  6.  6.]
 [ 2.  2.  5.  5.  5.  5.  6.  6.  6.  6.]
 [ 2.  2.  5.  5.  5.  5.  6.  6.  6.  6.]
 [ 2.  2.  7.  7.  7.  7.  7.  8.  8.  8.]
 [ 2.  2.  7.  7.  7.  7.  7.  8.  8.  8.]]
```

As is shown here, the blocks may not cover the full array, and hence it has to be gap-filled with NaNs.

Set up the azimuth noise reader.

_assemble_azimuth_noise_blocks(*chunks*)

Assemble the azimuth noise blocks into one single array.

_create_dask_slice_from_block_line(*current_line*, *chunks*)

Create a dask slice from the blocks at the current line.

_create_dask_slices_from_blocks(*chunks*)

Create full-width slices from azimuth noise blocks.

static _fill_dask_pieces(*dask_pieces*, *shape*, *chunks*)

_find_blocks_covering_line(*current_line*)

Find the blocks covering a given line.

_get_array_pieces_for_current_line(*current_line*)

Get the array pieces that cover the current line.

_get_next_start_line(*current_blocks*, *current_line*)

_get_padded_dask_pieces(*pieces*, *chunks*)

Get the padded pieces of a slice.

_read_azimuth_noise_blocks(*chunks*)

Read the azimuth noise blocks.

read_azimuth_noise_array(*chunks=4096*)

Read the azimuth noise vectors.

class satpy.readers.sar_c_safe.**SAFEGRD**(*filename*, *filename_info*, *filetype_info*, *calfh*, *noisefh*, *annotationfh*)

Bases: BaseFileHandler

Measurement file reader.

The measurement files are in geotiff format and read using rasterio. For performance reasons, the reading adapts the chunk size to match the file's block size.

Init the grd filehandler.

_calibrate(*dn*, *chunks*, *key*)

Calibrate the data.

_calibrate_and_denoise(*data*, *key*)

Calibrate and denoise the data.

static _change_quantity(*data*, *quantity*)

Change quantity to dB if needed.

_denoise(*dn*, *chunks*)

Denoise the data.

_get_digital_number(*data*)

Get the digital numbers (uncalibrated data).

_get_lonlatalts_uncached()

Obtain GCPs and construct latitude and longitude arrays.

Parameters

- **band** (*gdal band*) – Measurement band which comes with GCP's
- **array_shape** (*tuple*) – The size of the data array

Returns

A tuple with longitude and latitude arrays

Return type

coordinates (tuple)

property end_time

Get the end time.

get_dataset(*key*, *info*)

Load a dataset.

get_gcps()

Read GCP from the GDAL band.

Parameters

- **band** (*gdal band*) – Measurement band which comes with GCP's
- **coordinates** (*tuple*) – A tuple with longitude and latitude arrays

Returns

Pixel and Line indices 1d arrays gcp_coords (tuple): longitude and latitude 1d arrays

Return type
points (tuple)

property start_time
Get the start time.

class satpy.readers.sar_c_safe.**SAFEXML**(*filename*, *filename_info*, *filetype_info*, *header_file=None*)

Bases: *BaseFileHandler*

XML file reader for the SAFE format.

Init the xml filehandler.

property end_time
Get the end time.

get_metadata()
Convert the xml metadata to dict.

property start_time
Get the start time.

class satpy.readers.sar_c_safe.**SAFEXMLAnnotation**(*filename*, *filename_info*, *filetype_info*, *header_file=None*)

Bases: *SAFEXML*

XML file reader for the SAFE format, Annotation file.

Init the XML annotation reader.

_get_incidence_angle_uncached(*chunks*)
Get the incidence angle array.

get_dataset(*key*, *info*, *chunks=None*)
Load a dataset.

class satpy.readers.sar_c_safe.**SAFEXMLCalibration**(*filename*, *filename_info*, *filetype_info*, *header_file=None*)

Bases: *SAFEXML*

XML file reader for the SAFE format, Calibration file.

Init the XML calibration reader.

_get_calibration_uncached(*calibration*, *chunks=None*)
Get the calibration array.

_get_calibration_vector(*calibration_name*, *chunks*)
Get the calibration vector.

get_calibration_constant()
Load the calibration constant.

get_dataset(*key*, *info*, *chunks=None*)
Load a dataset.

class satpy.readers.sar_c_safe.**SAFEXMLNoise**(*filename*, *filename_info*, *filetype_info*, *header_file=None*)

Bases: *SAFEXML*

XML file reader for the SAFE format, Noise file.

Init the xml filehandler.

_get_noise_correction_uncached(*chunks=None*)

Get the noise correction array.

get_dataset(*key*, *info*, *chunks=None*)

Load a dataset.

read_legacy_noise(*chunks*)

Read noise for legacy GRD data.

read_range_noise_array(*chunks*)

Read the range-noise array.

class satpy.readers.sar_c_safe.**XMLArray**(*root*, *list_tag*, *element_tag*)

Bases: object

A proxy for getting xml data as an array.

Set up the XML array.

_read_xml_array()

Read an array from xml.

expand(*shape*, *chunks=None*)

Generate the full-blown array.

get_data_items()

Get the data items for this array.

interpolate_xml_array(*shape*, *chunks*)

Interpolate arbitrary size dataset to a full sized grid.

class satpy.readers.sar_c_safe.**_AzimuthBlock**(*xml_element*)

Bases: object

Implementation of an single azimuth-noise block.

Set up the block from an XML element.

expand(*chunks*)

Build an azimuth block from xml data.

property first_line

property first_pixel

property last_line

property last_pixel

property lines

property lut

satpy.readers.sar_c_safe.**_dictify**(*r*)

Convert an xml element to dict.

satpy.readers.sar_c_safe.**_get_calibration_name**(*calibration*)

Get the proper calibration name.

satpy.readers.sar_c_safe.**dictify**(*r*)

Convert an ElementTree into a dict.

satpy.readers.sar_c_safe.**interpolate_slice**(*slice_rows*, *slice_cols*, *interpolator*)

Interpolate the given slice of the larger array.

satpy.readers.sar_c_safe.**interpolate_xarray**(*xpoints*, *ypoints*, *values*, *shape*, *blocksize=4096*)

Interpolate, generating a dask array.

satpy.readers.sar_c_safe.**interpolate_xarray_linear**(*xpoints*, *ypoints*, *values*, *shape*, *chunks=4096*)

Interpolate linearly, generating a dask array.

satpy.readers.sar_c_safe.**intp**(*grid_x*, *grid_y*, *interpolator*)

Interpolate.

satpy.readers.satpy_cf_nc module

Reader for files produced with the cf netcdf writer in satpy.

Introduction

The satpy_cf_nc reader reads data written by the satpy cf_writer. Filenames for cf_writer are optional. There are several readers using the same satpy_cf_nc.py reader.

- Generic reader satpy_cf_nc
- EUMETSAT GAC FDR reader avhrr_l1c_eum_gac_fdr_nc

Generic reader

The generic satpy_cf_nc reader reads files of type:

```
'{platform_name}-{sensor}-{start_time:%Y%m%d%H%M%S}-{end_time:%Y%m%d%H%M%S}.nc'
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene

filenames = ['data/npp-viirs-mband-20201007075915-20201007080744.nc']
scn = Scene(reader='satpy_cf_nc', filenames=filenames)
scn.load(['M05'])
scn['M05']
```

Output:

```
<xarray.DataArray 'M05' (y: 4592, x: 3200)>
dask.array<open_dataset-d91cfbf1bf4f14710d27446d91cdc6e4M05, shape=(4592, 3200),
    dtype=float32, chunksize=(4096, 3200), chunktype=numpy.ndarray>
Coordinates:
    longitude  (y, x) float32 dask.array<chunksize=(4096, 3200), meta=np.ndarray>
    latitude   (y, x) float32 dask.array<chunksize=(4096, 3200), meta=np.ndarray>
Dimensions without coordinates: y, x
```

(continues on next page)

(continued from previous page)

```
Attributes:
    start_time:                    2020-10-07 07:59:15
    start_orbit:                   46350
    end_time:                      2020-10-07 08:07:44
    end_orbit:                     46350
    calibration:                   reflectance
    long_name:                     M05
    modifiers:                     ('sunz_corrected',)
    platform_name:                 Suomi-NPP
    resolution:                    742
    sensor:                        viirs
    standard_name:                 toa_bidirectional_reflectance
    units:                         %
    wavelength:                    0.672µm (0.662-0.682µm)
    date_created:                  2020-10-07T08:20:02Z
    instrument:                    VIIRS
```

Notes

Available datasets and attributes will depend on the data saved with the cf_writer.

EUMETSAT AVHRR GAC FDR L1C reader

The `avhrr_l1c_eum_gac_fdr_nc` reader reads files of type:

```
''AVHRR-GAC_FDR_1C_{platform}_{start_time:%Y%m%dT%H%M%SZ}_{end_time:%Y%m%dT%H%M%SZ}_
↪{processing_mode}_{disposition_mode}_{creation_time}_{version_int:04d}.nc'
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene

filenames = ['data/AVHRR-GAC_FDR_1C_N06_19810330T042358Z_19810330T060903Z_R_O_
↪20200101T000000Z_0100.nc']
scn = Scene(reader='avhrr_l1c_eum_gac_fdr_nc', filenames=filenames)
scn.load(['brightness_temperature_channel_4'])
scn['brightness_temperature_channel_4']
```

Output:

```
<xarray.DataArray 'brightness_temperature_channel_4' (y: 11, x: 409)>
dask.array<open_dataset-55ffbf3623b32077c67897f4283640a5brightness_temperature_channel_4,
↪ shape=(11, 409),
    dtype=float32, chunksize=(11, 409), chunktype=numpy.ndarray>
Coordinates:
  * x          (x) int16 0 1 2 3 4 5 6 7 8 ... 401 402 403 404 405 406 407 408
  * y          (y) int64 0 1 2 3 4 5 6 7 8 9 10
```

(continues on next page)

(continued from previous page)

```
    acq_time   (y) datetime64[ns] dask.array<chunksize=(11,), meta=np.ndarray>
    longitude  (y, x) float64 dask.array<chunksize=(11, 409), meta=np.ndarray>
    latitude   (y, x) float64 dask.array<chunksize=(11, 409), meta=np.ndarray>
Attributes:
    start_time:                           1981-03-30 04:23:58
    end_time:                             1981-03-30 06:09:03
    calibration:                          brightness_temperature
    modifiers:                            ()
    resolution:                           1050
    standard_name:                        toa_brightness_temperature
    units:                                K
    wavelength:                           10.8µm (10.3-11.3µm)
    Conventions:                          CF-1.8 ACDD-1.3
    comment:                              Developed in cooperation with EUME...
    creator_email:                        ops@eumetsat.int
    creator_name:                         EUMETSAT
    creator_url:                          https://www.eumetsat.int/
    date_created:                         2020-09-14T10:50:51.073707
    disposition_mode:                     O
    gac_filename:                         NSS.GHRR.NA.D81089.S0423.E0609.B09...
    geospatial_lat_max:                   89.95386902434623
    geospatial_lat_min:                   -89.97581969005503
    geospatial_lat_resolution:            1050 meters
    geospatial_lat_units:                 degrees_north
    geospatial_lon_max:                   179.99952992568998
    geospatial_lon_min:                   -180.0
    geospatial_lon_resolution:            1050 meters
    geospatial_lon_units:                 degrees_east
    ground_station:                       GC
    id:                                   DOI:10.5676/EUM/AVHRR_GAC_L1C_FDR/...
    institution:                          EUMETSAT
    instrument:                           Earth Remote Sensing Instruments >...
    keywords:                             ATMOSPHERE > ATMOSPHERIC RADIATION...
    keywords_vocabulary:                  GCMD Science Keywords, Version 9.1
    licence:                              EUMETSAT data policy https://www.e...
    naming_authority:                     int.eumetsat
    orbit_number_end:                     9123
    orbit_number_start:                   9122
    orbital_parameters_tle:               ['1 11416U 79057A   81090.16350942...
    platform:                             Earth Observation Satellites > NOA...
    processing_level:                     1C
    processing_mode:                      R
    product_version:                      1.0.0
    references:                           Devasthale, A., M. Raspaud, C. Sch...
    source:                               AVHRR GAC Level 1 Data
    standard_name_vocabulary:             CF Standard Name Table v73
    summary:                              Fundamental Data Record (FDR) of m...
    sun_earth_distance_correction_factor: 0.9975244779999585
    time_coverage_end:                    19820803T003900Z
    time_coverage_start:                  19800101T000000Z
    title:                                AVHRR GAC L1C FDR
    version_calib_coeffs:                 PATMOS-x, v2017r1
```

(continues on next page)

(continued from previous page)

```
    version_pygac:                          1.4.0
    version_pygac_fdr:                      0.1.dev107+gceb7b26.d20200910
    version_satpy:                          0.21.1.dev894+g5cf76e6
    history:                                Created by pytroll/satpy on 2020-0...
    name:                                   brightness_temperature_channel_4
    _satpy_id:                              DataID(name='brightness_temperatur...
    ancillary_variables:                    []
```

class satpy.readers.satpy_cf_nc.**SatpyCFFileHandler**(*filename*, *filename_info*, *filetype_info*, *numeric_name_prefix='CHANNEL_'*)

Bases: *BaseFileHandler*

File handler for Satpy's CF netCDF files.

Initialize file handler.

_assign_ds_info(*var_name*, *val*)

Assign ds_info.

_compare_attr(*_ds_id_dict*, *key*, *data*)

_coordinate_datasets(*configured_datasets=None*)

Add information of coordinate datasets.

_dataid_attrs_equal(*ds_id*, *data*)

_dynamic_datasets()

Add information of dynamic datasets.

_existing_datasets(*configured_datasets=None*)

Add information of existing datasets.

available_datasets(*configured_datasets=None*)

Add information of available datasets.

property end_time

Get end time.

fix_modifier_attr(*ds_info*)

Fix modifiers attribute.

get_area_def(*dataset_id*)

Get area definition from CF complient netcdf.

get_dataset(*ds_id*, *ds_info*)

Get dataset.

property sensor_names

Get sensor set.

property start_time

Get start time.

satpy.readers.satpy_cf_nc.**_str2dict**(*val*)

Convert string to dictionary.

satpy.readers.scmi module

SCMI NetCDF4 Reader.

SCMI files are typically used for data for the ABI instrument onboard the GOES-16/17 satellites. It is the primary format used for providing ABI data to the AWIPS visualization clients used by the US National Weather Service forecasters. The python code for this reader may be reused by other readers as NetCDF schemes/metadata change for different products. The initial reader using this code is the "scmi_abi" reader (see *abi_l1b_scmi.yaml* for more information).

There are two forms of these files that this reader supports:

1. **Official SCMI format: NetCDF4 files where the main data variable is stored** in a variable called "Sectorized_CMI". This variable name can be configured in the YAML configuration file.
2. **Satpy/Polar2Grid SCMI format: NetCDF4 files based on the official SCMI** format created for the Polar2Grid project. This format was migrated to Satpy as part of Polar2Grid's adoption of Satpy for the majority of its features. This format is what is produced by Satpy's *scmi* writer. This format can be identified by a single variable named "data" and a global attribute named `"awips_id"` that is set to a string starting with `"AWIPS_"`.

class `satpy.readers.scmi.`**`SCMIFileHandler`**(*filename*, *filename_info*, *filetype_info*)

Bases: `BaseFileHandler`

Handle a single SCMI NetCDF4 file.

Set up the SCMI file handler.

`_calc_extents`(*proj_dict*)

Calculate area extents from x/y variables.

`_get_cf_grid_mapping_var`()

Figure out which grid mapping should be used.

`_get_proj4_name`(*projection*)

Map CF projection name to PROJ.4 name.

`_get_proj_specific_params`(*projection*)

Convert CF projection parameters to PROJ.4 dict.

`_get_sensor`()

Determine the sensor for this file.

property `end_time`

Get the end time.

`get_area_def`(*key*)

Get the area definition of the data at hand.

`get_dataset`(*key*, *info*)

Load a dataset.

`get_shape`(*key*, *info*)

Get the shape of the data.

property `sensor_names`

Get the sensor names.

property `start_time`

Get the start time.

satpy.readers.seadas_l2 module

Reader for SEADAS L2 products.

This reader currently only supports MODIS and VIIRS Chlorophyll A from SEADAS.

The reader includes an additional keyword argument `apply_quality_flags` which can be used to mask out low-quality pixels based on quality flags contained in the file (`l2_flags`). This option defaults to `False`, but when set to `True` the "CHLWARN" pixels of the `l2_flags` variable are masked out. These pixels represent data where the chlorophyll algorithm warned about the quality of the result.

class satpy.readers.seadas_l2.**SEADASL2HDFFileHandler**(*filename*, *filename_info*, *filetype_info*, *apply_quality_flags=False*)

Bases: *_SEADASL2Base*, *HDF4FileHandler*

Simple handler of SEADAS L2 HDF4 files.

Initialize file handler and determine if data quality flags should be applied.

end_time_attr_name = '/attr/End Time'

l2_flags_var_name = 'l2_flags'

platform_attr_name = '/attr/Mission'

sensor_attr_name = '/attr/Sensor Name'

start_time_attr_name = '/attr/Start Time'

time_format = '%Y%j%H%M%S'

class satpy.readers.seadas_l2.**SEADASL2NetCDFFileHandler**(*filename*, *filename_info*, *filetype_info*, *apply_quality_flags=False*)

Bases: *_SEADASL2Base*, *NetCDF4FileHandler*

Simple handler of SEADAS L2 NetCDF4 files.

Initialize file handler and determine if data quality flags should be applied.

end_time_attr_name = '/attr/time_coverage_end'

l2_flags_var_name = 'geophysical_data/l2_flags'

platform_attr_name = '/attr/platform'

sensor_attr_name = '/attr/instrument'

start_time_attr_name = '/attr/time_coverage_start'

time_format = '%Y-%m-%dT%H:%M:%S.%f'

class satpy.readers.seadas_l2.**_SEADASL2Base**(*filename*, *filename_info*, *filetype_info*, *apply_quality_flags=False*)

Bases: `object`

Simple handler of SEADAS L2 files.

Initialize file handler and determine if data quality flags should be applied.

_add_satpy_metadata(*data*)

_filter_by_valid_min_max(*data_arr*)

_get_file_key_and_variable(*data_id*, *dataset_info*)

_mask_based_on_l2_flags(*data_arr*)

_platform_name()

_rename_2d_dims_if_necessary(*data_arr*)

_rows_per_scan()

_valid_min_max(*data_arr*)

property end_time

Get the ending observation time of this file's data.

get_dataset(*data_id*, *dataset_info*)

Get DataArray for the specified DataID.

property sensor_names

Get sensor for the current file's data.

property start_time

Get the starting observation time of this file's data.

satpy.readers.seviri_base module

Common functionality for SEVIRI L1.5 data readers.

Introduction

The Spinning Enhanced Visible and InfraRed Imager (SEVIRI) is the primary instrument on Meteosat Second Generation (MSG) and has the capacity to observe the Earth in 12 spectral channels.

Level 1.5 corresponds to image data that has been corrected for all unwanted radiometric and geometric effects, has been geolocated using a standardised projection, and has been calibrated and radiance-linearised. (From the EUMETSAT documentation)

Satpy provides the following readers for SEVIRI L1.5 data in different formats:

- Native: `satpy.readers.seviri_l1b_native`
- HRIT: `satpy.readers.seviri_l1b_hrit`
- netCDF: `satpy.readers.seviri_l1b_nc`

Calibration

This section describes how to control the calibration of SEVIRI L1.5 data.

Calibration to radiance

The SEVIRI L1.5 data readers allow for choosing between two file-internal calibration coefficients to convert counts to radiances:

- Nominal for all channels (default)
- GSICS where available (IR currently) and nominal for the remaining channels (VIS & HRV currently)

In order to change the default behaviour, use the `reader_kwargs` keyword argument upon Scene creation:

```
import satpy
scene = satpy.Scene(filenames=filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'calib_mode': 'GSICS'})
scene.load(['VIS006', 'IR_108'])
```

In addition, two other calibration methods are available:

1. It is possible to specify external calibration coefficients for the conversion from counts to radiances. External coefficients take precedence over internal coefficients and over the Meirink coefficients, but you can also mix internal and external coefficients: If external calibration coefficients are specified for only a subset of channels, the remaining channels will be calibrated using the chosen file-internal coefficients (nominal or GSICS). Calibration coefficients must be specified in [mW m-2 sr-1 (cm-1)-1].
2. The calibration mode `meirink-2023` uses coefficients based on an intercalibration with Aqua-MODIS for the visible channels, as found in Inter-calibration of polar imager solar channels using SEVIRI (2013) by J. F. Meirink, R. A. Roebeling, and P. Stammes.

In the following example we use external calibration coefficients for the `VIS006` & `IR_108` channels, and nominal coefficients for the remaining channels:

```
coefs = {'VIS006': {'gain': 0.0236, 'offset': -1.20},
         'IR_108': {'gain': 0.2156, 'offset': -10.4}}
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'ext_calib_coefs': coefs})
scene.load(['VIS006', 'VIS008', 'IR_108', 'IR_120'])
```

In the next example we use external calibration coefficients for the `VIS006` & `IR_108` channels, GSICS coefficients where available (other IR channels) and nominal coefficients for the rest:

```
coefs = {'VIS006': {'gain': 0.0236, 'offset': -1.20},
         'IR_108': {'gain': 0.2156, 'offset': -10.4}}
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'calib_mode': 'GSICS',
                                   'ext_calib_coefs': coefs})
scene.load(['VIS006', 'VIS008', 'IR_108', 'IR_120'])
```

In the next example we use the mode `meirink-2023` calibration coefficients for all visible channels and nominal coefficients for the rest:

```
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_...',
                    reader_kwargs={'calib_mode': 'meirink-2023'})
scene.load(['VIS006', 'VIS008', 'IR_016'])
```

Calibration to reflectance

When loading solar channels, the SEVIRI L1.5 data readers apply a correction for the Sun-Earth distance variation throughout the year - as recommended by the EUMETSAT document Conversion from radiances to reflectances for SEVIRI warm channels. In the unlikely situation that this correction is not required, it can be removed on a per-channel basis using *satpy.readers.utils.remove_earthsun_distance_correction()*.

Masking of bad quality scan lines

By default bad quality scan lines are masked and replaced with `np.nan` for radiance, reflectance and brightness temperature calibrations based on the quality flags provided by the data (for details on quality flags see MSG Level 1.5 Image Data Format Description page 109). To disable masking `reader_kwargs={'mask_bad_quality_scan_lines': False}` can be passed to the Scene.

Metadata

The SEVIRI L1.5 readers provide the following metadata:

- The `orbital_parameters` attribute provides the nominal and actual satellite position, as well as the projection centre. See the *Metadata* section in the *Reading* chapter for more information.
- The `acq_time` coordinate provides the mean acquisition time for each scanline. Use a `MultiIndex` to enable selection by acquisition time:

```
import pandas as pd
mi = pd.MultiIndex.from_arrays([scn['IR_108']['y'].data, scn['IR_108']['acq_time'].
↪data],
                               names=('y_coord', 'time'))
scn['IR_108']['y'] = mi
scn['IR_108'].sel(time=np.datetime64('2019-03-01T12:06:13.052000000'))
```

- Raw metadata from the file header can be included by setting the reader argument `include_raw_metadata=True` (HRIT and Native format only). Note that this comes with a performance penalty of up to 10% if raw metadata from multiple segments or scans need to be combined. By default, arrays with more than 100 elements are excluded to limit the performance penalty. This threshold can be adjusted using the `mda_max_array_size` reader keyword argument:

```
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_hrit/native',
                    reader_kwargs={'include_raw_metadata': True,
                                   'mda_max_array_size': 1000})
```

References

- MSG Level 1.5 Image Data Format Description
- Radiometric Calibration of MSG SEVIRI Level 1.5 Image Data in Equivalent Spectral Blackbody Radiance

class satpy.readers.seviri_base.**MeirinkCalibrationHandler**(*calib_mode*)

Bases: `object`

Re-calibration of the SEVIRI visible channels slope (see Meirink 2013).

Initialize the calibration handler.

get_slope(*platform*, *channel*, *time*)

Return the slope using the provided calibration coefficients.

class satpy.readers.seviri_base.**MpefProductHeader**

Bases: object

MPEF product header class.

get()

Return numpy record_array for MPEF product header.

property images_used

Return structure for images_used.

exception satpy.readers.seviri_base.**NoValidOrbitParams**

Bases: Exception

Exception when validOrbitParameters are missing.

class satpy.readers.seviri_base.**OrbitPolynomial**(*coefs*, *start_time*, *end_time*)

Bases: object

Polynomial encoding the satellite position.

Satellite position as a function of time is encoded in the coefficients of an 8th-order Chebyshev polynomial.

Initialize the polynomial.

evaluate(*time*)

Get satellite position in earth-centered cartesian coordinates.

Parameters

time – Timestamp where to evaluate the polynomial

Returns

Earth-centered cartesian coordinates (x, y, z) in meters

class satpy.readers.seviri_base.**OrbitPolynomialFinder**(*orbit_polynomials*)

Bases: object

Find orbit polynomial for a given timestamp.

Initialize with the given candidates.

Parameters

orbit_polynomials – Dictionary of orbit polynomials as found in SEVIRI L1B files:

```
{'X': x_polynomials,
 'Y': y_polynomials,
 'Z': z_polynomials,
 'StartTime': polynomials_valid_from,
 'EndTime': polynomials_valid_to}
```

_get_closest_interval(*time*)

Find interval closest to the given timestamp.

Returns

Index of closest interval, distance from its center

_get_closest_interval_within(*time*, *threshold*)

Find interval closest to the given timestamp within a given distance.

Parameters

- **time** – Timestamp of interest
- **threshold** – Maximum distance between timestamp and interval center

Returns
Index of closest interval

_get_enclosing_interval(*time*)

Find interval enclosing the given timestamp.

get_orbit_polynomial(*time*, *max_delta=6*)

Get orbit polynomial valid for the given time.

Orbit polynomials are only valid for certain time intervals. Find the polynomial, whose corresponding interval encloses the given timestamp. If there are multiple enclosing intervals, use the most recent one. If there is no enclosing interval, find the interval whose centre is closest to the given timestamp (but not more than `max_delta` hours apart).

Why are there gaps between those intervals? Response from EUM:

A manoeuvre is a discontinuity in the orbit parameters. The flight dynamic algorithms are not made to interpolate over the time-span of the manoeuvre; hence we have elements describing the orbit before a manoeuvre and a new set of elements describing the orbit after the manoeuvre. The flight dynamic products are created so that there is an intentional gap at the time of the manoeuvre. Also the two pre-manoeuvre elements may overlap. But the overlap is not of an issue as both sets of elements describe the same pre-manoeuvre orbit (with negligible variations).

class satpy.readers.seviri_base.**SEVIRICalibrationAlgorithm**(*platform_id*, *scan_time*)

Bases: `object`

SEVIRI calibration algorithms.

Initialize the calibration algorithm.

_erads2bt(*data*, *channel_name*)

Convert effective radiance to brightness temperature.

_srads2bt(*data*, *channel_name*)

Convert spectral radiance to brightness temperature.

_tl15(*data*, *wavenumber*)

Compute the L15 temperature.

convert_to_radiance(*data*, *gain*, *offset*)

Calibrate to radiance.

ir_calibrate(*data*, *channel_name*, *cal_type*)

Calibrate to brightness temperature.

vis_calibrate(*data*, *solar_irradiance*)

Calibrate to reflectance.

This uses the method described in Conversion from radiances to reflectances for SEVIRI warm channels: https://www-cdn.eumetsat.int/files/2020-04/pdf_msg_seviri_rad2refl.pdf

class satpy.readers.seviri_base.**SEVIRICalibrationHandler**(*platform_id*, *channel_name*, *coefs*, *calib_mode*, *scan_time*)

Bases: object

Calibration handler for SEVIRI HRIT-, native- and netCDF-formats.

Handles selection of calibration coefficients and calls the appropriate calibration algorithm.

Initialize the calibration handler.

calibrate(*data*, *calibration*)

Calibrate the given data.

get_gain_offset()

Get gain & offset for calibration from counts to radiance.

Choices for internal coefficients are nominal or GSICS. If no GSICS coefficients are available for a certain channel, fall back to nominal coefficients. External coefficients take precedence over internal coefficients.

satpy.readers.seviri_base.**_create_bad_quality_lines_mask**(*line_validity*, *line_geometric_quality*, *line_radiometric_quality*)

Create bad quality scan lines mask.

For details on quality flags see MSG Level 1.5 Image Data Format Description page 109.

Parameters

- **line_validity** (*numpy.ndarray*) – Quality flags with shape (nlines,).
- **line_geometric_quality** (*numpy.ndarray*) – Quality flags with shape (nlines,).
- **line_radiometric_quality** (*numpy.ndarray*) – Quality flags with shape (nlines,).

Returns

Indicating if the scan line is bad.

Return type

numpy.ndarray

satpy.readers.seviri_base.**add_scanline_acq_time**(*dataset*, *acq_time*)

Add scanline acquisition time to the given dataset.

satpy.readers.seviri_base.**calculate_area_extent**(*area_dict*)

Calculate the area extent seen by a geostationary satellite.

Parameters

area_dict – A dictionary containing the required parameters center_point: Center point for the projection north: Northmost row number east: Eastmost column number west: Westmost column number south: Southmost row number column_step: Pixel resolution in meters in east-west direction line_step: Pixel resolution in meters in south-north direction [column_offset: Column offset, defaults to 0 if not given] [line_offset: Line offset, defaults to 0 if not given]

Returns

An area extent for the scene defined by the lower left and
upper right corners

Return type

tuple

For Earth model 2 and full disk VISIR, (center_point - west - 0.5 + we_offset) must be -1856.5 . # See MSG Level 1.5 Image Data Format Description Figure 7 - Alignment and numbering of the non-HRV pixels.

satpy.readers.seviri_base.**chebyshev**(*coefs*, *time*, *domain*)

Evaluate a Chebyshev Polynomial.

Parameters

- **coefs** (*list, np.array*) – Coefficients defining the polynomial
- **time** (*int, float*) – Time where to evaluate the polynomial
- **domain** (*list, tuple*) – Domain (or time interval) for which the polynomial is defined: [left, right]

Reference: Appendix A in the MSG Level 1.5 Image Data Format Description.

satpy.readers.seviri_base.**chebyshev_3d**(*coefs*, *time*, *domain*)

Evaluate Chebyshev Polynomials for three dimensions (x, y, z).

Expects the three coefficient sets to be defined in the same domain.

Parameters

- **coefs** – (x, y, z) coefficient sets.
- **time** – See *chebyshev()*
- **domain** – See *chebyshev()*

Returns

Polynomials evaluated in (x, y, z) dimension.

satpy.readers.seviri_base.**create_coef_dict**(*coefs_nominal*, *coefs_gsics*, *radiance_type*, *ext_coefs*)

Create coefficient dictionary expected by calibration class.

satpy.readers.seviri_base.**dec10216**(*inbuf*)

Decode 10 bits data into 16 bits words.

```
/*
 * pack 4 10-bit words in 5 bytes into 4 16-bit words
 *
 * 0        1        2        3        4        5
 * 0123456789012345678901234567890123456789 0
 * 0          1          2          3          4
 */
ip = &in_buffer[i];
op = &out_buffer[j];
op[0] = ip[0]*4 + ip[1]/64;
op[1] = (ip[1] & 0x3F)*16 + ip[2]/16;
op[2] = (ip[2] & 0x0F)*64 + ip[3]/4;
op[3] = (ip[3] & 0x03)*256 +ip[4];
```

satpy.readers.seviri_base.**get_cds_time**(*days*, *msecs*)

Compute timestamp given the days since epoch and milliseconds of the day.

1958-01-01 00:00 is interpreted as fill value and will be replaced by NaT (Not a Time).

Parameters

- **days** (*int, either scalar or numpy.ndarray*) – Days since 1958-01-01
- **msecs** (*int, either scalar or numpy.ndarray*) – Milliseconds of the day

Returns

Timestamp(s)

Return type
numpy.datetime64

satpy.readers.seviri_base.**get_meirink_slope**(*meirink_coefs*, *acquisition_time*)

Compute the slope for the visible channel calibration according to Meirink 2013.

S = A + B * 1.e-3* Day

S is here in µW m-2 sr-1 (cm-1)-1

EUMETSAT calibration is given in mW m-2 sr-1 (cm-1)-1, so an extra factor of 1/1000 must be applied.

satpy.readers.seviri_base.**get_padding_area**(*shape*, *dtype*)

Create a padding area filled with no data.

satpy.readers.seviri_base.**get_satpos**(*orbit_polynomial*, *time*, *semi_major_axis*, *semi_minor_axis*)

Get satellite position in geodetic coordinates.

Parameters

- **orbit_polynomial** – OrbitPolynomial instance
- **time** – Timestamp where to evaluate the polynomial
- **semi_major_axis** – Semi-major axis of the ellipsoid
- **semi_minor_axis** – Semi-minor axis of the ellipsoid

Returns
Longitude [deg east], Latitude [deg north] and Altitude [m]

satpy.readers.seviri_base.**mask_bad_quality**(*data*, *line_validity*, *line_geometric_quality*, *line_radiometric_quality*)

Mask scan lines with bad quality.

Parameters

- **data** (*xarray.DataArray*) – Channel data
- **line_validity** (*numpy.ndarray*) – Quality flags with shape (nlines,).
- **line_geometric_quality** (*numpy.ndarray*) – Quality flags with shape (nlines,).
- **line_radiometric_quality** (*numpy.ndarray*) – Quality flags with shape (nlines,).

Returns
data with lines flagged as bad converted to np.nan.

Return type
xarray.DataArray

satpy.readers.seviri_base.**pad_data_horizontally**(*data*, *final_size*, *east_bound*, *west_bound*)

Pad the data given east and west bounds and the desired size.

satpy.readers.seviri_base.**pad_data_vertically**(*data*, *final_size*, *south_bound*, *north_bound*)

Pad the data given south and north bounds and the desired size.

satpy.readers.seviri_base.**round_nom_time**(*dt*, *time_delta*)

Round a datetime object to a multiple of a timedelta.

dt : datetime.datetime object, default now. time_delta : timedelta object, we round to a multiple of this, default 1 minute. adapted for SEVIRI from: https://stackoverflow.com/questions/3463930/how-to-round-the-minute-of-a-datetime-object-python

satpy.readers.seviri_base.**should_apply_meirink**(*calib_mode*, *channel_name*)

Decide whether to use the Meirink calibration coefficients.

satpy.readers.seviri_l1b_hrit module

SEVIRI Level 1.5 HRIT format reader.

Introduction

The `seviri_l1b_hrit` reader reads and calibrates MSG-SEVIRI L1.5 image data in HRIT format. The format is explained in the MSG Level 1.5 Image Data Format Description. The files are usually named as follows:

```
H-000-MSG4__-MSG4________-_________-PRO______-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000001___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000002___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000003___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000004___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000005___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000006___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000007___-201903011200-__
H-000-MSG4__-MSG4________-IR_108___-000008___-201903011200-__
H-000-MSG4__-MSG4________-_________-EPI______-201903011200-__
```

Each image is decomposed into 24 segments (files) for the high-resolution-visible (HRV) channel and 8 segments for other visible (VIS) and infrared (IR) channels. Additionally, there is one prologue and one epilogue file for the entire scan which contain global metadata valid for all channels.

Reader Arguments

Some arguments can be provided to the reader to change its behaviour. These are provided through the *Scene* instantiation, eg:

```
scn = Scene(filenames=filenames, reader="seviri_l1b_hrit", reader_kwargs={'fill_hrv':␣
↪False})
```

To see the full list of arguments that can be provided, look into the documentation of *HRITMSGFileHandler*.

Compression

This reader accepts compressed HRIT files, ending in `C_` as other HRIT readers, see *satpy.readers.hrit_base.HRITFileHandler*.

This reader also accepts bzipped file with the extension `.bz2` for the prologue, epilogue, and segment files.

Nominal start/end time

Warning: attribute access change

`nominal_start_time` and `nominal_end_time` should be accessed using the `time_parameters` attribute.

`nominal_start_time` and `nominal_end_time` are also available directly via `start_time` and `end_time` respectively.

Here is an exmaple of the content of the start/end time and `time_parameters` attibutes

```
Start time: 2019-08-29 12:00:00
End time:   2019-08-29 12:15:00
time_parameters:
                {'nominal_start_time': datetime.datetime(2019, 8, 29, 12, 0),
                 'nominal_end_time': datetime.datetime(2019, 8, 29, 12, 15),
                 'observation_start_time': datetime.datetime(2019, 8, 29, 12, 0, 9,␣
↪338000),
                 'observation_end_time': datetime.datetime(2019, 8, 29, 12, 15, 9,␣
↪203000)
                 }
```

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene
import glob

filenames = glob.glob('data/H-000-MSG4__-MSG4________-*201903011200*')
scn = Scene(filenames=filenames, reader='seviri_l1b_hrit')
scn.load(['VIS006', 'IR_108'])
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
    acq_time  (y) datetime64[ns] NaT NaT NaT NaT NaT NaT ... NaT NaT NaT NaT NaT
  * x         (x) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
  * y         (y) float64 -5.566e+06 -5.563e+06 ... 5.566e+06 5.569e+06
Attributes:
    orbital_parameters:       {'projection_longitude': 0.0, 'projection_latit...
    platform_name:            Meteosat-11
    georef_offset_corrected:  True
    standard_name:            brightness_temperature
    raw_metadata:             {'file_type': 0, 'total_header_length': 6198, '...
    wavelength:               (9.8, 10.8, 11.8)
    units:                    K
    sensor:                   seviri
```

(continues on next page)

(continued from previous page)

```
    platform_name:            Meteosat-11
    start_time:               2019-03-01 12:00:09.716000
    end_time:                 2019-03-01 12:12:42.946000
    area:                     Area ID: some_area_name\\nDescription: On-the-fl...
    name:                     IR_108
    resolution:               3000.403165817
    calibration:              brightness_temperature
    polarization:             None
    level:                    None
    modifiers:                ()
    ancillary_variables:      []
```

The *filenames* argument can either be a list of strings, see the example above, or a list of `satpy.readers.FSFile` objects. FSFiles can be used in conjunction with fsspec, e.g. to handle in-memory data:

```
import glob

from fsspec.implementations.memory import MemoryFile, MemoryFileSystem
from satpy import Scene
from satpy.readers import FSFile

# In this example, we will make use of `MemoryFile`s in a `MemoryFileSystem`.
memory_fs = MemoryFileSystem()

# Usually, the data already resides in memory.
# For explanatory reasons, we will load the files found with glob in memory,
#  and load the scene with FSFiles.
filenames = glob.glob('data/H-000-MSG4__-MSG4_______-*201903011200*')
fs_files = []
for fn in filenames:
    with open(fn, 'rb') as fh:
        fs_files.append(MemoryFile(
            fs=memory_fs,
            path="{}{}".format(memory_fs.root_marker, fn),
            data=fh.read()
        ))
        fs_files[-1].commit()  # commit the file to the filesystem
fs_files = [FSFile(open_file) for open_file in filenames]  # wrap MemoryFiles as FSFiles
# similar to the example above, we pass a list of FSFiles to the `Scene`
scn = Scene(filenames=fs_files, reader='seviri_l1b_hrit')
scn.load(['VIS006', 'IR_108'])
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray (y: 3712, x: 3712)>
dask.array<shape=(3712, 3712), dtype=float32, chunksize=(464, 3712)>
Coordinates:
    acq_time  (y) datetime64[ns] NaT NaT NaT NaT NaT NaT ... NaT NaT NaT NaT NaT
  * x         (x) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
  * y         (y) float64 -5.566e+06 -5.563e+06 ... 5.566e+06 5.569e+06
Attributes:
```

(continues on next page)

(continued from previous page)

```
    orbital_parameters:       {'projection_longitude': 0.0, 'projection_latit...
    platform_name:            Meteosat-11
    georef_offset_corrected:  True
    standard_name:            brightness_temperature
    raw_metadata:             {'file_type': 0, 'total_header_length': 6198, '...
    wavelength:               (9.8, 10.8, 11.8)
    units:                    K
    sensor:                   seviri
    platform_name:            Meteosat-11
    start_time:               2019-03-01 12:00:09.716000
    end_time:                 2019-03-01 12:12:42.946000
    area:                     Area ID: some_area_name\\nDescription: On-the-fl...
    name:                     IR_108
    resolution:               3000.403165817
    calibration:              brightness_temperature
    polarization:             None
    level:                    None
    modifiers:                ()
    ancillary_variables:      []
```

References

- EUMETSAT Product Navigator
- MSG Level 1.5 Image Data Format Description
- fsspec

class satpy.readers.seviri_l1b_hrit.**HRITMSGEpilogueFileHandler**(*filename*, *filename_info*, *filetype_info*, *calib_mode='nominal'*, *ext_calib_coefs=None*, *include_raw_metadata=False*, *mda_max_array_size=None*, *fill_hrv=None*, *mask_bad_quality_scan_lines=None*)

Bases: *HRITMSGPrologueEpilogueBase*

SEVIRI HRIT epilogue reader.

Initialize the reader.

read_epilogue()

Read the epilogue metadata.

reduce(*max_size*)

Reduce the epilogue metadata.

class satpy.readers.seviri_l1b_hrit.**HRITMSGFileHandler**(*filename*, *filename_info*, *filetype_info*, *prologue*, *epilogue*, *calib_mode='nominal'*, *ext_calib_coefs=None*, *include_raw_metadata=False*, *mda_max_array_size=100*, *fill_hrv=True*, *mask_bad_quality_scan_lines=True*)

Bases: *HRITFileHandler*

SEVIRI HRIT format reader.

Calibration

See *satpy.readers.seviri_base*.

Padding of the HRV channel

By default, the HRV channel is loaded padded with no-data, returning a full-disk dataset. If you want the original, unpadded data, just provide the *fill_hrv* as False in the *reader_kwargs*:

```
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_hrit',
                    reader_kwargs={'fill_hrv': False})
```

Metadata

See *satpy.readers.seviri_base*.

Initialize the reader.

_add_scanline_acq_time(*dataset*)

Add scanline acquisition time to the given dataset.

_get_area_extent(*pdict*)

Get the area extent of the file.

Until December 2017, the data is shifted by 1.5km SSP North and West against the nominal GEOS projection. Since December 2017 this offset has been corrected. A flag in the data indicates if the correction has been applied. If no correction was applied, adjust the area extent to match the shifted data.

For more information see Section 3.1.4.2 in the MSG Level 1.5 Image Data Format Description. The correction of the area extent is documented in a developer's memo.

_get_calib_coefs(*channel_name*)

Get coefficients for calibration from counts to radiance.

_get_header()

Read the header info, and fill the metadata dictionary.

_get_raw_mda()

Compile raw metadata to be included in the dataset attributes.

_mask_bad_quality(*data*)

Mask scanlines with bad quality.

property _repeat_cycle_duration

Get repeat cycle duration from epilogue.

_update_attrs(*res*, *info*)

Update dataset attributes.

calibrate(*data*, *calibration*)

Calibrate the data.

property end_time

Get the general end time for this file.

get_area_def(*dsid*)

Get the area definition of the band.

get_dataset(*key*, *info*)

Get the dataset.

property nominal_end_time

Get the end time and round it according to scan law.

property nominal_start_time

Get the start time and round it according to scan law.

property observation_end_time

Get the observation end time.

property observation_start_time

Get the observation start time.

pad_hrv_data(*res*)

Add empty pixels around the HRV.

property start_time

Get general start time for this file.

class satpy.readers.seviri_l1b_hrit.**HRITMSGPrologueEpilogueBase**(*filename*, *filename_info*, *filetype_info*, *hdr_info*)

Bases: *HRITFileHandler*

Base reader for prologue and epilogue files.

Initialize the file handler for prologue and epilogue files.

_reduce(*mda*, *max_size*)

Reduce the metadata.

reduce(*max_size*)

Reduce the metadata (placeholder).

class satpy.readers.seviri_l1b_hrit.**HRITMSGPrologueFileHandler**(*filename*, *filename_info*, *filetype_info*, *calib_mode='nominal'*, *ext_calib_coefs=None*, *include_raw_metadata=False*, *mda_max_array_size=None*, *fill_hrv=None*, *mask_bad_quality_scan_lines=None*)

Bases: *HRITMSGPrologueEpilogueBase*

SEVIRI HRIT prologue reader.

Initialize the reader.

get_earth_radii()

Get earth radii from prologue.

Returns

Equatorial radius, polar radius [m]

read_prologue()

Read the prologue metadata.

reduce(*max_size*)

Reduce the prologue metadata.

property satpos

Get actual satellite position in geodetic coordinates (WGS-84).

Evaluate orbit polynomials at the start time of the scan.

Returns: Longitude [deg east], Latitude [deg north] and Altitude [m]

satpy.readers.seviri_l1b_hrit.**pad_data**(*data*, *final_size*, *east_bound*, *west_bound*)

Pad the data given east and west bounds and the desired size.

satpy.readers.seviri_l1b_icare module

Interface to SEVIRI L1B data from ICARE (Lille).

Introduction

The `seviri_l1b_icare` reader reads MSG-SEVIRI L1.5 image data in HDF format that has been produced by the ICARE Data and Services Center Data can be accessed via: http://www.icare.univ-lille1.fr

Each SEVIRI timeslot comes as 12 HDF files, one per band. Only those bands that are of interest need to be passed to the reader. Others can be ignored. Filenames follow the format: GEO_L1B-MSG1_YYYY-MM-DDTHH-MM-SS_G_CHANN_VX-XX.hdf Where: YYYY, MM, DD, HH, MM, SS specify the timeslot starting time. CHANN is the channel (i.e: HRV, IR016, WV073, etc) VX-XX is the processing version number

Example:

Here is an example how to read the data in satpy:

```
from satpy import Scene
import glob

filenames = glob.glob('data/*2019-03-01T12-00-00*.hdf')
scn = Scene(filenames=filenames, reader='seviri_l1b_icare')
scn.load(['VIS006', 'IR_108'])
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray 'array-a1d52b7e19ec5a875e2f038df5b60d7e' (y: 3712, x: 3712)>
dask.array<add, shape=(3712, 3712), dtype=float32, chunksize=(1024, 1024),␣
↪chunktype=numpy.ndarray>
Coordinates:
    crs      object +proj=geos +a=6378169.0 +b=6356583.8 +lon_0=0.0 +h=35785831.0␣
↪+units=m +type=crs
  * y        (y) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
  * x        (x) float64 -5.566e+06 -5.563e+06 -5.56e+06 ... 5.566e+06 5.569e+06
Attributes:
```

(continues on next page)

(continued from previous page)

```
    start_time:           2004-12-29 12:15:00
    end_time:             2004-12-29 12:27:44
    area:                 Area ID: geosmsg\nDescription: MSG/SEVIRI low resol...
    name:                 IR_108
    resolution:           3000.403165817
    calibration:          brightness_temperature
    polarization:         None
    level:                None
    modifiers:            ()
    ancillary_variables:  []
```

class satpy.readers.seviri_l1b_icare.**SEVIRI_ICARE**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF4FileHandler*

SEVIRI L1B handler for HDF4 files.

Init the file handler.

_get_dsname(*ds_id*)

Return the correct dataset name based on requested band.

property alt

Get the altitude.

property end_time

Get the end time.

property geoloc

Get the geolocation.

get_area_def(*ds_id*)

Get the area def.

get_dataset(*ds_id*, *ds_info*)

Get the dataset.

get_metadata(*data*, *ds_info*)

Get the metadata.

property projection

Get the projection.

property projlon

Get the projection longitude.

property res

Get the resolution.

property satlon

Get the satellite longitude.

property sensor_name

Get the sensor name.

property start_time

Get the start time.

property zone

Get the zone.

satpy.readers.seviri_l1b_native module

SEVIRI Level 1.5 native format reader.

Introduction

The `seviri_l1b_native` reader reads and calibrates MSG-SEVIRI L1.5 image data in binary format. The format is explained in the MSG Level 1.5 Native Format File Definition. The files are usually named as follows:

```
MSG4-SEVI-MSG15-0100-NA-20210302124244.185000000Z-NA.nat
```

Reader Arguments

Some arguments can be provided to the reader to change its behaviour. These are provided through the *Scene* instantiation, eg:

```
scn = Scene(filenames=filenames, reader="seviri_l1b_native", reader_kwargs={'fill_disk':␣
↪True})
```

To see the full list of arguments that can be provided, look into the documentation of *NativeMSGFileHandler*.

Example:

Here is an example how to read the data in satpy.

NOTE: When loading the data, the orientation of the image can be set with `upper_right_corner`-keyword. Possible options are `NW`, `NE`, `SW`, `SE`, or `native`.

```
from satpy import Scene

filenames = ['MSG4-SEVI-MSG15-0100-NA-20210302124244.185000000Z-NA.nat']
scn = Scene(filenames=filenames, reader='seviri_l1b_native')
scn.load(['VIS006', 'IR_108'], upper_right_corner='NE')
print(scn['IR_108'])
```

Output:

```
<xarray.DataArray 'reshape-969ef97d34b7b0c70ca19f53c6abcb68' (y: 3712, x: 3712)>
dask.array<truediv, shape=(3712, 3712), dtype=float32, chunksize=(928, 3712),␣
↪chunktype=numpy.ndarray>
Coordinates:
    acq_time  (y) datetime64[ns] NaT NaT NaT NaT NaT NaT ... NaT NaT NaT NaT NaT
    crs       object PROJCRS["unknown",BASEGEOGCRS["unknown",DATUM["unknown",...
  * y         (y) float64 -5.566e+06 -5.563e+06 ... 5.566e+06 5.569e+06
  * x         (x) float64 5.566e+06 5.563e+06 5.56e+06 ... -5.566e+06 -5.569e+06
Attributes:
    orbital_parameters:       {'projection_longitude': 0.0, 'projection_latit...
```

(continues on next page)

(continued from previous page)

```
    time_parameters:          {'nominal_start_time': datetime.datetime(2021, ...
    units:                    K
    wavelength:               10.8 µm (9.8-11.8 µm)
    standard_name:            toa_brightness_temperature
    platform_name:            Meteosat-11
    sensor:                   seviri
    georef_offset_corrected:  True
    start_time:               2021-03-02 12:30:11.584603
    end_time:                 2021-03-02 12:45:09.949762
    reader:                   seviri_l1b_native
    area:                     Area ID: msg_seviri_fes_3km\\nDescription: MSG S...
    name:                     IR_108
    resolution:               3000.403165817
    calibration:              brightness_temperature
    modifiers:                ()
    _satpy_id:                DataID(name='IR_108', wavelength=WavelengthRang...
    ancillary_variables:      []
```

References

- EUMETSAT Product Navigator
- MSG Level 1.5 Native Format File Definition

class satpy.readers.seviri_l1b_native.**ImageBoundaries**(*header*, *trailer*, *mda*)

Bases: object

Collect image boundary information.

Initialize the class.

static _check_for_valid_bounds(*img_bounds*)

static _convert_visir_bound_to_hrv(*bound*)

_get_hrv_actual_img_bounds()

Get HRV (if not ROI) image boundaries from the ActualL15CoverageHRV information stored in the trailer.

_get_hrv_img_shape()

_get_selected_img_bounds(*dataset_id*)

Get VISIR and HRV (if ROI) image boundaries from the SelectedRectangle information stored in the header.

_get_visir_img_shape()

get_img_bounds(*dataset_id*, *is_roi*)

Get image line and column boundaries.

Returns

Dictionary with the four keys 'south_bound', 'north_bound', 'east_bound' and 'west_bound', each containing a list of the respective line/column numbers of the image boundaries.

Lists (rather than scalars) are returned since the HRV data in FES mode contain data from two windows/areas.

class satpy.readers.seviri_l1b_native.**NativeMSGFileHandler**(*filename*, *filename_info*, *filetype_info*, *calib_mode='nominal'*, *fill_disk=False*, *ext_calib_coefs=None*, *include_raw_metadata=False*, *mda_max_array_size=100*)

Bases: *BaseFileHandler*

SEVIRI native format reader.

Calibration

See *satpy.readers.seviri_base*.

Padding channel data to full disk

By providing the *fill_disk* as True in the *reader_kwargs*, the channel is loaded as full disk, padded with no-data where necessary. This is especially useful for the HRV channel, but can also be used for RSS and ROI data. By default, the original, unpadded, data are loaded:

```
scene = satpy.Scene(filenames,
                    reader='seviri_l1b_native',
                    reader_kwargs={'fill_disk': False})
```

Metadata

See *satpy.readers.seviri_base*.

Initialize the reader.

_add_scanline_acq_time(*dataset*, *dataset_id*)

Add scanline acquisition time to the given dataset.

_get_acq_time_hrv()

Get raw acquisition time for HRV channel.

_get_acq_time_visir(*dataset_id*)

Get raw acquisition time for VIS/IR channels.

_get_calib_coefs(*channel_name*)

Get coefficients for calibration from counts to radiance.

_get_data_dtype()

Get the dtype of the file based on the actual available channels.

_get_hrv_channel()

_get_memmap()

Get the memory map for the SEVIRI data.

_get_orbital_parameters()

_get_visir_channel(*dataset_id*)

_read_header()

Read the header info.

_read_trailer()

property _repeat_cycle_duration

Get repeat cycle duration from the trailer.

_update_attrs(*dataset*, *dataset_info*)

Update dataset attributes.

calibrate(*data*, *dataset_id*)

Calibrate the data.

property end_time

Get the general end time for this file.

get_area_def(*dataset_id*)

Get the area definition of the band.

In general, image data from one window/area is available. For the HRV channel in FES mode, however, data from two windows ('Lower' and 'Upper') are available. Hence, we collect lists of area-extents and corresponding number of image lines/columns. In case of FES HRV data, two area definitions are computed, stacked and squeezed. For other cases, the lists will only have one entry each, from which a single area definition is computed.

Note that the AreaDefinition area extents returned by this function for Native data will be slightly different compared to the area extents returned by the SEVIRI HRIT reader. This is due to slightly different pixel size values when calculated using the data available in the files. E.g. for the 3 km grid:

```
Native:  data15hd['ImageDescription']['ReferenceGridVIS_IR']['ColumnDirGridStep']
== 3000.4031658172607   HRIT: np.deg2rad(2.**16 / pdict['lfac']) * pdict['h'] ==
3000.4032785810186
```

This results in the Native 3 km full-disk area extents being approx. 20 cm shorter in each direction.

The method for calculating the area extents used by the HRIT reader (CFAC/LFAC mechanism) keeps the highest level of numeric precision and is used as reference by EUM. For this reason, the standard area definitions defined in the *areas.yaml* file correspond to the HRIT ones.

get_area_extent(*dataset_id*)

Get the area extent of the file.

Until December 2017, the data is shifted by 1.5km SSP North and West against the nominal GEOS projection. Since December 2017 this offset has been corrected. A flag in the data indicates if the correction has been applied. If no correction was applied, adjust the area extent to match the shifted data.

For more information see Section 3.1.4.2 in the MSG Level 1.5 Image Data Format Description. The correction of the area extent is documented in a developer's memo.

get_dataset(*dataset_id*, *dataset_info*)

Get the dataset.

is_roi()

Check if data covers a selected region of interest (ROI).

Standard RSS data consists of 3712 columns and 1392 lines, covering the three northmost segments of the SEVIRI disk. Hence, if the data does not cover the full disk, nor the standard RSS region in RSS mode, it's assumed to be ROI data.

property nominal_end_time

Get the repeat cycle nominal end time from file header and round it to expected nominal time slot.

property nominal_start_time

Get the repeat cycle nominal start time from file header and round it to expected nominal time slot.

property observation_end_time

Get observation end time from trailer.

property observation_start_time

Get observation start time from trailer.

property satpos

Get actual satellite position in geodetic coordinates (WGS-84).

Evaluate orbit polynomials at the start time of the scan.

Returns: Longitude [deg east], Latitude [deg north] and Altitude [m]

property start_time

Get general start time for this file.

class satpy.readers.seviri_l1b_native.**Padder**(*dataset_id*, *img_bounds*, *is_full_disk*)

Bases: object

Padding of HRV, RSS and ROI data to full disk.

Initialize the padder.

_extract_data_to_pad(*dataset*, *south_bound*, *north_bound*)

Extract the data that shall be padded.

In case of FES (HRV) data, 'dataset' contains data from two separate windows that are padded separately. Hence, we extract a subset of data.

pad_data(*dataset*)

Pad data to full disk with empty pixels.

satpy.readers.seviri_l1b_native.**get_available_channels**(*header*)

Get the available channels from the header information.

satpy.readers.seviri_l1b_native.**has_archive_header**(*filename*)

Check whether the file includes an ASCII archive header.

satpy.readers.seviri_l1b_native.**read_header**(*filename*)

Read SEVIRI L1.5 native header.

satpy.readers.seviri_l1b_native_hdr module

Header and trailer records of SEVIRI native format.

satpy.readers.seviri_l1b_native_hdr.**DEFAULT_15_SECONDARY_PRODUCT_HEADER** = {'EastColumnSelectedRectangle': {'Value': 1}, 'NorthLineSelectedRectangle': {'Value': 3712}, 'NumberColumnsHRV': {'Value': 11136}, 'NumberColumnsVISIR': {'Value': 3712}, 'NumberLinesHRV': {'Value': 11136}, 'NumberLinesVISIR': {'Value': 3712}, 'SelectedBandIDs': {'Value': 'XXXXXXXXXXXX'}, 'SouthLineSelectedRectangle': {'Value': 1}, 'WestColumnSelectedRectangle': {'Value': 3712}}

Default secondary product header for files containing all channels.

class satpy.readers.seviri_l1b_native_hdr.**GSDTRecords**

Bases: object

MSG Ground Segment Data Type records.

Reference Document (EUM/MSG/SPE/055): MSG Ground Segment Design Specification (GSDS)

gp_cpu_address = [('Qualifier_1', <class 'numpy.uint8'>), ('Qualifier_2', <class 'numpy.uint8'>), ('Qualifier_3', <class 'numpy.uint8'>), ('Qualifier_4', <class 'numpy.uint8'>)]

gp_fac_env

alias of uint8

gp_fac_id

alias of uint8

gp_pk_header = [('HeaderVersionNo', <class 'numpy.uint8'>), ('PacketType', <class 'numpy.uint8'>), ('SubHeaderType', <class 'numpy.uint8'>), ('SourceFacilityId', <class 'numpy.uint8'>), ('SourceEnvId', <class 'numpy.uint8'>), ('SourceInstanceId', <class 'numpy.uint8'>), ('SourceSUId', <class 'numpy.uint32'>), ('SourceCPUId', [('Qualifier_1', <class 'numpy.uint8'>), ('Qualifier_2', <class 'numpy.uint8'>), ('Qualifier_3', <class 'numpy.uint8'>), ('Qualifier_4', <class 'numpy.uint8'>)]), ('DestFacilityId', <class 'numpy.uint8'>), ('DestEnvId', <class 'numpy.uint8'>), ('SequenceCount', <class 'numpy.uint16'>), ('PacketLength', <class 'numpy.int32'>)]

gp_pk_sh1 = [('SubHeaderVersionNo', <class 'numpy.uint8'>), ('ChecksumFlag', <class 'bool'>), ('Acknowledgement', (<class 'numpy.uint8'>, 4)), ('ServiceType', <class 'numpy.uint8'>), ('ServiceSubtype', <class 'numpy.uint8'>), ('PacketTime', [('Days', '>u2'), ('Milliseconds', '>u4')]), ('SpacecraftId', <class 'numpy.uint16'>)]

gp_sc_id

alias of uint16

gp_su_id

alias of uint32

gp_svce_type

alias of uint8

class satpy.readers.seviri_l1b_native_hdr.**HritPrologue**

Bases: *L15DataHeaderRecord*

HRIT Prologue handler.

get()

Get record data array.

class satpy.readers.seviri_l1b_native_hdr.**L15DataHeaderRecord**

Bases: object

L15 Data Header handler.

Reference Document (EUM/MSG/ICD/105): MSG Level 1.5 Image Data Format Description

property celestial_events

Get celestial events data.

property geometric_processing

Get geometric processing data.

get()

Get header record data.

property image_acquisition

Get image acquisition data.

property image_description

Get image description data.

property impf_configuration

Get impf configuration information.

property radiometric_processing

Get radiometric processing data.

property satellite_status

Get satellite status data.

class satpy.readers.seviri_l1b_native_hdr.**L15MainProductHeaderRecord**

Bases: object

L15 Main Product header handler.

Reference Document: MSG Level 1.5 Native Format File Definition

get()

Get header data.

class satpy.readers.seviri_l1b_native_hdr.**L15PhData**

Bases: object

L15 Ph handler.

l15_ph_data = [('Name', 'S30'), ('Value', 'S50')]

class satpy.readers.seviri_l1b_native_hdr.**L15SecondaryProductHeaderRecord**

Bases: object

L15 Secondary Product header handler.

Reference Document: MSG Level 1.5 Native Format File Definition

get()

Get header data.

class satpy.readers.seviri_l1b_native_hdr.**Msg15NativeHeaderRecord**

Bases: object

SEVIRI Level 1.5 header for native-format.

get(*with_archive_header*)

Get the header type.

class satpy.readers.seviri_l1b_native_hdr.**Msg15NativeTrailerRecord**

Bases: object

SEVIRI Level 1.5 trailer for native-format.

Reference Document (EUM/MSG/ICD/105): MSG Level 1.5 Image Data Format Description

property geometric_quality

Get geometric quality record data.

get()

Get header record data.

property image_production_stats

Get image production statistics.

property navigation_extraction_results

Get navigation extraction data.

property radiometric_quality

Get radiometric quality record data.

property seviri_l15_trailer

Get file trailer data.

property timeliness_and_completeness

Get time and completeness record data.

satpy.readers.seviri_l1b_native_hdr.**get_native_header**(*with_archive_header=True*)

Get Native format header type.

There are two variants, one including an ASCII archive header and one without that header. The header is prepended if the data are ordered through the EUMETSAT data center.

satpy.readers.seviri_l1b_nc module

SEVIRI netcdf format reader.

class satpy.readers.seviri_l1b_nc.**NCSEVIRIFileHandler**(*filename*, *filename_info*, *filetype_info*, *ext_calib_coefs=None*, *mask_bad_quality_scan_lines=True*)

Bases: *BaseFileHandler*

File handler for NC seviri files.

Calibration

See *satpy.readers.seviri_base*. Note that there is only one set of calibration coefficients available in the netCDF files and therefore there is no *calib_mode* argument.

Metadata

See *satpy.readers.seviri_base*.

Init the file handler.

_add_scanline_acq_time(*dataset*, *dataset_id*)

_get_acq_time_hrv()

_get_acq_time_visir(*dataset_id*)

_get_calib_coefs(*dataset*, *channel*)

Get coefficients for calibration from counts to radiance.

_get_earth_model()

_mask_bad_quality(*dataset*, *dataset_info*)

Mask scanlines with bad quality.

property _repeat_cycle_duration

Get repeat cycle duration from the metadata.

_update_attrs(*dataset*, *dataset_info*)

Update dataset attributes.

calibrate(*dataset*, *dataset_id*)

Calibrate the data.

property end_time

Get the general end time for this file.

get_area_def(*dataset_id*)

Get the area def.

Note that the AreaDefinition area extents returned by this function for NetCDF data will be slightly different compared to the area extents returned by the SEVIRI HRIT reader. This is due to slightly different pixel size values when calculated using the data available in the files. E.g. for the 3 km grid:

```
NetCDF: self.nc.attrs['vis_ir_column_dir_grid_step'] == 3000.4031658172607 HRIT:
np.deg2rad(2.**16 / pdict['lfac']) * pdict['h'] == 3000.4032785810186
```

This results in the Native 3 km full-disk area extents being approx. 20 cm shorter in each direction.

The method for calculating the area extents used by the HRIT reader (CFAC/LFAC mechanism) keeps the highest level of numeric precision and is used as reference by EUM. For this reason, the standard area definitions defined in the *areas.yaml* file correspond to the HRIT ones.

get_area_extent(*dsid*)

Get the area extent.

get_dataset(*dataset_id*, *dataset_info*)

Get the dataset.

get_metadata()

Get metadata.

property nc

Read the file.

property nominal_end_time

Read the repeat cycle nominal end time from metadata and round it to expected nominal time slot.

property nominal_start_time

Read the repeat cycle nominal start time from metadata and round it to expected nominal time slot.

property observation_end_time

Get the repeat cycle observation end time from metadata.

property observation_start_time

Get the repeat cycle observation start time from metadata.

property satpos

Get actual satellite position in geodetic coordinates (WGS-84).

Evaluate orbit polynomials at the start time of the scan.

Returns: Longitude [deg east], Latitude [deg north] and Altitude [m]

property start_time

Get general start time for this file.

class satpy.readers.seviri_l1b_nc.**NCSEVIRIHRVFileHandler**(*filename*, *filename_info*, *filetype_info*, *ext_calib_coefs=None*, *mask_bad_quality_scan_lines=True*)

Bases: *NCSEVIRIFileHandler*, *SEVIRICalibrationHandler*

HRV filehandler.

Init the file handler.

get_area_extent(*dsid*)

Get HRV area extent.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset from file.

satpy.readers.seviri_l2_bufr module

SEVIRI L2 BUFR format reader.

References

EUMETSAT Product Navigator https://navigator.eumetsat.int/

class satpy.readers.seviri_l2_bufr.**SeviriL2BufrFileHandler**(*filename*, *filename_info*, *filetype_info*, *with_area_definition=False*, *rectification_longitude='default'*, ***kwargs*)

Bases: *BaseFileHandler*

File handler for SEVIRI L2 BUFR products.

Loading data with AreaDefinition

By providing the *with_area_definition* as True in the *reader_kwargs*, the dataset is loaded with an AreaDefinition using a standardized AreaDefinition in areas.yaml. By default, the dataset will be loaded with a SwathDefinition, i.e. similar to how the data are stored in the BUFR file:

scene = satpy.Scene(filenames,
reader='seviri_l2_bufr', reader_kwargs={'with_area_definition': False})

Defining dataset recticifation longitude

The BUFR data were originally extracted from a rectified two-dimensional grid with a given central longitude (typically the sub-satellite point). This information is not available in the file itself nor the filename (for files from the EUMETSAT archive). Also, it cannot be reliably derived from all datasets themselves. Hence, the rectification longitude can be defined by the user by providing *rectification_longitude* in the *reader_kwargs*:

scene = satpy.Scene(filenames,
reader='seviri_l2_bufr', reader_kwargs={'rectification_longitude': 0.0})

If not done, default values applicable to the operational grids of the respective SEVIRI instruments will be used.

Initialise the file handler for SEVIRI L2 BUFR data.

_add_attributes(*xarr*, *dataset_info*)

Add dataset attributes to xarray.

_construct_area_def(*dataset_id*)

Construct a standardized AreaDefinition based on satellite, instrument, resolution and sub-satellite point.

Returns
A pyresample AreaDefinition object containing the area definition.

Return type
AreaDefinition

_read_mpef_header()

Read MPEF header.

property end_time

Return the repeat cycle end time.

get_area_def(*key*)

Return the area definition.

get_array(*key*)

Get all data from file for the given BUFR key.

get_attribute(*key*)

Get BUFR attributes.

get_dataset(*dataset_id*, *dataset_info*)

Create dataset.

Load data from BUFR file using the BUFR key in dataset_info and create the dataset with or without an AreaDefinition.

get_dataset_with_area_def(*arr*, *dataset_id*)

Get dataset with an AreaDefinition.

property platform_name

Return spacecraft name.

property ssp_lon

Return subsatellite point longitude.

property start_time

Return the repeat cycle start time.

satpy.readers.seviri_l2_grib module

Reader for the SEVIRI L2 products in GRIB2 format.

References

FM 92 GRIB Edition 2 https://www.wmo.int/pages/prog/www/WMOCodes/Guides/GRIB/GRIB2_062006.pdf EUMETSAT Product Navigator https://navigator.eumetsat.int/

class satpy.readers.seviri_l2_grib.**SeviriL2GribFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Reader class for SEVIRI L2 products in GRIB format.

Read the global attributes and prepare for dataset reading.

_get_attributes()

Create a dictionary of attributes to be added to the dataset.

Returns

A dictionary of parameter attributes.
ssp_lon: longitude of subsatellite point sensor: name of sensor platform_name: name of the platform

Return type
dict

static _get_from_msg(*gid*, *key*)

Get a value from the GRIB message based on the key, return None if missing.

Parameters

- **gid** – The ID of the GRIB message.
- **key** – The key of the required attribute.

Returns
The retrieved attribute or None if the key is missing.

_get_proj_area(*gid*)

Compute the dictionary with the projection and area definition from a GRIB message.

Parameters
gid – The ID of the GRIB message.

Returns

A tuple of two dictionaries for the projection and the area definition.

pdict:
a: Earth major axis [m] b: Earth minor axis [m] h: Height over surface [m] ssp_lon: longitude of subsatellite point [deg] nlines: number of lines ncols: number of columns a_name: name of the area a_desc: description of the area p_id: id of the projection

area_dict:
center_point: coordinate of the center point north: coodinate of the north limit east: coodinate of the east limit west: coodinate of the west limit south: coodinate of the south limit

Return type
tuple

_get_xarray_from_msg(*gid*)

Read the values from the GRIB message and return a DataArray object.

Parameters
gid – The ID of the GRIB message.

Returns
The array containing the retrieved values.

Return type
DataArray

_read_attributes(*gid*)

Read the parameter attributes from the message and create the projection and area dictionaries.

static _scale_earth_axis(*data*)

Scale Earth axis data to make sure the value matched the expected unit [m].

The earthMinorAxis value stored in the aerosol over sea product is scaled incorrectly by a factor of 1e8. This method provides a flexible temporarily workaraound by making sure that all earth axis values are scaled such that they are on the order of millions of meters as expected by the reader. As soon as the scaling issue has been resolved by EUMETSAT this workaround can be removed.

property end_time

Return the sensing end time.

get_area_def(*dataset_id*)

Return the area definition for a dataset.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using the parameter_number key in dataset_info.

In a previous version of the reader, the attributes (nrows, ncols, ssp_lon) and projection information (pdict and area_dict) were computed while initializing the file handler. Also the code would break out from the While-loop below as soon as the correct parameter_number was found. This has now been revised becasue the reader would sometimes give corrupt information about the number of messages in the file and the dataset dimensions within a given message if the file was only partly read (not looping over all messages) in an earlier instance.

property start_time

Return the sensing start time.

satpy.readers.slstr_l1b module

SLSTR L1b reader.

class satpy.readers.slstr_l1b.**NCSLSTR1B**(*filename*, *filename_info*, *filetype_info*, *user_calibration=None*)

Bases: *BaseFileHandler*

Filehandler for l1 SLSTR data.

By default, the calibration factors recommended by EUMETSAT are applied. This is required as the SLSTR VIS channels are producing slightly incorrect radiances that require adjustment. Satpy uses the radiance corrections in S3.PN-SLSTR-L1.08, checked 11/03/2022. User-supplied coefficients can be passed via the *user_calibration* kwarg This should be a dict of channel names (such as *S1_nadir*, *S8_oblique*).

For example:

```
calib_dict = {'S1_nadir': 1.12}
scene = satpy.Scene(filenames,
```

(continues on next page)

(continued from previous page)

```
                    reader='slstr-l1b',
                    reader_kwargs={'user_calib': calib_dict})
```

Will multiply S1 nadir radiances by 1.12.

Initialize the SLSTR l1 data filehandler.

_apply_radiance_adjustment(*radiances*)

Adjust SLSTR radiances with default or user supplied values.

static _cal_rad(*rad*, *didx*, *solar_flux=None*)

Calibrate.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Get the start time.

class satpy.readers.slstr_l1b.**NCSLSTRAngles**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Filehandler for angles.

Initialize the angles reader.

_loadcart(*fname*)

Load a cartesian file of appropriate type.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Get the start time.

class satpy.readers.slstr_l1b.**NCSLSTRFlag**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

File handler for flags.

Initialize the flag reader.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Get the start time.

class satpy.readers.slstr_l1b.**NCSLSTRGeo**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Filehandler for geo info.

Initialize the geo filehandler.

property end_time

Get the end time.

get_dataset(*key*, *info*)

Load a dataset.

property start_time

Get the start time.

satpy.readers.smos_l2_wind module

SMOS L2 wind Reader.

Data can be found here after register: https://www.smosstorm.org/Data2/SMOS-NRT-wind-Products-access Format documentation at the same site after register: SMOS_WIND_DS_PDD_20191107_signed.pdf

class satpy.readers.smos_l2_wind.**SMOSL2WINDFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

File handler for SMOS L2 wind netCDF files.

Initialize object.

_adjust_lon_coord(*data*)

Adjust lon coordinate to -180 .. 180 (not 0 .. 360).

_create_area_extent(*width*, *height*)

Create area extent.

_mask_dataset(*data*)

Mask out fill values.

_remove_time_coordinate(*data*)

Remove time coordinate.

_rename_coords(*data*)

Rename coords.

_roll_dataset_lon_coord(*data*)

Roll dataset along the lon coordinate.

available_datasets(*configured_datasets=None*)

Automatically determine datasets provided by this file.

property end_time

Get end time.

get_area_def(*dsid*)

Define AreaDefintion.

get_dataset(*ds_id*, *ds_info*)

Get dataset.

get_metadata(*data*, *ds_info*)

Get metadata.

property platform_name

Get platform.

property platform_shortname

Get platform shortname.

property start_time

Get start time.

satpy.readers.tropomi_l2 module

Interface to TROPOMI L2 Reader.

The TROPOspheric Monitoring Instrument (TROPOMI) is the satellite instrument on board the Copernicus Sentinel-5 Precursor satellite. It measures key atmospheric trace gasses, such as ozone, nitrogen oxides, sulfur dioxide, carbon monoxide, methane, and formaldehyde.

Level 2 data products are available via the Copernicus Open Access Hub. For more information visit the following URL: http://www.tropomi.eu/data-products/level-2-products

class satpy.readers.tropomi_l2.**TROPOMIL2FileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

File handler for TROPOMI L2 netCDF files.

Initialize object.

_iterate_over_dataset_contents(*handled_variables*, *shape*)

Iterate over dataset contents.

This is where we dynamically add new datasets We will sift through all groups and variables, looking for data matching the geolocation bounds

_rename_dims(*data_arr*)

Normalize dimension names with the rest of Satpy.

available_datasets(*configured_datasets=None*)

Automatically determine datasets provided by this file.

property end_time

Get end time.

get_dataset(*ds_id*, *ds_info*)

Get dataset.

get_metadata(*data*, *ds_info*)

Get metadata.

property platform_shortname

Get platform shortname.

prepare_geo(*bounds_data*)

Prepare lat/lon bounds for pcolormesh.

lat/lon bounds are ordered in the following way:

```
3----2
|    |
0----1
```

Extend longitudes and latitudes with one element to support "pcolormesh":

```
(X[i+1, j], Y[i+1, j])         (X[i+1, j+1], Y[i+1, j+1])
                       +--------+
                       | C[i,j] |
                       +--------+
     (X[i, j], Y[i, j])        (X[i, j+1], Y[i, j+1])
```

property sensor

Get sensor.

property sensor_names

Get sensor set.

property start_time

Get start time.

property time_coverage_end

Get time_coverage_end.

property time_coverage_start

Get time_coverage_start.

satpy.readers.utils module

Helper functions for satpy readers.

satpy.readers.utils.**_get_geostationary_height**(*geos_area*)

satpy.readers.utils.**_get_geostationary_reference_longitude**(*geos_area*)

satpy.readers.utils.**_get_geostationary_semi_axes**(*geos_area*)

satpy.readers.utils.**_lonlat_from_geos_angle**(*x*, *y*, *geos_area*)

Get lons and lats from x, y in projection coordinates.

satpy.readers.utils.**_unzip_FSFile**(*filename:* FSFile, *prefix=None*)

Open and Unzip remote FSFile ending with 'bz2'.

Parameters

- **filename** – The FSFile to unzip.

- **prefix** (*str, optional*) – If file is one of many segments of data, prefix random filename
- **number.** (*for correct sorting. This is normally the segment*) –

Returns
Temporary filename path for decompressed file or None.

satpy.readers.utils.**_unzip_local_file**(*filename: str*, *prefix=None*)

Unzip the file ending with 'bz2'. Initially with pbzip2 if installed or bz2.

Parameters

- **filename** – The file to unzip.
- **prefix** (*str, optional*) – If file is one of many segments of data, prefix random filename
- **number.** (*for correct sorting. This is normally the segment*) –

Returns
Temporary filename path for decompressed file or None.

satpy.readers.utils.**_unzip_with_bz2**(*filename*, *tmpfilepath*)

satpy.readers.utils.**_unzip_with_pbzip**(*filename*, *tmpfilepath*, *fdn*)

satpy.readers.utils.**_write_uncompressed_file**(*content*, *fdn*, *filename*, *tmpfilepath*)

satpy.readers.utils.**apply_earthsun_distance_correction**(*reflectance*, *utc_date=None*)

Correct reflectance data to account for changing Earth-Sun distance.

satpy.readers.utils.**apply_rad_correction**(*data*, *slope*, *offset*)

Apply GSICS-like correction factors to radiance data.

satpy.readers.utils.**bbox**(*img*)

Find the bounding box around nonzero elements in the given array.

Copied from https://stackoverflow.com/a/31402351/5703449 .

Returns
rowmin, rowmax, colmin, colmax

satpy.readers.utils.**generic_open**(*filename*, **args*, ***kwargs*)

Context manager for opening either a regular file or a bzip2 file.

Returns a file-like object.

satpy.readers.utils.**get_array_date**(*scn_data*, *utc_date=None*)

Get start time from a channel data array.

satpy.readers.utils.**get_earth_radius**(*lon*, *lat*, *a*, *b*)

Compute radius of the earth ellipsoid at the given longitude and latitude.

Parameters

- **lon** – Geodetic longitude (degrees)
- **lat** – Geodetic latitude (degrees)
- **a** – Semi-major axis of the ellipsoid (meters)
- **b** – Semi-minor axis of the ellipsoid (meters)

Returns
Earth Radius (meters)

satpy.readers.utils.**get_geostationary_angle_extent**(*geos_area*)

Get the max earth (vs space) viewing angles in x and y.

satpy.readers.utils.**get_geostationary_bounding_box**(*geos_area*, *nb_points=50*)

Get the bbox in lon/lats of the valid pixels inside *geos_area*.

Parameters

- **geos_area** – The geostationary area to analyse.
- **nb_points** – Number of points on the polygon

satpy.readers.utils.**get_geostationary_mask**(*area*, *chunks=None*)

Compute a mask of the earth's shape as seen by a geostationary satellite.

Parameters

- **area** (*pyresample.geometry.AreaDefinition*) – Corresponding area definition
- **chunks** (*int or tuple*) – Chunk size for the 2D array that is generated.

Returns

Boolean mask, True inside the earth's shape, False outside.

satpy.readers.utils.**get_sub_area**(*area*, *xslice*, *yslice*)

Apply slices to the area_extent and size of the area.

satpy.readers.utils.**get_user_calibration_factors**(*band_name*, *correction_dict*)

Retrieve radiance correction factors from user-supplied dict.

satpy.readers.utils.**np2str**(*value*)

Convert an *numpy.string_* to str.

Parameters

value (*ndarray*) – scalar or 1-element numpy array to convert

Raises

ValueError – if value is array larger than 1-element, or it is not of type *numpy.string_* or it is not a numpy array

satpy.readers.utils.**reduce_mda**(*mda*, *max_size=100*)

Recursively remove arrays with more than *max_size* elements from the given metadata dictionary.

satpy.readers.utils.**remove_earthsun_distance_correction**(*reflectance*, *utc_date=None*)

Remove the sun-earth distance correction.

satpy.readers.utils.**unzip_context**(*filename*)

Context manager for decompressing a .bz2 file on the fly.

Uses *unzip_file*. Removes the uncompressed file on exit of the context manager.

Returns: the filename of the uncompressed file or of the original file if it was not compressed.

satpy.readers.utils.**unzip_file**(*filename: str* | FSFile, *prefix=None*)

Unzip the local/remote file ending with 'bz2'.

Parameters

- **filename** – The local/remote file to unzip.
- **prefix** (*str, optional*) – If file is one of many segments of data, prefix random filename
- **number.** (*for correct sorting. This is normally the segment*) –

Returns
Temporary filename path for decompressed file or None.

satpy.readers.vaisala_gld360 module

Vaisala Global Lightning Dataset 360 reader.

Vaisala Global Lightning Dataset GLD360 is data as a service that provides real-time lightning data for accurate and early detection and tracking of severe weather. The data provided is generated by a Vaisala owned and operated world-wide lightning detection sensor network.

References: - [GLD360] https://www.vaisala.com/en/products/data-subscriptions-and-reports/data-sets/gld360

class satpy.readers.vaisala_gld360.**VaisalaGLD360TextFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

ASCII reader for Vaisala GDL360 data.

Initialize VaisalaGLD360TextFileHandler.

property end_time
Get end time.

get_dataset(*dataset_id*, *dataset_info*)
Load a dataset.

property start_time
Get start time.

satpy.readers.vii_base_nc module

EUMETSAT EPS-SG Visible/Infrared Imager (VII) readers base class.

class satpy.readers.vii_base_nc.**ViiNCBaseFileHandler**(*filename*, *filename_info*, *filetype_info*, *orthorect=False*)

Bases: *NetCDF4FileHandler*

Base reader class for VII products in netCDF format.

Parameters

- **filename** (*str*) – File to read
- **filename_info** (*dict*) – Dictionary with filename information
- **filetype_info** (*dict*) – Dictionary with filetype information
- **orthorect** (*bool*) – activates the orthorectification correction where available

Prepare the class for dataset reading.

_get_global_attributes()
Create a dictionary of global attributes to be added to all datasets.

_perform_calibration(*variable*, *dataset_info*)
Perform the calibration.

static _perform_geo_interpolation(*longitude*, *latitude*)

Perform the interpolation of geographic coodinates from tie points to pixel points.

Parameters

- **longitude** – xarray DataArray containing the longitude dataset to interpolate.
- **latitude** – xarray DataArray containing the longitude dataset to interpolate.

Returns

tuple of arrays containing the interpolate values, all the original metadata and the updated dimension names.

static _perform_interpolation(*variable*)

Perform the interpolation from tie points to pixel points.

Parameters

variable – xarray DataArray containing the dataset to interpolate.

Returns

array containing the interpolate values, all the original metadata and the updated dimension names.

Return type

DataArray

_perform_orthorectification(*variable*, *orthorect_data_name*)

Perform the orthorectification.

_standardize_dims(*variable*)

Standardize dims to y, x.

property end_time

Get observation end time.

get_dataset(*dataset_id*, *dataset_info*)

Get dataset using file_key in dataset_info.

property sensor

Return sensor.

property spacecraft_name

Return spacecraft name.

property ssp_lon

Return subsatellite point longitude.

property start_time

Get observation start time.

satpy.readers.vii_l1b_nc module

EUMETSAT EPS-SG Visible/Infrared Imager (VII) Level 1B products reader.

The `vii_l1b_nc` reader reads and calibrates EPS-SG VII L1b image data in netCDF format. The format is explained in the EPS-SG VII Level 1B Product Format Specification V4A.

This version is applicable for the vii test data V2 to be released in Jan 2022.

class satpy.readers.vii_l1b_nc.**ViiL1bNCFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *ViiNCBaseFileHandler*

Reader class for VII L1B products in netCDF format.

Read the calibration data and prepare the class for dataset reading.

static _calibrate_bt(*radiance*, *cw*, *a*, *b*)

Perform the calibration to brightness temperature.

Parameters

- **radiance** – numpy ndarray containing the radiance values.
- **cw** – center wavelength [m].
- **a** – temperature coefficient [-].
- **b** – temperature coefficient [K].

Returns
array containing the calibrated brightness temperature values.

Return type
numpy ndarray

static _calibrate_refl(*radiance*, *angle_factor*, *isi*)

Perform the calibration to reflectance.

Parameters

- **radiance** – numpy ndarray containing the radiance values.
- **angle_factor** – numpy ndarray containing the inverse of cosine of solar zenith angle [-].
- **isi** – integrated solar irradiance [W/(m2 * m)].

Returns
array containing the calibrated reflectance values.

Return type
numpy ndarray

_perform_calibration(*variable*, *dataset_info*)

Perform the calibration.

Parameters

- **variable** – xarray DataArray containing the dataset to calibrate.
- **dataset_info** – dictionary of information about the dataset.

Returns
array containing the calibrated values and all the original metadata.

Return type
DataArray

_perform_orthorectification(*variable*, *orthorect_data_name*)

Perform the orthorectification.

Parameters

- **variable** – xarray DataArray containing the dataset to correct for orthorectification.
- **orthorect_data_name** – name of the orthorectification correction data in the product.

Returns
array containing the corrected values and all the original metadata.

Return type
DataArray

satpy.readers.vii_l2_nc module

EUMETSAT EPS-SG Visible/Infrared Imager (VII) Level 2 products reader.

class satpy.readers.vii_l2_nc.**ViiL2NCFileHandler**(*filename*, *filename_info*, *filetype_info*, *orthorect=False*)

Bases: *ViiNCBaseFileHandler*

Reader class for VII L2 products in netCDF format.

Prepare the class for dataset reading.

_perform_orthorectification(*variable*, *orthorect_data_name*)

Perform the orthorectification.

Parameters

- **variable** – xarray DataArray containing the dataset to correct for orthorectification.
- **orthorect_data_name** – name of the orthorectification correction data in the product.

Returns
array containing the corrected values and all the original metadata.

Return type
DataArray

satpy.readers.vii_utils module

Utilities for the management of VII products.

satpy.readers.viirs_atms_sdr_base module

Common utilities for reading VIIRS and ATMS SDR data.

class satpy.readers.viirs_atms_sdr_base.**JPSS_SDR_FileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDF5FileHandler*

Base class for reading JPSS VIIRS & ATMS SDR HDF5 Files.

Initialize file handler.

_adjust_scaling_factors(*factors*, *file_units*, *output_units*)

Adjust scaling factors .

_generate_file_key(*ds_id*, *ds_info*, *factors=False*)

_get_aggr_path(*fileinfo_key*, *aggr_default*)

_get_rows_per_granule(*dataset_group*)

_get_scans_per_granule(*dataset_group*)

static _get_valid_scaling_factors(*factors*)

_get_variable(*var_path*, ***kwargs*)

static _map_and_apply_factors(*data*, *factors*, *rows_per_gran*)

static _mask_and_reshape_factors(*factors*)

_parse_datetime(*datestr*, *timestr*)

static _scale_factors_for_units(*factors*, *file_units*, *output_units*)

_scan_size(*dataset_group_name*)

Get how many rows of data constitute one scanline.

_update_data_attributes(*data*, *dataset_id*, *ds_info*)

available_datasets(*configured_datasets=None*)

Generate dataset info and their availablity.

See `satpy.readers.file_handlers.BaseFileHandler.available_datasets()` for details.

concatenate_dataset(*dataset_group*, *var_path*, ***kwargs*)

Concatenate dataset.

property end_orbit_number

Get end orbit number.

property end_time

Get end time.

static expand_single_values(*var*, *scans*)

Expand single valued variable to full scan lengths.

mask_fill_values(*data*, *ds_info*)

Mask fill values.

property platform_name

Get platform name.

scale_data_to_specified_unit(*data*, *dataset_id*, *ds_info*)

Get sscale and offset factors and convert/scale data to given physical unit.

scale_swath_data(*data*, *scaling_factors*, *dataset_group*)

Scale swath data using scaling factors and offsets.

Multi-granule (a.k.a. aggregated) files will have more than the usual two values.

property sensor_name

Get sensor name.

property start_orbit_number

Get start orbit number.

property start_time

Get start time.

satpy.readers.viirs_atms_sdr_base.**_apply_factors**(*data*, *factor_set*)

satpy.readers.viirs_atms_sdr_base.**_get_file_units**(*dataset_id*, *ds_info*)

Get file units from metadata.

satpy.readers.viirs_atms_sdr_base.**_get_scale_factors_for_units**(*factors*, *file_units*, *output_units*)

satpy.readers.viirs_compact module

Compact viirs format.

This is a reader for the Compact VIIRS format shipped on Eumetcast for the VIIRS SDR. The format is compressed in multiple ways, notably by shipping only tie-points for geographical data. The interpolation of this data is done using dask operations, so it should be relatively performant.

For more information on this format, the reader can refer to the *Compact VIIRS SDR Product Format User Guide* that can be found on this EARS page.

class satpy.readers.viirs_compact.**VIIRSCompactFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

A file handler class for VIIRS compact format.

Initialize the reader.

_get_geographical_chunks()

angles(*azi_name*, *zen_name*)

Generate the angle datasets.

property end_time

Get the end time.

expand_angle_and_nav(*arrays*)

Expand angle and navigation datasets.

property expansion_coefs

Compute the expansion coefficients.

get_bounding_box()

Get the bounding box of the data.

get_dataset(*key*, *info*)

Load a dataset.

navigate()

Generate the navigation datasets.

read_dataset(*dataset_key*, *info*)

Read a dataset.

read_geo(*key*, *info*)

Read angles.

property start_time

Get the start time.

satpy.readers.viirs_compact.**_interpolate_data**(*data*, *corner_coefficients*, *scans*)

Interpolate the data using the provided coefficients.

satpy.readers.viirs_compact.**convert_from_angles**(*azi*, *zen*)

Convert the angles to cartesian coordinates.

satpy.readers.viirs_compact.**convert_to_angles**(*x*, *y*, *z*)

Convert the cartesian coordinates to angles.

satpy.readers.viirs_compact.**expand**(*data*, *coefs*, *scans*, *scan_size*)

Perform the expansion in numpy domain.

satpy.readers.viirs_compact.**expand_arrays**(*arrays*, *scans*, *c_align*, *c_exp*, *scan_size=16*, *tpz_size=16*, *nties=200*, *track_offset=0.5*, *scan_offset=0.5*)

Expand *data* according to alignment and expansion.

satpy.readers.viirs_compact.**get_coefs**(*c_align*, *c_exp*, *tpz_size*, *nb_tpz*, *v_track*, *scans*, *scan_size*, *scan_offset*)

Compute the coeffs in numpy domain.

satpy.readers.viirs_edr module

VIIRS NOAA enterprise EDR product reader.

This module defines the `VIIRSJRRFileHandler` file handler, to be used for reading VIIRS EDR products generated by the NOAA enterprise suite, which are downloadable via NOAA CLASS or on NOAA's AWS buckets.

A wide variety of such products exist and, at present, only a subset are supported.

- Cloud mask: JRR-CloudMask_v2r3_j01_s202112250807275_e202112250808520_c202112250837300.nc
- Cloud products: JRR-CloudHeight_v2r3_j01_s202112250807275_e202112250808520_c202112250837300.nc
- Aerosol detection: JRR-ADP_v2r3_j01_s202112250807275_e202112250808520_c202112250839550.nc
- Aerosol optical depth: JRR-AOD_v2r3_j01_s202112250807275_e202112250808520_c202112250839550.nc
- Surface reflectance: SurfRefl_v1r1_j01_s202112250807275_e202112250808520_c202112250845080.nc
- Land Surface Temperature: LST_v2r0_npp_s202307241724558_e202307241726200_c202307241854058.nc

All products use the same base reader `viirs_edr` and can be read through satpy with:

```
import satpy
import glob

filenames = glob.glob('JRR-ADP*.nc')
scene = satpy.Scene(filenames, reader='viirs_edr')
scene.load(['smoke_concentration'])
```

Note: Multiple products contain datasets with the same name! For example, both the cloud mask and aerosol detection files contain a cloud mask, but these are not identical. For clarity, the aerosol file cloudmask is named *cloud_mask_adp* in this reader.

Vegetation Indexes

The NDVI and EVI products can be loaded from CSPP-produced Surface Reflectance files. By default, these products are filtered based on the Surface Reflectance Quality Flags. This is used to remove/mask pixels in certain cloud or water regions. This behavior can be disabled by providing the reader keyword argument `filter_veg` and setting it to `False`. For example:

```
scene = satpy.Scene(filenames, reader='viirs_edr', reader_kwargs={"filter_veg": False})
```

class satpy.readers.viirs_edr.**VIIRSJRRFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

NetCDF4 reader for VIIRS Active Fires.

Initialize the geo filehandler.

static _decode_flag_meanings(*data_arr: DataArray*)

_dynamic_variables_from_file(*handled_var_names: set*) → Iterable[tuple[bool, dict]]

_mask_invalid(*data_arr: DataArray*, *ds_info: dict*) → DataArray

available_datasets(*configured_datasets=None*)

Get information of available datasets in this file.

Parameters

configured_datasets (*list*) – Series of (bool or None, dict) in the same way as is returned by this method (see below). The bool is whether the dataset is available from at least one of the current file handlers. It can also be `None` if no file handler before us knows how to handle it. The dictionary is existing dataset metadata. The dictionaries are typically provided from a YAML configuration file and may be modified, updated, or used as a "template" for additional available datasets. This argument could be the result of a previous file handler's implementation of this method.

Returns

Iterator of (bool or None, dict) pairs where dict is the dataset's metadata. If the dataset is available in the current file type then the boolean value should be `True`, `False` if we **know** about the dataset but it is unavailable, or `None` if this file object is not responsible for it.

property end_time

Get last date/time when observations were recorded.

get_dataset(*dataset_id: DataID*, *info: dict*) → DataArray

Get the dataset.

property platform_name

Get platform name.

rows_per_scans(*data_arr: DataArray*) → int

Get number of array rows per instrument scan based on data resolution.

property start_time

Get first date/time when observations were recorded.

class satpy.readers.viirs_edr.**VIIRSLSTHandler**(**args*, ***kwargs*)

Bases: *VIIRSJRRFileHandler*

File handler to handle LST file scale factor and offset weirdness.

Initialize the file handler and unscale necessary variables.

_manual_scalings = {'Satellite_Azimuth_Angle': ('AZI_ScaleFact', 'AZI_Offset'), 'VLST': ('LST_ScaleFact', 'LST_Offset'), 'emis_bbe': ('LSE_ScaleFact', 'LSE_Offset'), 'emis_m15': ('LSE_ScaleFact', 'LSE_Offset'), 'emis_m16': ('LSE_ScaleFact', 'LSE_Offset')}

_scale_data()

class satpy.readers.viirs_edr.**VIIRSSurfaceReflectanceWithVIHandler**(**args*, *filter_veg: bool = True*, ***kwargs*)

Bases: *VIIRSJRRFileHandler*

File handler for surface reflectance files with optional vegetation indexes.

Initialize file handler and keep track of vegetation index filtering.

_get_veg_index_good_mask() → DataArray

_mask_invalid(*data_arr: DataArray*, *ds_info: dict*) → DataArray

satpy.readers.viirs_edr_active_fires module

VIIRS Active Fires reader.

This module implements readers for VIIRS Active Fires NetCDF and ASCII files.

class satpy.readers.viirs_edr_active_fires.**VIIRSActiveFiresFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*)

Bases: *NetCDF4FileHandler*

NetCDF4 reader for VIIRS Active Fires.

Open and perform initial investigation of NetCDF file.

property end_time

Get last date/time when observations were recorded.

get_dataset(*dsid*, *dsinfo*)

Get requested data as DataArray.

Parameters

- **dsid** – Dataset ID
- **param2** – Dataset Information

Returns

Data

Return type
Dask DataArray

property platform_name

Name of platform/satellite for this file.

property sensor_name

Name of sensor for this file.

property start_time

Get first date/time when observations were recorded.

class satpy.readers.viirs_edr_active_fires.**VIIRSActiveFiresTextFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

ASCII reader for VIIRS Active Fires.

Make sure filepath is valid and then reads data into a Dask DataFrame.

Parameters

- **filename** – Filename
- **filename_info** – Filename information
- **filetype_info** – Filetype information

property end_time

Get last date/time when observations were recorded.

get_dataset(*dsid*, *dsinfo*)

Get requested data as DataArray.

property start_time

Get first date/time when observations were recorded.

satpy.readers.viirs_edr_flood module

Interface to VIIRS flood product.

class satpy.readers.viirs_edr_flood.**VIIRSEDRFlood**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF4FileHandler*

VIIRS EDR Flood-product handler for HDF4 files.

Open file and collect information.

property end_time

Get end time.

get_area_def(*ds_id*)

Get area definition.

get_dataset(*ds_id*, *ds_info*)

Get dataset.

get_metadata(*data*, *ds_info*)

Get metadata.

property platform_name

Get platform name.

property sensor_name

Get sensor name.

property start_time

Get start time.

satpy.readers.viirs_l1b module

Interface to VIIRS L1B format.

class satpy.readers.viirs_l1b.**VIIRSL1BFileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*)

Bases: *NetCDF4FileHandler*

VIIRS L1B File Reader.

Initialize object.

static _dataset_name_to_var_path(*dataset_name: str*, *ds_info: dict*) → str

_get_dataset_file_units(*dataset_id*, *ds_info*, *var_path*)

_get_dataset_valid_range(*dataset_id*, *ds_info*, *var_path*)

_is_scan_based_array(*shape*)

_parse_datetime(*datestr*)

Parse datetime.

adjust_scaling_factors(*factors*, *file_units*, *output_units*)

Adjust scaling factors.

available_datasets(*configured_datasets=None*)

Generate dataset info and their availablity.

See *satpy.readers.file_handlers.BaseFileHandler.available_datasets()* for details.

property end_orbit_number

Get end orbit number.

property end_time

Get end time.

get_dataset(*dataset_id*, *ds_info*)

Get dataset.

get_metadata(*dataset_id*, *ds_info*)

Get metadata.

get_shape(*ds_id*, *ds_info*)

Get shape.

property platform_name

Get platform name.

property sensor_name

Get sensor name.

property start_orbit_number

Get start orbit number.

property start_time

Get start time.

satpy.readers.viirs_sdr module

Interface to VIIRS SDR format.

This reader implements the support of VIIRS SDR files as produced by CSPP and CLASS. It is comprised of two parts:

- A subclass of the YAMLFileReader class to allow handling all the files
- A filehandler class to implement the actual reading

Format documentation:

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

class satpy.readers.viirs_sdr.**VIIRSSDRFileHandler**(*filename*, *filename_info*, *filetype_info*, *use_tc=None*, ***kwargs*)

Bases: *JPSS_SDR_FileHandler*

VIIRS SDR HDF5 File Reader.

Initialize file handler.

get_bounding_box()

Get the bounding box of this file.

get_dataset(*dataset_id*, *ds_info*)

Get the dataset corresponding to *dataset_id*.

The size of the return DataArray will be dependent on the number of scans actually sensed, and not necessarily the regular 768 scanlines that the file contains for each granule. To that end, the number of scans for each granule is read from: `Data_Products/...Gran_x/N_Number_Of_Scans`.

class satpy.readers.viirs_sdr.**VIIRSSDRReader**(*config_files*, *use_tc=None*, ***kwargs*)

Bases: *FileYAMLReader*

Custom file reader for finding VIIRS SDR geolocation at runtime.

Initialize file reader and adjust geolocation preferences.

Parameters

- **config_files** (*iterable*) – yaml config files passed to base class
- **use_tc** (*boolean*) – If *True* use the terrain corrected files. If *False*, switch to non-TC files. If *None* (default), use TC if available, non-TC otherwise.

_abc_impl = <_abc._abc_data object>

_create_new_geo_file_handlers(*geo_filenames*)

_geo_dataset_groups(*c_info*)

_get_coordinates_for_dataset_key(*dsid*)

Get the coordinate dataset keys for *dsid*.

Wraps the base class method in order to load geolocation files from the geo reference attribute in the datasets file.

_get_file_handlers(*dsid*)

Get the file handler to load this dataset.

_get_primary_secondary_geo_groups(*ds_info*)

Find out which geolocation files are needed.

_is_viirs_dataset(*datasets*)

_load_filenames_from_geo_ref(*dsid*)

Load filenames from the N_GEO_Ref attribute of a dataset's file.

_remove_datasets_from_files(*filename_items*, *files_to_edit*, *considered_datasets*)

_remove_geo_datasets_from_files(*filename_items*, *files_to_edit*)

_remove_non_viirs_datasets_from_files(*filename_items*, *files_to_edit*)

_remove_not_loaded_geo_dataset_group(*c_dataset_groups*, *prime_geo*, *second_geo*)

filter_filenames_by_info(*filename_items*)

Filter out file using metadata from the filenames.

This sorts out the different lon and lat datasets depending on TC is desired or not.

get_right_geo_fhs(*dsid*, *fhs*)

Find the right geographical file handlers for given dataset ID *dsid*.

satpy.readers.viirs_sdr.**_get_invalid_info**(*granule_data*)

Get a detailed report of the missing data.

N/A: not applicable MISS: required value missing at time of processing OBPT: onboard pixel trim (overlapping/bow-tie pixel removed during SDR processing) OGPT: on-ground pixel trim (overlapping/bow-tie pixel removed during EDR processing) ERR: error occurred during processing / non-convergence ELINT: ellipsoid intersect failed / instrument line-of-sight does not intersect the Earth's surface VDNE: value does not exist / processing algorithm did not execute SOUB: scaled out-of-bounds / solution not within allowed range

satpy.readers.viirs_sdr.**split_desired_other**(*fhs*, *prime_geo*, *second_geo*)

Split the provided filehandlers *fhs* into desired filehandlers and others.

satpy.readers.viirs_vgac_l1c_nc module

Reading VIIRS VGAC data.

class satpy.readers.viirs_vgac_l1c_nc.**VGACFileHandler**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Reader VGAC data.

Init the file handler.

calibrate(*data*, *yaml_info*, *file_key*, *nc*)

Calibrate data.

convert_to_bt(*data*, *data_lut*, *scale_factor*)

Convert radances to brightness temperatures.

property end_time

Get the end time.

fix_radiances_not_in_percent(*data*)

Scale radiances to percent. This was not done in first version of data.

get_dataset(*key*, *yaml_info*)

Get dataset.

set_time_attrs(*data*)

Set time from attributes.

property start_time

Get the start time.

satpy.readers.virr_l1b module

Interface to VIRR (Visible and Infra-Red Radiometer) level 1b format.

The file format is HDF5. Important attributes:

- Latitude
- Longitude
- SolarZenith
- EV_Emissive
- EV_RefSB
- Emissive_Radiance_Offsets
- Emissive_Radiance_Scales
- RefSB_Cal_Coefficients
- RefSB_Effective_Wavelength
- Emmisive_Centroid_Wave_Number

Supported satellites:

- FY-3B and FY-3C.

For more information:

- https://www.wmo-sat.info/oscar/instruments/view/607.

class satpy.readers.virr_l1b.**VIRR_L1B**(*filename*, *filename_info*, *filetype_info*)

Bases: *HDF5FileHandler*

VIRR Level 1b reader.

Open file and perform initial setup.

_calibrate_emissive(*data*, *band_index*)

_calibrate_reflective(*data*, *band_index*)

_correct_slope(*slope*)

property end_time

Get ending observation time.

get_dataset(*dataset_id*, *ds_info*)

Create DataArray from file content for *dataset_id*.

property start_time

Get starting observation time.

satpy.readers.xmlformat module

Reads a format from an xml file to create dtypes and scaling factor arrays.

class satpy.readers.xmlformat.**XMLFormat**(*filename*)

Bases: object

XMLFormat object.

Init the format reader.

apply_scales(*array*)

Apply scales to *array*.

dtype(*key*)

Get the dtype for the format object.

satpy.readers.xmlformat.**_apply_scales**(*array*, *scales*, *dtype*)

Apply scales to the array.

satpy.readers.xmlformat.**parse_format**(*xml_file*)

Parse the xml file to create types, scaling factor types, and scales.

satpy.readers.xmlformat.**process_array**(*elt*, *text=False*)

Process an 'array' tag.

satpy.readers.xmlformat.**process_delimiter**(*elt*, *text=False*)

Process a 'delimiter' tag.

satpy.readers.xmlformat.**process_field**(*elt*, *text=False*)

Process a 'field' tag.

satpy.readers.xmlformat.**to_dtype**(*val*)

Parse *val* to return a dtype.

satpy.readers.xmlformat.**to_scaled_dtype**(*val*)

Parse *val* to return a dtype.

satpy.readers.xmlformat.**to_scales**(*val*)

Parse *val* to return an array of scale factors.

satpy.readers.yaml_reader module

Base classes and utilities for all readers configured by YAML files.

class satpy.readers.yaml_reader.**AbstractYAMLReader**(*config_dict*)

Bases: object

Base class for all readers that use YAML configuration files.

This class should only be used in rare cases. Its child class *FileYAMLReader* should be used in most cases.

Load information from YAML configuration file about how to read data files.

_abc_impl = <_abc._abc_data object>

_build_id_permutations(*dataset*, *id_keys*)

Build each permutation/product of the dataset.

property all_dataset_ids

Get DataIDs of all datasets known to this reader.

property all_dataset_names

Get names of all datasets known to this reader.

property available_dataset_ids

Get DataIDs that are loadable by this reader.

property available_dataset_names

Get names of datasets that are loadable by this reader.

abstract property end_time

End time of the reader.

abstract filter_selected_filenames(*filenames*)

Filter provided filenames by parameters in reader configuration.

Returns: iterable of usable files

classmethod from_config_files(**config_files*, ***reader_kwargs*)

Create a reader instance from one or more YAML configuration files.

get_dataset_key(*key*, ***kwargs*)

Get the fully qualified *DataID* matching *key*.

See *satpy.readers.get_key* for more information about kwargs.

abstract load(*dataset_keys*)

Load *dataset_keys*.

load_ds_ids_from_config()

Get the dataset ids from the config.

select_files_from_directory(*directory=None*, *fs=None*)

Find files for this reader in *directory*.

If directory is None or ‘’, look in the current directory.

Searches the local file system by default. Can search on a remote filesystem by passing an instance of a suitable implementation of `fsspec.spec.AbstractFileSystem`.

Parameters

- **directory** (*Optional[str]*) – Path to search.
- **fs** (*Optional[FileSystem]*) – fsspec FileSystem implementation to use. Defaults to None, using local file system.

Returns
list of strings describing matching files

select_files_from_pathnames(*filenames*)

Select the files from *filenames* this reader can handle.

property sensor_names

Names of sensors whose data is being loaded by this reader.

abstract property start_time

Start time of the reader.

supports_sensor(*sensor*)

Check if *sensor* is supported.

Returns True is *sensor* is None.

class satpy.readers.yaml_reader.**FileYAMLReader**(*config_dict*, *filter_parameters=None*, *filter_filenames=True*, ***kwargs*)

Bases: *AbstractYAMLReader*, *DataDownloadMixin*

Primary reader base class that is configured by a YAML file.

This class uses the idea of per-file "file handler" objects to read file contents and determine what is available in the file. This differs from the base *AbstractYAMLReader* which does not depend on individual file handler objects. In almost all cases this class should be used over its base class and can be used as a reader by itself and requires no subclassing.

Set up initial internal storage for loading file data.

_abc_impl = <_abc._abc_data object>

static _assign_coords_from_dataarray(*coords*, *ds*)

Assign coords from the *ds* dataarray if needed.

_coords_cache: WeakValueDictionary = <WeakValueDictionary>

_file_handlers_available_datasets()

Generate a series of available dataset information.

This is done by chaining file handler's *satpy.readers.file_handlers.BaseFileHandler.available_datasets()* together. See that method's documentation for more information.

Returns
Generator of (bool, dict) where the boolean tells whether the current dataset is available from any of the file handlers. The boolean can also be None in the case where no loaded file handler is configured to load the dataset. The dictionary is the metadata provided either by the YAML configuration files or by the file handler itself if it is a new dataset. The file handler may have also supplemented or modified the information.

_gather_ancillary_variables_ids(*datasets*)

Gather ancillary variables' ids.

This adds/modifies the dataset's *ancillary_variables* attr.

_get_coordinates_for_dataset_key(*dsid*)

Get the coordinate dataset keys for *dsid*.

_get_coordinates_for_dataset_keys(*dsids*)

Get all coordinates.

_get_file_handlers(*dsid*)

Get the file handler to load this dataset.

_get_lons_lats_from_coords(*coords*)

Get lons and lats from the coords list.

_load_ancillary_variables(*datasets*, ***kwargs*)

Load the ancillary variables of *datasets*.

_load_area_def(*dsid*, *file_handlers*, ***kwargs*)

Load the area definition of *dsid*.

static _load_dataset(*dsid*, *ds_info*, *file_handlers*, *dim='y'*, ***kwargs*)

Load only a piece of the dataset.

_load_dataset_area(*dsid*, *file_handlers*, *coords*, ***kwargs*)

Get the area for *dsid*.

_load_dataset_data(*file_handlers*, *dsid*, ***kwargs*)

_load_dataset_with_area(*dsid*, *coords*, ***kwargs*)

Load *dsid* and its area if available.

_make_area_from_coords(*coords*)

Create an appropriate area with the given *coords*.

_make_swath_definition_from_lons_lats(*lons*, *lats*)

Make a swath definition instance from lons and lats.

_new_filehandler_instances(*filetype_info*, *filename_items*, *fh_kwargs=None*)

Generate new filehandler instances.

_new_filehandlers_for_filetype(*filetype_info*, *filenames*, *fh_kwargs=None*)

Create filehandlers for a given filetype.

_preferred_filetype(*filetypes*)

Get the preferred filetype out of the *filetypes* list.

At the moment, it just returns the first filetype that has been loaded.

property available_dataset_ids

Get DataIDs that are loadable by this reader.

static check_file_covers_area(*file_handler*, *check_area*)

Check if the file covers the current area.

If the file doesn't provide any bounding box information or 'area' was not provided in *filter_parameters*, the check returns True.

create_filehandlers(*filenames*, *fh_kwargs=None*)

Organize the filenames into file types and create file handlers.

property end_time

End time of the latest file used by this reader.

static filename_items_for_filetype(*filenames*, *filetype_info*)

Iterate over the filenames matching *filetype_info*.

filter_fh_by_metadata(*filehandlers*)

Filter out filehandlers using provide filter parameters.

filter_filenames_by_info(*filename_items*)

Filter out file using metadata from the filenames.

Currently only uses start and end time. If only start time is available from the filename, keep all the filename that have a start time before the requested end time.

filter_selected_filenames(*filenames*)

Filter provided files based on metadata in the filename.

find_required_filehandlers(*requirements*, *filename_info*)

Find the necessary file handlers for the given requirements.

We assume here requirements are available.

Raises

- **KeyError, if no handler for the given requirements is available.** –
- **RuntimeError, if there is a handler for the given requirements,** –
- **but it doesn't match the filename info.** –

get_dataset_key(*key*, *available_only=False*, ***kwargs*)

Get the fully qualified *DataID* matching *key*.

This will first search through available DataIDs, datasets that should be possible to load, and fallback to "known" datasets, those that are configured but aren't loadable from the provided files. Providing `available_only=True` will stop this fallback behavior and raise a `KeyError` exception if no available dataset is found.

Parameters

- **key** (*str, float, DataID, DataQuery*) – Key to search for in this reader.
- **available_only** (*bool*) – Search only loadable datasets for the provided key. Loadable datasets are always searched first, but if `available_only=False` (default) then all known datasets will be searched.
- **kwargs** – See `satpy.readers.get_key()` for more information about kwargs.

Returns

Best matching DataID to the provided `key`.

Raises

KeyError – if no key match is found.

load(*dataset_keys*, *previous_datasets=None*, ***kwargs*)

Load *dataset_keys*.

If *previous_datasets* is provided, do not reload those.

metadata_matches(*sample_dict*, *file_handler=None*)

Check that file metadata matches filter_parameters of this reader.

property sensor_names

Names of sensors whose data is being loaded by this reader.

sorted_filetype_items()

Sort the instance's filetypes in using order.

property start_time

Start time of the earlier file used by this reader.

time_matches(*fstart*, *fend*)

Check that a file's start and end time mtach filter_parameters of this reader.

update_ds_ids_from_file_handlers()

Add or modify available dataset information.

Each file handler is consulted on whether or not it can load the dataset with the provided information dictionary. See `satpy.readers.file_handlers.BaseFileHandler.available_datasets()` for more information.

class satpy.readers.yaml_reader.**GEOFlippableFileYAMLReader**(*config_dict*, *filter_parameters=None*, *filter_filenames=True*, ***kwargs*)

Bases: `FileYAMLReader`

Reader for flippable geostationary data.

Set up initial internal storage for loading file data.

_abc_impl = <_abc._abc_data object>

_load_dataset_with_area(*dsid*, *coords*, *upper_right_corner='native'*, ***kwargs*)

Load *dsid* and its area if available.

class satpy.readers.yaml_reader.**GEOSegmentYAMLReader**(*config_dict*, *filter_parameters=None*, *filter_filenames=True*, ***kwargs*)

Bases: `GEOFlippableFileYAMLReader`

Reader for segmented geostationary data.

This reader pads the data to full geostationary disk if necessary.

This reader uses an optional `pad_data` keyword argument that can be passed to `Scene.load()` to control if padding is done (True by default). Passing *pad_data=False* will return data unpadded.

When using this class in a reader's YAML configuration, segmented file types (files that may have multiple segments) should specify an extra `expected_segments` piece of file_type metadata. This tells this reader how many total segments it should expect when padding data. Alternatively, the file patterns for a file type can include a `total_segments` field which will be used if `expected_segments` is not defined. This will default to 1 segment.

Set up initial internal storage for loading file data.

_abc_impl = <_abc._abc_data object>

_get_empty_segment(***kwargs*)

_get_new_areadef_for_padded_segment(*area*, *filetype*, *seg_size*, *segment*, *padding_type*)

_get_new_areadef_heights(*previous_area*, *previous_seg_size*, ***kwargs*)

_get_segments_areadef_with_later_padded(*file_handlers*, *filetype*, *dsid*, *available_segments*, *expected_segments*)

_get_y_area_extents_for_padded_segment(*area*, *filetype*, *padding_type*, *seg_size*, *segment*)

_load_area_def(*dsid*, *file_handlers*, *pad_data=True*)

Load the area definition of *dsid* with padding.

_load_area_def_with_padding(*dsid*, *file_handlers*)

Load the area definition of *dsid* with padding.

_load_dataset(*dsid*, *ds_info*, *file_handlers*, *dim='y'*, *pad_data=True*)

Load only a piece of the dataset.

_pad_earlier_segments_area(*file_handlers*, *dsid*, *area_defs*)

Pad area definitions for missing segments that are earlier in sequence than the first available.

_pad_later_segments_area(*file_handlers*, *dsid*)

Pad area definitions for missing segments that are later in sequence than the first available.

create_filehandlers(*filenames*, *fh_kwargs=None*)

Create file handler objects and determine expected segments for each.

class satpy.readers.yaml_reader.**GEOVariableSegmentYAMLReader**(*config_dict*, *filter_parameters=None*, *filter_filenames=True*, ***kwargs*)

Bases: *GEOSegmentYAMLReader*

GEOVariableSegmentYAMLReader for handling segmented GEO products with segments of variable height.

This YAMLReader overrides parts of the GEOSegmentYAMLReader to account for formats where the segments can have variable heights. It computes the sizes of the padded segments using the information available in the file(handlers), so that gaps of any size can be filled as needed.

This implementation was motivated by the FCI L1c format, where the segments (called chunks in the FCI world) can have variable heights. It is however generic, so that any future reader can use it. The requirement for the reader is to have a method called *get_segment_position_info*, returning a dictionary containing the positioning info for each segment (see example in *satpy.readers.fci_l1c_nc.FCIL1cNCFileHandler.get_segment_position_info()*).

For more information on please see the documentation of *satpy.readers.yaml_reader.GEOSegmentYAMLReader()*.

Initialise the GEOVariableSegmentYAMLReader object.

_abc_impl = <_abc._abc_data object>

_collect_segment_position_infos(*filetype*)

_extract_segment_location_dicts(*filetype*)

_get_empty_segment(*dim=None*, *idx=None*, *filetype=None*)

_get_new_areadef_heights(*previous_area*, *previous_seg_size*, *segment_n=None*, *filetype=None*)

_initialise_segment_infos(*filetype*)

_segment_heights(*filetype*, *grid_width*)

Compute optimal padded segment heights (in number of pixels) based on the location of available segments.

satpy.readers.yaml_reader.**_compute_optimal_missing_segment_heights**(*seg_infos*, *grid_type*, *expected_vertical_size*)

satpy.readers.yaml_reader.**_compute_positioning_data_for_missing_group**(*segment_start_rows*, *segment_end_rows*, *segment_heights*, *group*)

satpy.readers.yaml_reader.**_compute_proposed_sizes_of_missing_segments_in_group**(*group*, *segment_end_rows*, *segment_start_rows*)

satpy.readers.yaml_reader.**_find_missing_segments**(*file_handlers*, *ds_info*, *dsid*)

Find missing segments.

satpy.readers.yaml_reader.**_flip_dataset_data_and_area_extents**(*dataset*, *area_extents_to_update*, *flip_direction*)

Flip the data and area extents array for a dataset.

satpy.readers.yaml_reader.**_get_current_scene_orientation**(*area_extents_to_update*)

Get the current scene orientation from the area_extents.

satpy.readers.yaml_reader.**_get_dataset_area_extents_array**(*dataset_area_attr*)

Get dataset area extents in a numpy array for further flipping.

satpy.readers.yaml_reader.**_get_empty_segment_with_height**(*empty_segment*, *new_height*, *dim*)

Get a new empty segment with the specified height.

satpy.readers.yaml_reader.**_get_filebase**(*path*, *pattern*)

Get the end of *path* of same length as *pattern*.

satpy.readers.yaml_reader.**_get_grid_width_to_grid_type**(*seg_info*)

satpy.readers.yaml_reader.**_get_new_flipped_area_definition**(*dataset_area_attr*, *area_extents_to_update*, *flip_areadef_stacking*)

Get a new area definition with updated area_extents for flipped geostationary datasets.

satpy.readers.yaml_reader.**_get_projection_type**(*dataset_area_attr*)

Get the projection type from the crs coordinate operation method name.

satpy.readers.yaml_reader.**_get_target_scene_orientation**(*upper_right_corner*)

Get the target scene orientation from the target upper_right_corner.

'NE' corresponds to target_eastright and target_northup being True.

satpy.readers.yaml_reader.**_init_positioning_arrays_for_variable_padding**(*chk_infos*, *grid_type*, *exp_segment_nr*)

satpy.readers.yaml_reader.**_load_area_def**(*dsid*, *file_handlers*)

Load the area definition of *dsid*.

satpy.readers.yaml_reader.**_match_filenames**(*filenames*, *pattern*)

Get the filenames matching *pattern*.

satpy.readers.yaml_reader.**_populate_group_end_row_using_later_segment**(*group*, *segment_end_rows*, *segment_start_rows*)

satpy.readers.yaml_reader.**_populate_group_start_end_row_using_neighbour_segments**(*group*, *segment_end_rows*, *segment_start_rows*)

satpy.readers.yaml_reader.**_populate_group_start_row_using_previous_segment**(*group*, *segment_end_rows*, *segment_start_rows*)

satpy.readers.yaml_reader.**_populate_positioning_arrays_with_available_segment_info**(*chk_infos*, *grid_type*, *segment_start_rows*, *segment_end_rows*, *segment_heights*)

satpy.readers.yaml_reader.**_populate_start_end_rows_of_missing_segments_with_proposed_sizes**(*group*, *proposed_sizes_mis*, *segment_start_rows*, *segment_end_rows*, *segment_heights*)

satpy.readers.yaml_reader.**_set_orientation**(*dataset*, *upper_right_corner*)

Set the orientation of geostationary datasets.

Allows to flip geostationary imagery when loading the datasets. Example call: scn.load(['VIS008'], upper_right_corner='NE')

Parameters

- **dataset** – Dataset to be flipped.
- **upper_right_corner** (*str*) – Direction of the upper right corner of the image after flipping. Possible options are 'NW', 'NE', 'SW', 'SE', or 'native'. The common upright image orientation corresponds to 'NE'. Defaults to 'native' (no flipping is applied).

satpy.readers.yaml_reader.**_stack_area_defs**(*area_def_dict*)

Stack given dict of area definitions and return a StackedAreaDefinition.

satpy.readers.yaml_reader.**_verify_reader_info_assign_config_files**(*config*, *config_files*)

satpy.readers.yaml_reader.**listify_string**(*something*)

Take *something* and make it a list.

something is either a list of strings or a string, in which case the function returns a list containing the string. If *something* is None, an empty list is returned.

satpy.readers.yaml_reader.**load_yaml_configs**(**config_files*, *loader=<class 'yaml.cyaml.CLoader'>*)

Merge a series of YAML reader configuration files.

Parameters

- ***config_files** (*str*) – One or more pathnames to YAML-based reader configuration files that will be merged to create a single configuration.
- **loader** – Yaml loader object to load the YAML with. Defaults to *CLoader* if libyaml is available, *Loader* otherwise.

Returns: dict

Dictionary representing the entire YAML configuration with the addition of *config['reader']['config_files']* (the list of YAML pathnames that were merged).

satpy.readers.yaml_reader.**split_integer_in_most_equal_parts**(*x*, *n*)

Split an integer number x in n parts that are as equally-sizes as possible.

Module contents

Shared objects of the various reader classes.

class satpy.readers.**FSFile**(*file*, *fs=None*)

Bases: PathLike

Implementation of a PathLike file object, that can be opened.

Giving the filenames to Scene with valid transfer protocols will automatically use this class so manual usage of this class is needed mainly for fine-grained control.

This class is made to be used in conjuction with fsspec or s3fs. For example:

```
from satpy import Scene

import fsspec
filename = 'noaa-goes16/ABI-L1b-RadC/2019/001/17/*_G16_s20190011702186*'

the_files = fsspec.open_files("simplecache::s3://" + filename, s3={'anon': True})

from satpy.readers import FSFile
fs_files = [FSFile(open_file) for open_file in the_files]

scn = Scene(filenames=fs_files, reader='abi_l1b')
scn.load(['true_color_raw'])
```

Initialise the FSFile instance.

Parameters

- **file** (*str, Pathlike, or OpenFile*) – String, object implementing the *os.PathLike* protocol, or an *fsspec.OpenFile* instance. If passed an instance of *fsspec.OpenFile*, the following argument fs has no effect.
- **fs** (*fsspec filesystem, optional*) – Object implementing the fsspec filesystem protocol.

_abc_impl = <_abc._abc_data object>

_update_with_fs_open_kwargs(*user_kwargs*)

Complement keyword arguments for opening a file via file system.

open(**args*, ***kwargs*)

Open the file.

This is read-only.

satpy.readers.**_assign_files_to_readers**(*files_to_sort*, *reader_names*, *reader_kwargs*)

Assign files to readers.

Given a list of file names (paths), match those to reader instances.

Internal helper for group_files.

Parameters

- **files_to_sort** (*Collection[str]*) – Files to assign to readers.
- **reader_names** (*Collection[str]*) – Readers to consider
- **reader_kwargs** (*Mapping*) –

Returns

Mapping[str, Tuple[reader, Set[str]]] Mapping where the keys are reader names and the values are tuples of (reader_configs, filenames).

satpy.readers.**_check_reader_instances**(*reader_instances*)

satpy.readers.**_check_remaining_files**(*remaining_filenames*)

satpy.readers.**_early_exit**(*filenames*, *reader*)

satpy.readers.**_filter_groups**(*groups*, *missing='pass'*)

Filter multi-reader group-files behavior.

Helper for *group_files*. When *group_files* is called with multiple readers, make sure that the desired behaviour for missing files is enforced: if missing is `"raise"`, raise an exception if at least one group has at least one reader without files; if it is `"skip"`, remove those. If it is `"pass"`, do nothing. Yields groups to be kept.

Parameters

- **groups** (*List[Mapping[str, List[str]]]*) – groups as found by *group_files*.
- **missing** (*str*) – String controlling behaviour, see documentation above.

Yields

`Mapping[str:, List[str]]` – groups to be retained

satpy.readers.**_get_compression**(*file*)

satpy.readers.**_get_file_keys_for_reader_files**(*reader_files*, *group_keys=None*)

From a mapping from _assign_files_to_readers, get file keys.

Given a mapping where each key is a reader name and each value is a tuple of reader instance (typically FileYAMLReader) and a collection of files, return a mapping with the same keys, but where the values are lists of tuples of (keys, filename), where keys are extracted from the filenames according to group_keys and filenames are the names those keys were extracted from.

Internal helper for group_files.

Returns

Mapping[str, List[Tuple[Tuple, str]]], as described.

satpy.readers.**_get_fs_open_kwargs**(*file*)

Get keyword arguments for opening a file via file system.

For example compression.

satpy.readers.**_get_keys_with_empty_values**(*grp*)

Find mapping keys where values have length zero.

Helper for *_filter_groups*, which is in turn a helper for *group_files*. Given a mapping key -> Collection[Any], return the keys where the length of the collection is zero.

Parameters

grp (*Mapping[Any, Collection[Any]]*) – dictionary to check

Returns

set of keys

satpy.readers.**_get_loadables_for_reader_config**(*base_dir*, *reader*, *sensor*, *reader_configs*, *reader_kwargs*, *fs*)

Get loadables for reader configs.

Helper for find_files_and_readers.

Parameters

- **base_dir** – as for *find_files_and_readers*
- **reader** – as for *find_files_and_readers*
- **sensor** – as for *find_files_and_readers*
- **reader_configs** – reader metadata such as returned by *configs_for_reader*.
- **reader_kwargs** – Keyword arguments to be passed to reader.
- **fs** (*FileSystem*) – as for *find_files_and_readers*

satpy.readers.**_get_reader_and_filenames**(*reader*, *filenames*)

satpy.readers.**_get_reader_kwargs**(*reader*, *reader_kwargs*)

Help load_readers to form reader_kwargs.

Helper for load_readers to get reader_kwargs and reader_kwargs_without_filter in the desirable form.

satpy.readers.**_get_sorted_file_groups**(*all_file_keys*, *time_threshold*)

Get sorted file groups.

Get a list of dictionaries, where each list item consists of a dictionary mapping a tuple of keys to a mapping of reader names to files. The files listed in each list item are considered to be grouped within the same time.

Parameters

- **all_file_keys** –
- **_get_file_keys_for_reader_files** (*as returned by*) –
- **time_threshold** – temporal threshold

Returns

List[Mapping[Tuple, Mapping[str, List[str]]]], as described

Internal helper for group_files.

satpy.readers.**available_readers**(*as_dict=False*, *yaml_loader=<class 'yaml.loader.UnsafeLoader'>*)

Available readers based on current configuration.

Parameters

- **as_dict** (*bool*) – Optionally return reader information as a dictionary. Default: False.
- **yaml_loader** (*Optional[Union[yaml.BaseLoader, yaml.FullLoader, yaml.UnsafeLoader]]*) – The yaml loader type. Default: `yaml.UnsafeLoader`.

Returns

List of available reader names. If *as_dict* is *True* then a list of dictionaries including additionally reader information is returned.

Return type

Union[list[str], list[dict]]

satpy.readers.**configs_for_reader**(*reader=None*)

Generate reader configuration files for one or more readers.

Parameters

reader (*Optional[str]*) – Yield configs only for this reader

Returns: Generator of lists of configuration files

satpy.readers.**find_files_and_readers**(*start_time=None*, *end_time=None*, *base_dir=None*, *reader=None*, *sensor=None*, *filter_parameters=None*, *reader_kwargs=None*, *missing_ok=False*, *fs=None*)

Find files matching the provided parameters.

Use *start_time* and/or *end_time* to limit found filenames by the times in the filenames (not the internal file metadata). Files are matched if they fall anywhere within the range specified by these parameters.

Searching is **NOT** recursive.

Files may be either on-disk or on a remote file system. By default, files are searched for locally. Users can search on remote filesystems by passing an instance of an implementation of *fsspec.spec.AbstractFileSystem* (strictly speaking, any object of a class implementing a `glob` method works).

If locating files on a local file system, the returned dictionary can be passed directly to the *Scene* object through the *filenames* keyword argument. If it points to a remote file system, it is the responsibility of the user to download the files first (directly reading from cloud storage is not currently available in Satpy).

The behaviour of time-based filtering depends on whether or not the filename contains information about the end time of the data or not:

- if the end time is not present in the filename, the start time of the filename is used and has to fall between (inclusive) the requested start and end times
- otherwise, the timespan of the filename has to overlap the requested timespan

Example usage for querying a s3 filesystem using the s3fs module:

```
>>> import s3fs, satpy.readers, datetime
>>> satpy.readers.find_files_and_readers(
...     base_dir="s3://noaa-goes16/ABI-L1b-RadF/2019/321/14/",
...     fs=s3fs.S3FileSystem(anon=True),
...     reader="abi_l1b",
...     start_time=datetime.datetime(2019, 11, 17, 14, 40))
{'abi_l1b': [...]}
```

Parameters

- **start_time** (*datetime*) – Limit used files by starting time.
- **end_time** (*datetime*) – Limit used files by ending time.
- **base_dir** (*str*) – The directory to search for files containing the data to load. Defaults to the current directory.
- **reader** (*str or list*) – The name of the reader to use for loading the data or a list of names.
- **sensor** (*str or list*) – Limit used files by provided sensors.
- **filter_parameters** (*dict*) – Filename pattern metadata to filter on. *start_time* and *end_time* are automatically added to this dictionary. Shortcut for *reader_kwargs['filter_parameters']*.
- **reader_kwargs** (*dict*) – Keyword arguments to pass to specific reader instances to further configure file searching.
- **missing_ok** (*bool*) – If False (default), raise ValueError if no files are found. If True, return empty dictionary if no files are found.
- **fs** (fsspec.spec.AbstractFileSystem) – Optional, instance of implementation of fsspec.spec.AbstractFileSystem (strictly speaking, any object of a class implementing `.glob` is enough). Defaults to searching the local filesystem.

Returns
Dictionary mapping reader name string to list of filenames

Return type
dict

satpy.readers.**get_valid_reader_names**(*reader*)

Check for old reader names or readers pending deprecation.

satpy.readers.**group_files**(*files_to_sort*, *reader=None*, *time_threshold=10*, *group_keys=None*, *reader_kwargs=None*, *missing='pass'*)

Group series of files by file pattern information.

By default this will group files by their filename `start_time` assuming it exists in the pattern. By passing the individual dictionaries returned by this function to the Scene classes' `filenames`, a series *Scene* objects can be easily created.

Parameters

- **files_to_sort** (*iterable*) – File paths to sort in to group
- **reader** (*str or Collection[str]*) – Reader or readers whose file patterns should be used to sort files. If not given, try all readers (slow, adding a list of readers is strongly recommended).
- **time_threshold** (*int*) – Number of seconds used to consider time elements in a group as being equal. For example, if the 'start_time' item is used to group files then any time within *time_threshold* seconds of the first file's 'start_time' will be seen as occurring at the same time.
- **group_keys** (*list or tuple*) – File pattern information to use to group files. Keys are sorted in order and only the first key is used when comparing datetime elements with *time_threshold* (see above). This means it is recommended that datetime values should only come from the first key in `group_keys`. Otherwise, there is a good chance that files will not

be grouped properly (datetimes being barely unequal). Defaults to a reader's `group_keys` configuration (set in YAML), otherwise `('start_time',)`. When passing multiple readers, passing group_keys is strongly recommended as the behaviour without doing so is undefined.

- **reader_kwargs** (*dict*) – Additional keyword arguments to pass to reader creation.
- **missing** (*str*) – Parameter to control the behavior in the scenario where multiple readers were passed, but at least one group does not have files associated with every reader. Valid values are `"pass"` (the default), `"skip"`, and `"raise"`. If set to `"pass"`, groups are passed as-is. Some groups may have zero files for some readers. If set to `"skip"`, groups for which one or more readers have zero files are skipped (meaning that some files may not be associated to any group). If set to `"raise"`, raise a *FileNotFoundError* in case there are any groups for which one or more readers have no files associated.

Returns

List of dictionaries mapping 'reader' to a list of filenames. Each of these dictionaries can be passed as `filenames` to a *Scene* object.

`satpy.readers.`**`load_reader`**(*reader_configs*, ***reader_kwargs*)

Import and setup the reader from *reader_info*.

`satpy.readers.`**`load_readers`**(*filenames=None*, *reader=None*, *reader_kwargs=None*)

Create specified readers and assign files to them.

Parameters

- **filenames** (*iterable or dict*) – A sequence of files that will be used to load data from. A `dict` object should map reader names to a list of filenames for that reader.
- **reader** (*str or list*) – The name of the reader to use for loading the data or a list of names.
- **reader_kwargs** (*dict*) – Keyword arguments to pass to specific reader instances. This can either be a single dictionary that will be passed to all reader instances, or a mapping of reader names to dictionaries. If the keys of `reader_kwargs` match exactly the list of strings in `reader` or the keys of filenames, each reader instance will get its own keyword arguments accordingly.

Returns: Dictionary mapping reader name to reader instance

`satpy.readers.`**`open_file_or_filename`**(*unknown_file_thing*)

Try to open the *unknown_file_thing*, otherwise return the filename.

`satpy.readers.`**`read_reader_config`**(*config_files*, *loader=<class 'yaml.loader.UnsafeLoader'>*)

Read the reader *config_files* and return the extracted reader metadata.

satpy.tests package

Subpackages

satpy.tests.cf_tests package

Submodules

satpy.tests.cf_tests._test_data module

Functions and fixture to test CF code.

satpy.tests.cf_tests._test_data.**get_test_attrs**()

Create some dataset attributes for testing purpose.

Returns
Attributes, encoded attributes, encoded and flattened attributes

satpy.tests.cf_tests.test_area module

Tests for the CF Area.

class satpy.tests.cf_tests.test_area.**TestCFArea**

Bases: object

Test case for CF Area.

test_add_grid_mapping_cf_repr(*input_data_arr*)

Test the conversion from pyresample area object to CF grid mapping.

Projection has a corresponding CF representation (e.g. geos).

test_add_grid_mapping_cf_repr_no_ab(*input_data_arr*)

Test the conversion from pyresample area object to CF grid mapping.

Projection has a corresponding CF representation but no explicit a/b.

test_add_grid_mapping_no_cf_repr(*input_data_arr*)

Test the conversion from pyresample area object to CF grid mapping.

Projection does not have a corresponding CF representation (e.g. COSMO).

test_add_grid_mapping_oblique_mercator(*input_data_arr*)

Test the conversion from pyresample area object to CF grid mapping.

Projection is oblique mercator.

test_add_grid_mapping_transverse_mercator(*input_data_arr*)

Test the conversion from pyresample area object to CF grid mapping.

Projection is transverse mercator.

test_add_lonlat_coords(*dims*)

Test the conversion from areas to lon/lat.

test_area2cf_geos_area_nolonlats(*input_data_arr*, *include_lonlats*)

Test the conversion of an area to CF standards.

test_area2cf_swath(*input_data_arr*)

Test area2cf for swath definitions.

satpy.tests.cf_tests.test_area.**_gm_matches**(*gmapping*, *expected*)

Assert that all keys in `expected` match the values in `gmapping`.

satpy.tests.cf_tests.test_area.**input_data_arr**() → DataArray

Create a data array.

satpy.tests.cf_tests.test_attrs module

Tests for CF-compatible attributes encoding.

class satpy.tests.cf_tests.test_attrs.**TestCFAttributeEncoding**

Bases: object

Test case for CF attribute encodings.

test__encode_nc_attrs()

Test attributes encoding.

satpy.tests.cf_tests.test_coords module

CF processing of time information (coordinates and dimensions).

class satpy.tests.cf_tests.test_coords.**TestCFcoords**

Bases: object

Test cases for CF spatial dimension and coordinates.

datasets()

Create test dataset.

test__is_lon_or_lat_dataarray(*datasets*)

Test the _is_lon_or_lat_dataarray function.

test_add_coordinates_attrs_coords()

Check that coordinates link has been established correctly.

test_check_unique_projection_coords()

Test that the x and y coordinates are unique.

test_ensure_unique_nondimensional_coords()

Test that created coordinate variables are unique.

test_has_projection_coords(*datasets*)

Test the has_projection_coords function.

test_is_projected(*caplog*)

Tests for private _is_projected function.

class satpy.tests.cf_tests.test_coords.**TestCFtime**

Bases: object

Test cases for CF time dimension and coordinates.

test_add_time_bounds_dimension()

Test addition of CF-compliant time attributes.

satpy.tests.cf_tests.test_dataaarray module

Tests CF-compliant DataArray creation.

class satpy.tests.cf_tests.test_dataaarray.**TestCfDataArray**

Bases: object

Test creation of CF DataArray.

test_make_cf_dataarray()

Test the conversion of a DataArray to a CF-compatible DataArray.

test_make_cf_dataarray_one_dimensional_array()

Test the conversion of an 1d DataArray to a CF-compatible DataArray.

satpy.tests.cf_tests.test_dataaarray.**test_make_cf_dataarray_lonlat()**

Test correct CF encoding for area with lon/lat units.

satpy.tests.cf_tests.test_dataaarray.**test_preprocess_dataarray_name()**

Test saving an array to netcdf/cf where dataset name starting with a digit with prefix include orig name.

satpy.tests.cf_tests.test_datasets module

Tests CF-compliant Dataset(s) creation.

class satpy.tests.cf_tests.test_datasets.**TestCollectCfDataset**

Bases: object

Test case for collect_cf_dataset.

test_collect_cf_dataset()

Test collecting CF datasets from a DataArray objects.

test_collect_cf_dataset_with_latitude_named_lat()

Test collecting CF datasets with latitude named lat.

test_geographic_area_coords_attrs()

Test correct storage for area with lon/lat units.

class satpy.tests.cf_tests.test_datasets.**TestCollectCfDatasets**

Bases: object

Test case for collect_cf_datasets.

test_empty_collect_cf_datasets()

Test that if no DataArrays, collect_cf_datasets raise error.

satpy.tests.cf_tests.test_encoding module

Tests for compatible netCDF/Zarr DataArray encodings.

class satpy.tests.cf_tests.test_encoding.**TestUpdateEncoding**

Bases: object

Test update of dataset encodings.

fake_ds()

Create fake data for testing.

fake_ds_digit()

Create fake data for testing.

test_dataset_name_digit(*fake_ds_digit*)

Test data with dataset name staring with a digit.

test_with_time(*fake_ds*)

Test data with a time dimension.

test_without_time(*fake_ds*)

Test data with no time dimension.

Module contents

The CF dataset tests package.

satpy.tests.compositor_tests package

Submodules

satpy.tests.compositor_tests.test_abi module

Tests for ABI compositors.

class satpy.tests.compositor_tests.test_abi.**TestABIComposites**(*methodName='runTest'*)

Bases: TestCase

Test ABI-specific composites.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_load_composite_yaml()

Test loading the yaml for this sensor.

test_simulated_green()

Test creating a fake 'green' band.

satpy.tests.compositor_tests.test_agri module

Tests for AGRI compositors.

class satpy.tests.compositor_tests.test_agri.**TestAGRIComposites**(*methodName='runTest'*)

Bases: TestCase

Test AGRI-specific composites.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_load_composite_yaml()

Test loading the yaml for this sensor.

test_simulated_red()

Test creating a fake 'red' band.

satpy.tests.compositor_tests.test_ahi module

Tests for AHI compositors.

class satpy.tests.compositor_tests.test_ahi.**TestAHIComposites**(*methodName='runTest'*)

Bases: TestCase

Test AHI-specific composites.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_load_composite_yaml()

Test loading the yaml for this sensor.

satpy.tests.compositor_tests.test_glm module

Tests for GLM compositors.

class satpy.tests.compositor_tests.test_glm.**TestGLMComposites**

Bases: object

Test GLM-specific composites.

test_highlight_compositor()

Test creating a highlight composite.

test_load_composite_yaml()

Test loading the yaml for this sensor.

satpy.tests.compositor_tests.test_sar module

Tests for SAR compositors.

class satpy.tests.compositor_tests.test_sar.**TestSARComposites**(*methodName='runTest'*)

Bases: TestCase

Test SAR-specific composites.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_sar_ice()

Test creating a the sar_ice composite.

test_sar_ice_log()

Test creating a the sar_ice_log composite.

satpy.tests.compositor_tests.test_spectral module

Tests for spectral correction compositors.

class satpy.tests.compositor_tests.test_spectral.**TestNdviHybridGreenCompositor**

Bases: object

Test NDVI-weighted hybrid green correction of green band.

setup_method()

Initialize channels.

test_invalid_strength()

Test using invalid *strength* term for non-linear scaling.

test_ndvi_hybrid_green()

Test General functionality with linear scaling from ndvi to blend fraction.

test_ndvi_hybrid_green_dtype()

Test that the datatype is not altered by the compositor.

test_nonlinear_scaling()

Test non-linear scaling using *strength* term.

class satpy.tests.compositor_tests.test_spectral.**TestSpectralComposites**

Bases: object

Test composites for spectral channel corrections.

setup_method()

Initialize channels.

test_bad_lengths()

Test that error is raised if the amount of channels to blend does not match the number of weights.

test_green_corrector()

Test the deprecated class for green corrections.

test_hybrid_green()

Test hybrid green correction of the 'green' band.

test_spectral_blender()

Test the base class for spectral blending of channels.

satpy.tests.compositor_tests.test_viirs module

Tests for VIIRS compositors.

class satpy.tests.compositor_tests.test_viirs.**TestVIIRSComposites**

Bases: object

Test various VIIRS-specific composites.

area()

Return fake area for use with DNB tests.

dnb(*area*)

Return fake channel 1 data for DNB tests.

lza(*area*)

Return fake lunal zenith angle dataset for DNB tests.

sza(*area*)

Return fake sza dataset for DNB tests.

test_adaptive_dnb(*dnb*, *sza*)

Test the 'adaptive_dnb' compositor.

test_erf_dnb(*dnb_units*, *saturation_correction*, *area*, *sza*, *lza*)

Test the 'dynamic_dnb' or ERF DNB compositor.

test_histogram_dnb(*dnb*, *sza*)

Test the 'histogram_dnb' compositor.

test_hncc_dnb(*area*, *dnb*, *sza*, *lza*)

Test the 'hncc_dnb' compositor.

test_hncc_dnb_nomoonpha(*area*, *dnb*, *sza*, *lza*)

Test the 'hncc_dnb' compositor when no moon phase data is provided.

test_load_composite_yaml()

Test loading the yaml for this sensor.

test_snow_age(*area*)

Test the 'snow_age' compositor.

Module contents

Tests for compositors.

satpy.tests.enhancement_tests package

Submodules

satpy.tests.enhancement_tests.test_abi module

Unit testing for the ABI enhancement functions.

class satpy.tests.enhancement_tests.test_abi.**TestABIEnhancement**(*methodName='runTest'*)

Bases: TestCase

Test the ABI enhancement functions.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create fake data for the tests.

test_cimss_true_color_contrast()

Test the cimss_true_color_contrast enhancement.

satpy.tests.enhancement_tests.test_atmosphere module

Tests for enhancements in enhancements/atmosphere.py.

satpy.tests.enhancement_tests.test_atmosphere.**test_essl_moisture**()

Test ESSL moisture compositor.

satpy.tests.enhancement_tests.test_enhancements module

Unit testing the enhancements functions, e.g. cira_stretch.

class satpy.tests.enhancement_tests.test_enhancements.**TestColormapLoading**

Bases: object

Test utilities used with colormaps.

test_cmap_bad_mode(*real_mode*, *forced_mode*, *filename_suffix*)

Test that reading colormaps with the wrong mode fails.

test_cmap_from_config_path(*tmp_path*)

Test loading a colormap relative to a config path.

test_cmap_from_file(*color_scale*, *colormap_mode*, *extra_kwargs*, *filename_suffix*)

Test that colormaps can be loaded from a binary file.

test_cmap_from_file_bad_shape()

Test that unknown array shape causes an error.

test_cmap_from_trollimage()

Test that colormaps in trollimage can be loaded.

test_cmap_list()

Test that colors can be a list/tuple.

test_cmap_no_colormap()

Test that being unable to create a colormap raises an error.

test_cmap_vrgb_as_rgba()

Test that data created as VRGB still reads as RGBA.

class satpy.tests.enhancement_tests.test_enhancements.**TestEnhancementStretch**

Bases: object

Class for testing enhancements in satpy.enhancements.

setup_method()

Create test data used by every test.

tearDown()

Clean up.

test_apply_enhancement(*input_data_name*, *decorator*, *exp_call_cls*)

Test the 'apply_enhancement' utility function.

test_btemp_threshold()

Test applying the cira_stretch.

test_cira_stretch()

Test applying the cira_stretch.

test_colorize()

Test the colorize enhancement function.

test_lookup()

Test the lookup enhancement function.

test_merge_colormaps()

Test merging colormaps.

test_palettize()

Test the palettize enhancement function.

test_piecewise_linear_stretch()

Test the piecewise_linear_stretch enhancement function.

test_reinhard()

Test the reinhard algorithm.

test_three_d_effect()

Test the three_d_effect enhancement function.

class satpy.tests.enhancement_tests.test_enhancements.**TestTCREnhancement**

Bases: object

Test the AHI enhancement functions.

setup_method()

Create test data.

test_jma_true_color_reproduction()

Test the jma_true_color_reproduction enhancement.

satpy.tests.enhancement_tests.test_enhancements.**_generate_cmap_test_data**(*color_scale*, *colormap_mode*)

satpy.tests.enhancement_tests.test_enhancements.**_write_cmap_to_file**(*cmap_filename*, *cmap_data*)

satpy.tests.enhancement_tests.test_enhancements.**closed_named_temp_file**(***kwargs*)

Named temporary file context manager that closes the file after creation.

This helps with Windows systems which can get upset with opening or deleting a file that is already open.

satpy.tests.enhancement_tests.test_enhancements.**fake_area**()

Return a fake 2×2 area.

satpy.tests.enhancement_tests.test_enhancements.**identical_decorator**(*func*)

Decorate but do nothing.

satpy.tests.enhancement_tests.test_enhancements.**run_and_check_enhancement**(*func*, *data*, *expected*, ***kwargs*)

Perform basic checks that apply to multiple tests.

satpy.tests.enhancement_tests.test_enhancements.**test_nwcsaf_comps**(*fake_area*, *tmp_path*, *data*)

Test loading NWCSAF composites.

satpy.tests.enhancement_tests.test_enhancements.**test_on_dask_array**()

Test the *on_dask_array* decorator.

satpy.tests.enhancement_tests.test_enhancements.**test_on_separate_bands**()

Test the *on_separate_bands* decorator.

satpy.tests.enhancement_tests.test_enhancements.**test_using_map_blocks**()

Test the *using_map_blocks* decorator.

satpy.tests.enhancement_tests.test_viirs module

Unit testing for the VIIRS enhancement function.

class satpy.tests.enhancement_tests.test_viirs.**TestVIIRSEnhancement**(*methodName='runTest'*)

Bases: TestCase

Class for testing the VIIRS enhancement function in satpy.enhancements.viirs.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data.

test_viirs()

Test VIIRS flood enhancement.

Module contents

The enhancements tests package.

satpy.tests.modifier_tests package

Submodules

satpy.tests.modifier_tests.test_angles module

Tests for the angles in modifiers.

class satpy.tests.modifier_tests.test_angles.**TestAngleGeneration**

Bases: object

Test the angle generation utility functions.

static _check_cache_and_clear(*tmp_path*, *exp_num_zarr*)

static _check_cached_result(*results*, *exp_zarr_chunks*)

test_cache_get_angles(*input_func*, *num_normalized_chunks*, *exp_zarr_chunks*, *input2_func*, *exp_equal_sun*, *exp_num_zarr*, *force_bad_glob*, *tmp_path*)

Test get_angles when caching is enabled.

test_cached_no_chunks_fails(*tmp_path*)

Test that trying to pass non-dask arrays and no chunks fails.

test_cached_result_numpy_fails(*tmp_path*)

Test that trying to cache with non-dask arrays fails.

test_caching_with_array_in_args_does_not_warn_when_caching_is_not_enabled(*tmp_path*, *recwarn*)

Test that trying to cache with non-dask arrays fails.

test_caching_with_array_in_args_warns(*tmp_path*)

Test that trying to cache with non-dask arrays fails.

test_get_angles(*input_func*, *exp_calls*)

Test sun and satellite angle calculation.

test_get_angles_satpos_preference(*forced_preference*)

Test that 'actual' satellite position is used for generating sensor angles.

test_no_cache_dir_fails(*tmp_path*)

Test that 'cache_dir' not being set fails.

test_relative_azimuth_calculation()

Test relative azimuth calculation.

test_solazi_correction()

Test that solar azimuth angles are corrected into the right range.

satpy.tests.modifier_tests.test_angles.**_angle_cache_area_def**()

satpy.tests.modifier_tests.test_angles.**_angle_cache_stacked_area_def**()

satpy.tests.modifier_tests.test_angles.**_assert_allclose_if**(*expect_equal*, *arr1*, *arr2*)

satpy.tests.modifier_tests.test_angles.**_diff_sat_pos_datetime**(*orig_data*)

satpy.tests.modifier_tests.test_angles.**_get_angle_test_data**(*area_def: AreaDefinition | StackedAreaDefinition | None = None*, *chunks: int | tuple | None = 2*, *shape: tuple = (5, 5)*, *dims: tuple | None = None*) → DataArray

satpy.tests.modifier_tests.test_angles.**_get_angle_test_data_odd_chunks**()

satpy.tests.modifier_tests.test_angles.**_get_angle_test_data_odd_chunks2**()

satpy.tests.modifier_tests.test_angles.**_get_angle_test_data_rgb**()

satpy.tests.modifier_tests.test_angles.**_get_angle_test_data_rgb_nodims**()

satpy.tests.modifier_tests.test_angles.**_get_stacked_angle_test_data**()

satpy.tests.modifier_tests.test_angles.**_glob_reversed**(*pat*)

Behave like glob but force results to be in the wrong order.

satpy.tests.modifier_tests.test_angles.**_mock_glob_if**(*mock_glob*)

satpy.tests.modifier_tests.test_angles.**_similar_sat_pos_datetime**(*orig_data*, *lon_offset=0.04*)

satpy.tests.modifier_tests.test_crefl module

Tests for the CREFL ReflectanceCorrector modifier.

class satpy.tests.modifier_tests.test_crefl.**TestReflectanceCorrectorModifier**

Bases: object

Test the CREFL modifier.

static data_area_ref_corrector()

Create test area definition and data.

test_reflectance_corrector_abi(*name*, *wavelength*, *resolution*, *exp_mean*, *exp_unique*)

Test ReflectanceCorrector modifier with ABI data.

test_reflectance_corrector_bad_prereqs()

Test ReflectanceCorrector modifier with wrong number of inputs.

test_reflectance_corrector_different_chunks(*tmpdir*, *url*, *dem_mock_cm*, *dem_sds*)

Test that the modifier works with different chunk sizes for inputs.

The modifier uses dask's "map_blocks". If the input chunks aren't the same an error is raised.

test_reflectance_corrector_modis()

Test ReflectanceCorrector modifier with MODIS data.

test_reflectance_corrector_viirs(*tmpdir*, *url*, *dem_mock_cm*, *dem_sds*)

Test ReflectanceCorrector modifier with VIIRS data.

satpy.tests.modifier_tests.test_crefl.**_create_fake_dem_file**(*dem_fn*, *var_name*, *fill_value*)

satpy.tests.modifier_tests.test_crefl.**_make_viirs_xarray**(*data*, *area*, *name*, *standard_name*, *wavelength=None*, *units='degrees'*, *calibration=None*)

satpy.tests.modifier_tests.test_crefl.**_mock_and_create_dem_file**(*tmpdir*, *url*, *var_name*, *fill_value=None*)

satpy.tests.modifier_tests.test_crefl.**_mock_dem_retrieve**(*tmpdir*, *url*)

satpy.tests.modifier_tests.test_crefl.**mock_cmgdem**(*tmpdir*, *url*)

Create fake file representing CMGDEM.hdf.

satpy.tests.modifier_tests.test_crefl.**mock_tbase**(*tmpdir*, *url*)

Create fake file representing tbase.hdf.

satpy.tests.modifier_tests.test_filters module

Implementation of some image filters.

satpy.tests.modifier_tests.test_filters.**test_median**(*caplog*)

Test the median filter modifier.

satpy.tests.modifier_tests.test_parallax module

Tests related to parallax correction.

class satpy.tests.modifier_tests.test_parallax.**TestForwardParallax**

Bases: object

Test the forward parallax function with various inputs.

test_get_parallax_corrected_lonlats_clearsky()

Test parallax correction for clearsky case (returns NaN).

test_get_parallax_corrected_lonlats_cloudy_slant()

Test parallax correction for fully cloudy scene (not SSP).

test_get_parallax_corrected_lonlats_cloudy_ssp(*lat*, *lon*, *resolution*)

Test parallax correction for fully cloudy scene at SSP.

test_get_parallax_corrected_lonlats_horizon()

Test that exception is raised if satellites exactly at the horizon.

Test the rather unlikely case of a satellite elevation of exactly 0

test_get_parallax_corrected_lonlats_mixed()

Test parallax correction for mixed cloudy case.

test_get_parallax_corrected_lonlats_ssp()

Test that at SSP, parallax correction does nothing.

test_get_surface_parallax_displacement()

Test surface parallax displacement.

class satpy.tests.modifier_tests.test_parallax.**TestParallaxCorrectionClass**

Bases: object

Test that the ParallaxCorrection class is behaving sensibly.

test_correct_area_clearsky(*sat_pos*, *ar_pos*, *resolution*, *caplog*)

Test that ParallaxCorrection doesn't change clearsky geolocation.

test_correct_area_clearsky_different_resolutions(*res1*, *res2*)

Test clearsky correction when areas have different resolutions.

test_correct_area_cloudy_no_overlap()

Test cloudy correction when areas have no overlap.

test_correct_area_cloudy_partly_shifted()

Test cloudy correction when areas overlap only partly.

test_correct_area_cloudy_same_area()

Test cloudy correction when areas are the same.

test_correct_area_no_orbital_parameters(*caplog*, *fake_tle*)

Test ParallaxCorrection when CTH has no orbital parameters.

Some CTH products, such as NWCSAF-GEO, do not include information on satellite location directly. Rather, they include platform name, sensor, start time, and end time, that we have to use instead.

test_correct_area_partlycloudy(*daskify*)

Test ParallaxCorrection for partly cloudy situation.

test_correct_area_ssp(*lat*, *lon*, *resolution*)

Test that ParallaxCorrection doesn't touch SSP.

test_init_parallaxcorrection(*center*, *sizes*, *resolution*)

Test that ParallaxCorrection class can be instantiated.

class satpy.tests.modifier_tests.test_parallax.**TestParallaxCorrectionModifier**

Bases: object

Test that the parallax correction modifier works correctly.

_get_fake_cloud_datasets(*test_area*, *cth*, *use_dask*)

Return datasets for BT and CTH for fake cloud.

test_area(*request*)

Produce test area for parallax correction unit tests.

Produce test area for the modifier-interface parallax correction unit tests.

test_modifier_interface_cloud_moves_to_observer(*cth*, *use_dask*, *test_area*)

Test that a cloud moves to the observer.

With the modifier interface, use a high resolution area and test that pixels are moved in the direction of the observer and not away from it.

test_modifier_interface_fog_no_shift(*test_area*)

Test that fog isn't masked or shifted.

test_parallax_modifier_interface()

Test the modifier interface.

test_parallax_modifier_interface_with_cloud()

Test the modifier interface with a cloud.

Test corresponds to a real bug encountered when using CTH data from NWCSAF-GEO, which created strange speckles in Africa (see https://github.com/pytroll/satpy/pull/1904#issuecomment-1011161623 for an example). Create fake CTH corresponding to NWCSAF-GEO area and BT corresponding to full disk SEVIRI, and test that no strange speckles occur.

class satpy.tests.modifier_tests.test_parallax.**TestParallaxCorrectionSceneLoad**

Bases: object

Test that scene load interface works as expected.

conf_file(*yaml_code*, *tmp_path*)

Produce a fake configuration file.

fake_scene(*yaml_code*)

Produce fake scene and prepare fake composite config.

test_double_load(*fake_scene*, *conf_file*, *fake_tle*)

Test that loading corrected and uncorrected works correctly.

When the modifier `__call__` method fails to call `self.apply_modifier_info(new, old)` and the original and parallax-corrected dataset are requested at the same time, the DataArrays differ but the underlying dask arrays have object identity, which in turn leads to both being parallax corrected. This unit test confirms that there is no such object identity.

test_enhanced_image(*fake_scene*, *conf_file*, *fake_tle*)

Test that image enhancement is the same.

test_no_compute(*fake_scene*, *conf_file*)

Test that no computation occurs.

yaml_code()

Return YAML code for parallax_corrected_VIS006.

satpy.tests.modifier_tests.test_parallax.**_get_attrs**(*lat*, *lon*, *height=35000*)

Get attributes for datasets in fake scene.

satpy.tests.modifier_tests.test_parallax.**_get_fake_areas**(*center*, *sizes*, *resolution*, *code=4326*)

Get multiple square areas with the same center.

Returns multiple square areas centered at the same location

Parameters

- **center** (*Tuple[float, float]*) – Center of all areass
- **sizes** (*List[int]*) – Sizes of areas
- **resolution** (*float*) – Resolution of fake area.

Returns

List of areas.

satpy.tests.modifier_tests.test_parallax.**fake_tle**()

Produce fake Two Line Element (TLE) object from pyorbital.

Module contents

Tests for modifiers.

satpy.tests.multiscene_tests package

Submodules

satpy.tests.multiscene_tests.test_blend module

Unit tests for blending datasets with the Multiscene object.

class satpy.tests.multiscene_tests.test_blend.**TestBlendFuncs**

Bases: object

Test individual functions used for blending.

datasets_and_weights()

X-Array datasets with area definition plus weights for input to tests.

test_blend_function_stack(*datasets_and_weights*)

Test the 'stack' function.

test_blend_function_stack_weighted(*datasets_and_weights*, *line*, *column*)

Test the 'stack_weighted' function.

test_blend_two_scenes_bad_blend_type(*multi_scene_and_weights*, *groups*)

Test exception is raised when bad 'blend_type' is used.

test_blend_two_scenes_using_stack(*multi_scene_and_weights*, *groups*, *scene1_with_weights*, *scene2_with_weights*)

Test blending two scenes by stacking them on top of each other using function 'stack'.

test_blend_two_scenes_using_stack_weighted(*multi_scene_and_weights*, *groups*, *scene1_with_weights*, *scene2_with_weights*, *combine_times*, *blend_func*, *exp_result_func*)

Test stacking two scenes using weights.

Here we test that the start and end times can be combined so that they describe the start and times of the entire data series. We also test the various types of weighted stacking functions (ex. select, blend).

test_timeseries(*datasets_and_weights*)

Test the 'timeseries' function.

class satpy.tests.multiscene_tests.test_blend.**TestTemporalRGB**

Bases: object

Test the temporal RGB blending method.

static _assert_results(*res*, *expected_start_time*, *expected_result*)

expected_result()

Return the expected result arrays.

nominal_data()

Return the input arrays for the nominal use case.

test_extra_datasets(*nominal_data*, *expected_result*)

Test that only the first three arrays affect the usage.

test_nominal(*nominal_data*, *expected_result*)

Test that nominal usage with 3 datasets works.

satpy.tests.multiscene_tests.test_blend.**_check_stacked_metadata**(*data_arr: DataArray*, *exp_name: str*) → None

satpy.tests.multiscene_tests.test_blend.**_get_expected_stack_blend**(*scene1: Scene*, *scene2: Scene*) → DataArray

satpy.tests.multiscene_tests.test_blend.**_get_expected_stack_select**(*scene1: Scene*, *scene2: Scene*) → DataArray

satpy.tests.multiscene_tests.test_blend.**cloud_type_data_array1**(*test_area*, *data_type*, *image_mode*)

Get DataArray for cloud type in the first test Scene.

satpy.tests.multiscene_tests.test_blend.**cloud_type_data_array2**(*test_area*, *data_type*, *image_mode*)

Get DataArray for cloud type in the second test Scene.

satpy.tests.multiscene_tests.test_blend.**data_type**(*request*)

Get array data type of the DataArray being tested.

satpy.tests.multiscene_tests.test_blend.**groups**()

Get group definitions for the MultiScene.

satpy.tests.multiscene_tests.test_blend.**image_mode**(*request*)

Get image mode of the main DataArray being tested.

satpy.tests.multiscene_tests.test_blend.**multi_scene_and_weights**(*scene1_with_weights*, *scene2_with_weights*)

Create small multi-scene for testing.

satpy.tests.multiscene_tests.test_blend.**scene1_with_weights**(*cloud_type_data_array1*, *test_area*)

Create first test scene with a dataset of weights.

satpy.tests.multiscene_tests.test_blend.**scene2_with_weights**(*cloud_type_data_array2*, *test_area*)

Create second test scene.

satpy.tests.multiscene_tests.test_blend.**test_area**()

Get area definition used by test DataArrays.

satpy.tests.multiscene_tests.test_misc module

Unit tests for the Multiscene object.

class satpy.tests.multiscene_tests.test_misc.**TestMultiScene**(*methodName='runTest'*)

Bases: TestCase

Test basic functionality of MultiScene.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_from_files()

Test creating a multiscene from multiple files.

test_init_children()

Test creating a multiscene with children.

test_init_empty()

Test creating a multiscene with no children.

test_properties()

Test basic properties/attributes of the MultiScene.

class satpy.tests.multiscene_tests.test_misc.**TestMultiSceneGrouping**

Bases: object

Test dataset grouping in MultiScene.

groups()

Get group definitions for the MultiScene.

multi_scene(*scene1*, *scene2*)

Create small multi scene for testing.

scene1()

Create first test scene.

scene2()

Create second test scene.

test_fails_to_add_multiple_datasets_from_the_same_scene_to_a_group(*multi_scene*)

Test that multiple datasets from the same scene in one group fails.

test_multi_scene_grouping(*multi_scene*, *groups*, *scene1*)

Test grouping a MultiScene.

satpy.tests.multiscene_tests.test_save_animation module

Unit tests for saving animations using Multiscene.

class satpy.tests.multiscene_tests.test_save_animation.**TestMultiSceneSave**(*methodName='runTest'*)

Bases: TestCase

Test saving a MultiScene to various formats.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create temporary directory to save files to.

tearDown()

Remove the temporary directory created for a test.

test_crop()

Test the crop method.

test_save_datasets_distributed_delayed()

Test distributed save for writers returning delayed obejcts e.g. simple_image.

test_save_datasets_distributed_source_target()

Test distributed save for writers returning sources and targets e.g. geotiff writer.

test_save_datasets_simple()

Save a series of fake scenes to an PNG images.

test_save_mp4_distributed()

Save a series of fake scenes to an mp4 video.

test_save_mp4_no_distributed()

Save a series of fake scenes to an mp4 video when distributed isn't available.

satpy.tests.multiscene_tests.test_save_animation.**test_save_mp4**(*smg*, *tmp_path*)

Save a series of fake scenes to an mp4 video.

satpy.tests.multiscene_tests.test_utils module

Utilties to assist testing the Multiscene functionality.

Creating fake test data for use in the other Multiscene test modules.

satpy.tests.multiscene_tests.test_utils.**_create_test_area**(*proj_str=None*, *shape=(5, 10)*, *extents=None*)

Create a test area definition.

satpy.tests.multiscene_tests.test_utils.**_create_test_dataset**(*name*, *shape=(5, 10)*, *area=None*, *values=None*, *dims=('y', 'x')*)

Create a test DataArray object.

satpy.tests.multiscene_tests.test_utils.**_create_test_int8_dataset**(*name*, *shape=(5, 10)*, *area=None*, *values=None*, *dims=('y', 'x')*)

Create a test DataArray object.

satpy.tests.multiscene_tests.test_utils.**_create_test_scenes**(*num_scenes=2*, *shape=(5, 10)*, *area=None*)

Create some test scenes for various test cases.

satpy.tests.multiscene_tests.test_utils.**_fake_get_enhanced_image**(*img*, *enhance=None*, *overlay=None*, *decorate=None*)

Module contents

Unit tests for Multiscene.

satpy.tests.reader_tests package

Subpackages

satpy.tests.reader_tests.gms package

Submodules

satpy.tests.reader_tests.gms.test_gms5_vissr_data module

Real world test data for GMS-5 VISSR unit tests.

satpy.tests.reader_tests.gms.test_gms5_vissr_l1b module

Unit tests for GMS-5 VISSR reader.

class satpy.tests.reader_tests.gms.test_gms5_vissr_l1b.**TestCorruptFile**

Bases: object

Test reading corrupt files.

corrupt_file(*file_contents*, *tmp_path*)

Write corrupt VISSR file to disk.

file_contents()

Get corrupt file contents (all zero).

test_corrupt_file(*corrupt_file*)

Test reading a corrupt file.

class satpy.tests.reader_tests.gms.test_gms5_vissr_l1b.**TestEarthMask**

Bases: object

Test getting the earth mask.

test_get_earth_mask()

Test getting the earth mask.

class satpy.tests.reader_tests.gms.test_gms5_vissr_l1b.**TestFileHandler**

Bases: object

Test VISSR file handler.

_patch_number_of_pixels_per_scanline(*monkeypatch*)

Patch data types so that each scanline has two pixels.

area_def_exp(*dataset_id*)

Get expected area definition.

attitude_prediction()

Get attitude prediction.

attrs_exp(*area_def_exp*)

Get expected dataset attributes.

cal_params(*vis_calibration*, *ir1_calibration*, *ir2_calibration*, *wv_calibration*)

Get calibration parameters.

control_block(*dataset_id*)

Get VISSR control block.

coord_conv()

Get parameters for coordinate conversions.

Adjust pixel offset so that the first column is at the image center. This has the advantage that we can test with very small 2x2 images. Otherwise, all pixels would be in space.

coordinate_conversion(*coord_conv*, *simple_coord_conv_table*)

Get all coordinate conversion parameters.

dataset_exp(*dataset_id*, *ir1_counts_exp*, *ir1_bt_exp*, *vis_refl_exp*)

Get expected dataset.

dataset_id(*request*)

Get dataset ID.

file_contents(*control_block*, *image_parameters*, *image_data*)

Get VISSR file contents.

file_handler(*vissr_file_like*, *mask_space*)

Get file handler to be tested.

image_data(*dataset_id*, *image_data_ir1*, *image_data_vis*)

Get VISSR image data.

image_data_ir1()

Get IR1 image data.

image_data_vis()

Get VIS image data.

image_parameters(*mode_block*, *cal_params*, *nav_params*)

Get VISSR image parameters.

ir1_bt_exp(*lons_lats_exp*)

Get expected IR1 brightness temperature.

ir1_calibration()

Get IR1 calibration block.

ir1_counts_exp(*lons_lats_exp*)

Get expected IR1 counts.

ir2_calibration()

Get IR2 calibration block.

lons_lats_exp(*dataset_id*)

Get expected lon/lat coordinates.

Computed with JMA's Msial library for 2 pixels near the central column (6688.5/1672.5 for VIS/IR).

VIS:

pix = [6688, 6688, 6689, 6689] lin = [2744, 8356, 2744, 8356]

IR1:

pix = [1672, 1672, 1673, 1673] lin = [686, 2089, 686, 2089]

mask_space(*request*)

Mask space pixels.

mode_block()

Get VISSR mode block.

nav_params(*coordinate_conversion*, *attitude_prediction*, *orbit_prediction*)

Get navigation parameters.

open_function(*with_compression*)

Get open function for writing test files.

orbit_prediction(*orbit_prediction_1*, *orbit_prediction_2*)

Get predictions of orbital parameters.

orbit_prediction_1()

Get first block of orbit prediction data.

orbit_prediction_2()

Get second block of orbit prediction data.

simple_coord_conv_table()

Get simple coordinate conversion table.

test_get_dataset(*file_handler*, *dataset_id*, *dataset_exp*, *attrs_exp*)

Test getting the dataset.

test_time_attributes(*file_handler*, *attrs_exp*)

Test the file handler's time attributes.

vis_calibration()

Get VIS calibration block.

vis_refl_exp(*mask_space*, *lons_lats_exp*)

Get expected VIS reflectance.

vissr_file(*dataset_id*, *file_contents*, *open_function*, *tmp_path*)

Get test VISSR file.

vissr_file_like(*vissr_file*, *with_compression*)

Get file-like object for VISSR test file.

with_compression(*request*)

Enable compression.

wv_calibration()

Get WV calibration block.

class satpy.tests.reader_tests.gms.test_gms5_vissr_l1b.**VissrFileWriter**(*ch_type*, *open_function*)

Bases: object

Write data in VISSR archive format.

Initialize the writer.

Parameters

- **ch_type** – Channel type (VIS or IR)
- **open_function** – Open function to be used (e.g. open or gzip.open)

_fill(*fd*, *target_byte*)

Write placeholders from current position to target byte.

_write(*fd*, *data*, *offset=None*)

Write data to file.

If specified, prepend with 'offset' placeholder bytes.

_write_control_block(*fd*, *contents*)

_write_image_data(*fd*, *contents*)

_write_image_parameter(*fd*, *im_param*, *name*)

_write_image_parameters(*fd*, *contents*)

image_params_order = ['mode', 'coordinate_conversion', 'attitude_prediction', 'orbit_prediction_1', 'orbit_prediction_2', 'vis_calibration', 'ir1_calibration', 'ir2_calibration', 'wv_calibration', 'simple_coordinate_conversion_table']

write(*filename*, *contents*)

Write file contents to disk.

satpy.tests.reader_tests.gms.test_gms5_vissr_l1b.**_disable_jit**(*request*, *monkeypatch*)

Run tests with jit enabled and disabled.

Reason: Coverage report is only accurate with jit disabled.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation module

Unit tests for GMS-5 VISSR navigation.

class satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**TestImageNavigation**

Bases: object

Test navigation of an entire image.

expected()

Get expected coordinates.

test_get_lons_lats(*navigation_params*, *expected*)

Test getting lon/lat coordinates.

class satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**TestPredictionInterpolation**

Bases: object

Test interpolation of orbit and attitude predictions.

attitude_expected()

Get expected attitude.

obs_time()

Get observation time.

orbit_expected()

Get expected orbit.

test_interpolate_angles(*obs_time*, *expected*)

Test interpolation of periodic angles.

test_interpolate_attitude_prediction(*obs_time*, *attitude_prediction*, *attitude_expected*)

Test interpolating attitude prediction.

test_interpolate_continuous(*obs_time*, *expected*)

Test interpolation of continuous variables.

test_interpolate_nearest(*obs_time*, *expected*)

Test nearest neighbour interpolation.

test_interpolate_orbit_prediction(*obs_time*, *orbit_prediction*, *orbit_expected*)

Test interpolating orbit prediction.

class satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**TestSinglePixelNavigation**

Bases: object

Test navigation of a single pixel.

test_get_lon_lat(*point*, *nav_params*, *expected*)

Test getting lon/lat coordinates for a given pixel.

test_intersect_view_vector_with_earth()

Test intersection of a view vector with the earth's surface.

test_normalize_vector()

Test vector normalization.

test_transform_earth_fixed_to_geodetic_coords(*point_earth_fixed*, *point_geodetic_exp*)

Test transformation from earth-fixed to geodetic coordinates.

test_transform_image_coords_to_scanning_angles()

Test transformation from image coordinates to scanning angles.

test_transform_satellite_to_earth_fixed_coords()

Test transformation from satellite to earth-fixed coordinates.

test_transform_scanning_angles_to_satellite_coords()

Test transformation from scanning angles to satellite coordinates.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**_assert_namedtuple_close**(*a*, *b*)

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**_disable_jit**(*request*, *monkeypatch*)

Run tests with jit enabled and disabled.

Reason: Coverage report is only accurate with jit disabled.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**_is_namedtuple**(*obj*)

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**attitude_prediction**()

Get attitude prediction.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**navigation_params**(*static_nav_params*, *predicted_nav_params*)

Get image navigation parameters.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**orbit_prediction**()

Get orbit prediction.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**predicted_nav_params**(*attitude_prediction*, *orbit_prediction*)

Get predicted navigation parameters.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**proj_params**(*sampling_angle*)

Get projection parameters.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**sampling_angle**()

Get sampling angle.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**scan_params**(*sampling_angle*)

Get scanning parameters.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**static_nav_params**(*proj_params*, *scan_params*)

Get static navigation parameters.

satpy.tests.reader_tests.gms.test_gms5_vissr_navigation.**test_get_observation_time**()

Test getting a pixel's observation time.

Module contents

Unit tests for GMS reader.

satpy.tests.reader_tests.modis_tests package

Submodules

satpy.tests.reader_tests.modis_tests._modis_fixtures module

MODIS L1b and L2 test fixtures.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_add_variable_to_file**(*h*, *var_name*, *var_info*)

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_create_core_metadata**(*file_shortname: str*) → str

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_create_header_metadata**() → str

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_create_struct_metadata**(*geo_resolution: int*) → str

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_create_struct_metadata_cmg**(*ftype: str*) → str

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_generate_angle_data**(*resolution: int*) → ndarray

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_generate_lonlat_data**(*resolution: int*) → tuple[ndarray, ndarray]

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_generate_visible_data**(*resolution: int*, *num_bands: int*, *dtype=<class 'numpy.uint16'>*) → ndarray

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_generate_visible_uncertainty_data**(*shape: tuple*) → ndarray

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_angles_variable_info**(*resolution: int*) → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_basic_variable_info**(*var_name: str*, *resolution: int*) → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_cloud_mask_variable_info**(*var_name: str, resolution: int*) → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_emissive_variable_info**(*var_name: str, resolution: int, bands: list[str]*)

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_l1b_geo_variable_info**(*filename: str, geo_resolution: int, include_angles: bool = True*) → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_l3_refl_variable_info**(*var_name: str*) → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_lonlat_variable_info**(*resolution: int*) → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_mask_byte1_variable_info**() → dict

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_get_visible_variable_info**(*var_name: str, resolution: int, bands: list[str]*)

satpy.tests.reader_tests.modis_tests._modis_fixtures.**_shape_for_resolution**(*resolution: int*) → tuple[int, int]

satpy.tests.reader_tests.modis_tests._modis_fixtures.**create_hdfeos_test_file**(*filename: str, variable_infos: dict, struct_meta: str | None = None, core_meta: str | None = None, archive_meta: str | None = None*) → None

Create a fake MODIS L1b HDF4 file with headers.

Parameters

- **filename** – Full path of filename to be created.
- **variable_infos** – Dictionary mapping HDF4 variable names to dictionary of variable information (see `_add_variable_to_file`).

- **struct_meta** – Contents of the 'StructMetadata.0' header.
- **core_meta** – Contents of the 'CoreMetadata.0' header.
- **archive_meta** – Contents of the 'ArchiveMetadata.0' header.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**generate_imapp_filename**(*suffix*)

Generate a filename that follows IMAPP MODIS L1b convention.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**generate_nasa_l1b_filename**(*prefix*)

Generate a filename that follows NASA MODIS L1b convention.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**generate_nasa_l2_filename**(*prefix: str*) → str

Generate a file name that follows MODIS 35 L2 convention in a temporary directory.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**generate_nasa_l3_filename**(*prefix: str*) → str

Generate a file name that follows MODIS 09 L3 convention in a temporary directory.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_imapp_1000m_file**(*tmpdir_factory*) → list[str]

Create a single MOD021KM file following IMAPP file scheme.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_imapp_geo_file**(*tmpdir_factory*) → list[str]

Create a single geo file following standard IMAPP file scheme.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_nasa_1km_mod03_files**(*modis_l1b_nasa_mod021 modis_l1b_nasa_mod03_* → list[str]

Create input files including the 1KM and MOD03 files.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_nasa_mod021km_file**(*tmpdir_factory*) → list[str]

Create a single MOD021KM file following standard NASA file scheme.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_nasa_mod02hkm_file**(*tmpdir_factory*) → list[str]

Create a single MOD02HKM file following standard NASA file scheme.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_nasa_mod02qkm_file**(*tmpdir_factory*) → list[str]

Create a single MOD02QKM file following standard NASA file scheme.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l1b_nasa_mod03_file**(*tmpdir_factory*) → list[str]

Create a single MOD03 file following standard NASA file scheme.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_imapp_mask_byte1_file**(*tmpdir_factory*) → list[str]

Create a single IMAPP mask_byte1 L2 HDF4 file with headers.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_imapp_mask_byte1_geo_files**(*modis_l2_imapp_m*
modis_l1b_nasa_m
→ list[str]

Create the IMAPP mask_byte1 and geo HDF4 files.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_imapp_snowmask_file**(*tmpdir_factory*)
→ list[str]

Create a single IMAPP snowmask L2 HDF4 file with headers.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_imapp_snowmask_geo_files**(*modis_l2_imapp_sno*
modis_l1b_nasa_mod
→ list[str]

Create the IMAPP snowmask and geo HDF4 files.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_nasa_mod06_file**(*tmpdir_factory*)
→ list[str]

Create a single MOD06 L2 HDF4 file with headers.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_nasa_mod35_file**(*tmpdir_factory*)
→ list[str]

Create a single MOD35 L2 HDF4 file with headers.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l2_nasa_mod35_mod03_files**(*modis_l2_nasa_mod35_*
modis_l1b_nasa_mod03
→ list[str]

Create a MOD35 L2 HDF4 file and MOD03 L1b geolocation file.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l3_file**(*tmpdir_factory*, *f_prefix*, *var_name*, *f_short*)

Create a MODIS L3 file of the desired type.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l3_nasa_mod09_file**(*tmpdir_factory*)
→ list[str]

Create a single MOD09 L3 HDF4 file with headers.

satpy.tests.reader_tests.modis_tests._modis_fixtures.**modis_l3_nasa_mod43_file**(*tmpdir_factory*)
→ list[str]

Create a single MVCD43 L3 HDF4 file with headers.

satpy.tests.reader_tests.modis_tests.conftest module

Setup and configuration for all reader tests.

satpy.tests.reader_tests.modis_tests.test_modis_l1b module

Unit tests for MODIS L1b HDF reader.

class satpy.tests.reader_tests.modis_tests.test_modis_l1b.**TestModisL1b**

Bases: object

Test MODIS L1b reader.

test_available_reader()

Test that MODIS L1b reader is available.

test_load_longitude_latitude(*input_files*, *has_5km*, *has_500*, *has_250*, *default_res*)

Test that longitude and latitude datasets are loaded correctly.

test_load_sat_zenith_angle(*modis_l1b_nasa_mod021km_file*)

Test loading satellite zenith angle band.

test_load_vis(*modis_l1b_nasa_mod021km_file*)

Test loading visible band.

test_load_vis_saturation(*mask_saturated*, *modis_l1b_nasa_mod021km_file*)

Test loading visible band.

test_scene_available_datasets(*input_files*, *expected_names*, *expected_data_res*, *expected_geo_res*)

Test that datasets are available.

satpy.tests.reader_tests.modis_tests.test_modis_l1b.**_check_shared_metadata**(*data_arr*)

satpy.tests.reader_tests.modis_tests.test_modis_l1b.**_load_and_check_geolocation**(*scene*, *resolution*, *exp_res*, *exp_shape*, *has_res*, *check_callback=<function _check_shared_metadata>*)

satpy.tests.reader_tests.modis_tests.test_modis_l2 module

Unit tests for MODIS L2 HDF reader.

class satpy.tests.reader_tests.modis_tests.test_modis_l2.**TestModisL2**

Bases: object

Test MODIS L2 reader.

test_available_reader()

Test that MODIS L2 reader is available.

test_load_250m_cloud_mask_dataset(*input_files*, *exp_area*)

Test loading 250m cloud mask.

test_load_category_dataset(*input_files*, *loadables*, *request_resolution*, *exp_resolution*, *exp_area*)

Test loading category products.

test_load_l2_dataset(*input_files*, *loadables*, *exp_resolution*, *exp_area*, *exp_value*)

Load and check an L2 variable.

test_load_longitude_latitude(*input_files*, *has_5km*, *has_500*, *has_250*, *default_res*)

Test that longitude and latitude datasets are loaded correctly.

test_load_quality_assurance(*modis_l2_nasa_mod35_file*)

Test loading quality assurance.

test_scene_available_datasets(*modis_l2_nasa_mod35_file*)

Test that datasets are available.

satpy.tests.reader_tests.modis_tests.test_modis_l2.**_check_shared_metadata**(*data_arr*, *expect_area=False*)

satpy.tests.reader_tests.modis_tests.test_modis_l3 module

Unit tests for MODIS L3 HDF reader.

class satpy.tests.reader_tests.modis_tests.test_modis_l3.**TestModisL3**

Bases: object

Test MODIS L3 reader.

test_available_reader()

Test that MODIS L3 reader is available.

test_load_l3_dataset(*modis_l3_nasa_mod09_file*)

Load and check an L2 variable.

test_scene_available_datasets(*loadable*, *filename*)

Test that datasets are available.

satpy.tests.reader_tests.modis_tests.test_modis_l3.**_expected_area**()

Module contents

Unit tests for MODIS readers.

This subdirectory mostly exists to have MODIS-based pytest fixtures only loaded for MODIS tests.

Submodules

satpy.tests.reader_tests._li_test_utils module

Common utility modules used for LI mock-oriented unit tests.

class satpy.tests.reader_tests._li_test_utils.**FakeLIFileHandlerBase**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Class for faking the NetCDF4 Filehandler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Get the content of the test data.

Here we generate the default content we want to provide depending on the provided filename infos.

get_variable_writer(*dset*, *settings*)

Get a variable writer.

schema_parameters = None

write_sector_variables(*settings*, *write_variable*)

Write the sector variables.

write_variables(*settings*, *write_variable*)

Write raw (i.e. not in sectors) variables.

satpy.tests.reader_tests._li_test_utils.**accumulation_dimensions**(*nacc*, *nobs*)

Set dimensions for the accumulated products.

satpy.tests.reader_tests._li_test_utils.**add_attributes**(*attribs*, *ignored_attrs*, *desc*)

Add all the custom properties directly as attributes.

satpy.tests.reader_tests._li_test_utils.**extract_filetype_info**(*filetype_infos*, *filetype*)

Extract Satpy-conform filetype_info from filetype_infos fixture.

satpy.tests.reader_tests._li_test_utils.**fci_grid_definition**(*axis*, *nobs*)

FCI grid definition on X or Y axis.

satpy.tests.reader_tests._li_test_utils.**get_product_schema**(*pname*, *settings=None*)

Retrieve an LI product schema given its name.

satpy.tests.reader_tests._li_test_utils.**l2_af_schema**(*settings=None*)

Define schema for LI L2 AF product.

satpy.tests.reader_tests._li_test_utils.**l2_afa_schema**(*settings=None*)

Define schema for LI L2 AFA product.

satpy.tests.reader_tests._li_test_utils.**l2_afr_schema**(*settings=None*)

Define schema for LI L2 AFR product.

satpy.tests.reader_tests._li_test_utils.**l2_le_schema**(*settings=None*)

Define schema for LI L2 LE product.

satpy.tests.reader_tests._li_test_utils.**l2_lef_schema**(*settings=None*)

Define schema for LI L2 LEF product.

satpy.tests.reader_tests._li_test_utils.**l2_lfl_schema**(*settings=None*)

Define schema for LI L2 LFL product.

satpy.tests.reader_tests._li_test_utils.**l2_lgr_schema**(*settings=None*)

Define schema for LI L2 LGR product.

satpy.tests.reader_tests._li_test_utils.**mtg_geos_projection**()

MTG geos projection definition.

satpy.tests.reader_tests._li_test_utils.**populate_dummy_data**(*data*, *names*, *details*)

Populate variable with dummy data.

satpy.tests.reader_tests._li_test_utils.**set_variable_path**(*var_path*, *desc*, *sname*)

Replace variable default path if applicable and ensure trailing separator.

satpy.tests.reader_tests.conftest module

Setup and configuration for all reader tests.

satpy.tests.reader_tests.test_aapp_l1b module

Test module for the avhrr aapp l1b reader.

class satpy.tests.reader_tests.test_aapp_l1b.**TestAAPPL1BAllChannelsPresent**(*methodName='runTest'*)

Bases: TestCase

Test the filehandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_angles()

Test reading the angles.

test_interpolation()

Test reading the lon and lats.

test_interpolation_angles()

Test reading the lon and lats.

test_navigation()

Test reading the lon and lats.

test_read()

Test the reading.

class satpy.tests.reader_tests.test_aapp_l1b.**TestAAPPL1BChannel3AMissing**(*methodName='runTest'*)

Bases: TestCase

Test the filehandler when channel 3a is missing.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_available_datasets_miss_3a()

Test that channel 3a is missing from available datasets.

test_loading_missing_channels_returns_none()

Test that loading a missing channel raises a keyerror.

class satpy.tests.reader_tests.test_aapp_l1b.**TestNegativeCalibrationSlope**(*methodName='runTest'*)

Bases: TestCase

Case for testing correct behaviour when the data has negative slope2 coefficients.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

tearDown()

Tear down the test case.

test_bright_channel2_has_reflectance_greater_than_100()

Test that a bright channel 2 has reflectances greater that 100.

satpy.tests.reader_tests.test_aapp_mhs_amsub_l1c module

Test module for the MHS AAPP level-1c reader.

class satpy.tests.reader_tests.test_aapp_mhs_amsub_l1c.**TestMHS_AMSUB_AAPPL1CReadData**(*methodName='runTest'*)

Bases: TestCase

Test the filehandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_angles()

Test reading the angles.

test_navigation()

Test reading the longitudes and latitudes.

test_platform_name()

Test getting the platform name.

test_read()

Test getting the platform name.

test_sensor_name()

Test getting the sensor name.

satpy.tests.reader_tests.test_abi_l1b module

The abi_l1b reader tests package.

class satpy.tests.reader_tests.test_abi_l1b.**Test_NC_ABI_L1B**

Bases: object

Test the NC_ABI_L1B reader.

pytestmark = [Mark(name='parametrize', args=('c01_data_arr', [<LazyFixture "c01_rad">, <LazyFixture "c01_rad_h5netcdf">]), kwargs={})]

test_get_dataset(*c01_data_arr*)

Test the get_dataset method.

satpy.tests.reader_tests.test_abi_l1b.**_apply_dask_chunk_size**()

satpy.tests.reader_tests.test_abi_l1b.**_check_area**(*data_arr: DataArray*) → None

satpy.tests.reader_tests.test_abi_l1b.**_check_dims_and_coords**(*data_arr: DataArray*) → None

satpy.tests.reader_tests.test_abi_l1b.**_create_fake_rad_dataarray**(*rad: DataArray | None = None, resolution: int = 2000*) → DataArray

satpy.tests.reader_tests.test_abi_l1b.**_create_fake_rad_dataset**(*rad: DataArray, resolution: int*) → Dataset

satpy.tests.reader_tests.test_abi_l1b.**_create_reader_for_data**(*tmp_path: Path, channel_name: str, rad: DataArray | None, resolution: int, reader_kwargs: dict[str, Any] | None = None*) → *FileYAMLReader*

satpy.tests.reader_tests.test_abi_l1b.**_fake_c07_data**() → DataArray

satpy.tests.reader_tests.test_abi_l1b.**_get_and_check_array**(*data_arr: DataArray, exp_dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) → ndarray[Any, dtype[_ScalarType_co]]

satpy.tests.reader_tests.test_abi_l1b.**c01_counts**(*tmp_path*) → DataArray

Load c01 counts.

satpy.tests.reader_tests.test_abi_l1b.**c01_rad**(*tmp_path*) → DataArray

Load c01 radiances.

satpy.tests.reader_tests.test_abi_l1b.**c01_rad_h5netcdf**(*tmp_path*) → DataArray

Load c01 radiances through h5netcdf.

satpy.tests.reader_tests.test_abi_l1b.**c01_refl**(*tmp_path*) → DataArray

Load c01 reflectances.

satpy.tests.reader_tests.test_abi_l1b.**c07_bt_creator**(*tmp_path*) → Callable

Create a loader for c07 brightness temperatures.

satpy.tests.reader_tests.test_abi_l1b.**generate_l1b_filename**(*chan_name: str*) → str

Generate a l1b filename.

satpy.tests.reader_tests.test_abi_l1b.**test_file_patterns_match**(*channel*, *suffix*)

Test that the configured file patterns work.

satpy.tests.reader_tests.test_abi_l1b.**test_ir_calibrate**(*c07_bt_creator*, *clip_negative_radiances*)

Test IR calibration.

satpy.tests.reader_tests.test_abi_l1b.**test_open_dataset**(_)

Test opening a dataset.

satpy.tests.reader_tests.test_abi_l1b.**test_raw_calibrate**(*c01_counts*)

Test RAW calibration.

satpy.tests.reader_tests.test_abi_l1b.**test_vis_calibrate**(*c01_refl*)

Test VIS calibration.

satpy.tests.reader_tests.test_abi_l2_nc module

The abi_l2_nc reader tests package.

class satpy.tests.reader_tests.test_abi_l2_nc.**TestMCMIPReading**

Bases: object

Test cases of the MCMIP file format.

test_mcmip_get_dataset(*xr_*, *product*, *exp_metadata*)

Test getting channel from MCMIP file.

class satpy.tests.reader_tests.test_abi_l2_nc.**Test_NC_ABI_L2_area_AOD**

Bases: object

Test the NC_ABI_L2 reader for the AOD product.

setup_method(*xr_*)

Create fake data for the tests.

test_get_area_def_xy(*adef*)

Test the area generation.

class satpy.tests.reader_tests.test_abi_l2_nc.**Test_NC_ABI_L2_area_fixedgrid**

Bases: object

Test the NC_ABI_L2 reader.

test_get_area_def_fixedgrid(*adef*)

Test the area generation.

class satpy.tests.reader_tests.test_abi_l2_nc.**Test_NC_ABI_L2_area_latlon**

Bases: object

Test the NC_ABI_L2 reader.

setup_method()

Create fake data for the tests.

test_get_area_def_latlon(*adef*)

Test the area generation.

class satpy.tests.reader_tests.test_abi_l2_nc.**Test_NC_ABI_L2_get_dataset**

Bases: object

Test get dataset function of the NC_ABI_L2 reader.

test_get_dataset()

Test basic L2 load.

test_get_dataset_gfls()

Test that Low Cloud and Fog filenames work.

satpy.tests.reader_tests.test_abi_l2_nc.**_assert_orbital_parameters**(*orb_params*)

satpy.tests.reader_tests.test_abi_l2_nc.**_compare_subdict**(*actual_dict*, *exp_sub_dict*)

satpy.tests.reader_tests.test_abi_l2_nc.**_create_cmip_dataset**(*data_variable: str = 'HT'*)

satpy.tests.reader_tests.test_abi_l2_nc.**_create_mcmip_dataset**()

satpy.tests.reader_tests.test_abi_l2_nc.**_create_reader_for_fake_data**(*observation_type: str*, *fake_dataset: Dataset*, *filename_info: dict | None = None*)

satpy.tests.reader_tests.test_acspo module

Module for testing the satpy.readers.acspo module.

class satpy.tests.reader_tests.test_acspo.**FakeNetCDF4FileHandler2**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_acspo.**TestACSPOReader**

Bases: object

Test ACSPO Reader.

setup_method()

Wrap NetCDF4 file handler with our own fake handler.

teardown_method()

Stop wrapping the NetCDF4 file handler.

test_init(*filename*)

Test basic init with no extra parameters.

test_load_every_dataset()

Test loading all datasets.

yaml_file = 'acspo.yaml'

satpy.tests.reader_tests.test_agri_l1 module

The agri_l1 reader tests package.

class satpy.tests.reader_tests.test_agri_l1.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_create_channel_data(*chs*, *cwls*, *file_type*)

_create_coeff_array(*nb_channels*)

_get_1km_data(*file_type*)

_get_2km_data(*file_type*)

_get_4km_data(*file_type*)

_get_500m_data(*file_type*)

_get_geo_data(*file_type*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

make_test_data(*cwl*, *ch*, *prefix*, *dims*, *file_type*)

Make test data.

class satpy.tests.reader_tests.test_agri_l1.**Test_HDF_AGRI_L1_cal**

Bases: object

Test VIRR L1B Reader.

static _assert_which_channels_are_loaded(*available_datasets*, *band_names*, *resolution_to_test*)

_check_calibration_and_units(*band_names*, *result*)

static _check_keys_for_dsq(*available_datasets*, *resolution_to_test*)

static _check_units(*band_name*, *result*)

_create_reader_for_resolutions(**resolutions*)

setup_method()

Wrap HDF5 file handler with our own fake handler.

teardown_method()

Stop wrapping the HDF5 file handler.

test_agri_all_bands_have_right_units()

Test all bands have the right units.

test_agri_counts_calibration()

Test loading data at counts calibration.

test_agri_for_one_resolution(*resolution_to_test*, *satname*)

Test loading data when only one resolution is available.

test_agri_geo(*satname*)

Test loading data for angles.

test_agri_orbital_parameters_are_correct()

Test orbital parameters are set correctly.

test_fy4a_channels_are_loaded_with_right_resolution()

Test all channels are loaded with the right resolution.

test_times_correct()

Test that the reader handles the two possible time formats correctly.

yaml_file = 'agri_fy4a_l1.yaml'

satpy.tests.reader_tests.test_agri_l1.**_create_filenames_from_resolutions**(*satname*, **resolutions*)

Create filenames from the given resolutions.

satpy.tests.reader_tests.test_ahi_hrit module

The hrit ahi reader tests package.

class satpy.tests.reader_tests.test_ahi_hrit.**TestHRITJMAFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRITJMAFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_acq_time(*nlines*)

Get sample header entry for scanline acquisition times.

Lines: 1, 21, 41, 61, …, nlines Times: 1970-01-01 00:00 + (1, 21, 41, 61, …, nlines) seconds

So the interpolated times are expected to be 1970-01-01 + (1, 2, 3, 4, …, nlines) seconds. Note that there will be some floating point inaccuracies, because timestamps are stored with only 6 decimals precision.

_get_mda(*loff=5500.0*, *coff=5500.0*, *nlines=11000*, *ncols=11000*, *segno=0*, *numseg=1*, *vis=True*, *platform='Himawari-8'*)

Create metadata dict like HRITFileHandler would do it.

_get_reader(*mocked_init*, *mda*, *filename_info=None*, *filetype_info=None*, *reader_kwargs=None*)

test_calibrate()

Test calibration.

test_get_acq_time()

Test computation of scanline acquisition times.

test_get_area_def()

Test getting an AreaDefinition.

test_get_dataset(*base_get_dataset*)

Test getting a dataset.

test_get_platform(*mocked_init*)

Test platform identification.

test_init()

Test creating the file handler.

test_mask_space()

Test masking of space pixels.

test_mjd2datetime64()

Test conversion from modified julian day to datetime64.

test_start_time_from_aqc_time()

Test that by the datetime from the metadata returned when *use_acquisition_time_as_start_time=True*.

test_start_time_from_filename()

Test that by default the datetime in the filename is returned.

satpy.tests.reader_tests.test_ahi_hsd module

The ahi_hsd reader tests package.

class satpy.tests.reader_tests.test_ahi_hsd.**TestAHICalibration**(*methodName='runTest'*)

Bases: TestCase

Test case for various AHI calibration types.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(**mocks*)

Create fake data for testing.

test_default_calibrate(**mocks*)

Test default in-file calibration modes.

test_updated_calibrate()

Test updated in-file calibration modes.

test_user_calibration()

Test user-defined calibration modes.

class satpy.tests.reader_tests.test_ahi_hsd.**TestAHIHSDFileHandler**

Bases: object

Tests for the AHI HSD file handler.

test_actual_satellite_position(*round_actual_position*, *expected_result*)

Test that rounding of the actual satellite position can be controlled.

test_bad_calibration()

Test that a bad calibration mode causes an exception.

test_blocklen_error(**mocks*)

Test erraneous blocklength.

test_is_valid_time()

Test that valid times are correctly identified.

test_read_band(*calibrate*, **mocks*)

Test masking of space pixels.

test_read_band_from_actual_file(*hsd_file_jp01*)

Test read_bands on real data.

test_read_header(**mocks*)

Test header reading.

test_scanning_frequencies()

Test scanning frequencies.

test_scene_loading(*calibrate*, **mocks*)

Test masking of space pixels.

test_time_properties()

Test start/end/scheduled time properties.

test_time_rounding()

Test rounding of the nominal time.

class satpy.tests.reader_tests.test_ahi_hsd.**TestAHIHSDNavigation**(*methodName='runTest'*)

Bases: TestCase

Test the AHI HSD reader navigation.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_region(*fromfile*, *np2str*)

Test region navigation.

test_segment(*fromfile*, *np2str*)

Test segment navigation.

satpy.tests.reader_tests.test_ahi_hsd.**_create_fake_file_handler**(*in_fname*, *filename_info=None*, *filetype_info=None*, *fh_kwargs=None*)

satpy.tests.reader_tests.test_ahi_hsd.**_custom_fromfile**(**args*, ***kwargs*)

satpy.tests.reader_tests.test_ahi_hsd.**_fake_hsd_handler**(*fh_kwargs=None*)

Create a test file handler.

satpy.tests.reader_tests.test_ahi_hsd.**_new_unzip**(*fname*, *prefix=''*)

Fake unzipping.

satpy.tests.reader_tests.test_ahi_hsd.**hsd_file_jp01**(*tmp_path*)

Create a jp01 hsd file.

satpy.tests.reader_tests.test_ahi_l1b_gridded_bin module

The ahi_l1b_gridded_bin reader tests package.

class satpy.tests.reader_tests.test_ahi_l1b_gridded_bin.**TestAHIGriddedArea**(*methodName='runTest'*)

Bases: TestCase

Test the AHI gridded reader definition.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

static make_fh(*filetype*, *area='fld'*)

Create a test file handler.

setUp()

Create fake data for testing.

test_area_def()

Check that a valid full disk area is produced.

test_bad_area()

Ensure an error is raised for an usupported area.

test_hi_res()

Check size of the low resolution (0.5km) grid.

test_low_res()

Check size of the low resolution (2km) grid.

test_med_res()

Check size of the low resolution (1km) grid.

class satpy.tests.reader_tests.test_ahi_l1b_gridded_bin.**TestAHIGriddedFileCalibration**(*methodName='runTest*

Bases: TestCase

Test case for the file calibration types.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create a test file handler.

test_calibrate(*np_loadtxt*, *os_exist*, *get_luts*)

Test the calibration modes of AHI using the LUTs.

class satpy.tests.reader_tests.test_ahi_l1b_gridded_bin.**TestAHIGriddedFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test case for the file reading.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

new_unzip()

Fake unzipping.

setUp()

Create a test file handler.

test_dataread(*memmap*)

Check that a dask array is returned from the read function.

test_destructor(*exist_patch*, *remove_patch*)

Check that file handler deletes files if needed.

test_get_dataset(*mocked_read*)

Check that a good dataset is returned on request.

class satpy.tests.reader_tests.test_ahi_l1b_gridded_bin.**TestAHIGriddedLUTs**(*methodName='runTest'*)

Bases: TestCase

Test case for the downloading and preparing LUTs.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

mocked_ftp_dl()

Fake download of LUT tar file by creating a local tar.

setUp()

Create a test file handler.

tearDown()

Remove files and directories created by the tests.

test_download_luts(*mock_dl*, *mock_shutil*)

Test that the FTP library is called for downloading LUTS.

test_get_luts()

Check that the function to download LUTs operates successfully.

satpy.tests.reader_tests.test_ahi_l2_nc module

Tests for the Himawari L2 netCDF reader.

satpy.tests.reader_tests.test_ahi_l2_nc.**ahil2_filehandler**(*fname*, *platform='h09'*)

Instantiate a Filehandler.

satpy.tests.reader_tests.test_ahi_l2_nc.**himl2_filename**(*tmp_path_factory*)

Create a fake himawari l2 file.

satpy.tests.reader_tests.test_ahi_l2_nc.**himl2_filename_bad**(*tmp_path_factory*)

Create a fake himawari l2 file.

satpy.tests.reader_tests.test_ahi_l2_nc.**test_ahi_l2_area_def**(*himl2_filename*, *caplog*)

Test reader handles area definition correctly.

satpy.tests.reader_tests.test_ahi_l2_nc.**test_bad_area_name**(*himl2_filename_bad*)

Check case where area name is not correct.

satpy.tests.reader_tests.test_ahi_l2_nc.**test_load_data**(*himl2_filename*)

Test that data is loaded successfully.

satpy.tests.reader_tests.test_ahi_l2_nc.**test_startend**(*himl2_filename*)

Test start and end times are set correctly.

satpy.tests.reader_tests.test_ami_l1b module

The ami_l1b reader tests package.

class satpy.tests.reader_tests.test_ami_l1b.**FakeDataset**(*info*, *attrs*)

Bases: object

Mimic xarray Dataset object.

Initialize test data.

close()

Act like close method.

rename(**args*, ***kwargs*)

Mimic rename method.

class satpy.tests.reader_tests.test_ami_l1b.**TestAMIL1bNetCDF**(*methodName='runTest'*)

Bases: *TestAMIL1bNetCDFBase*

Test the AMI L1b reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_check_orbital_parameters(*orb_params*)

Check that orbital parameters match expected values.

_classSetupFailed = False

_class_cleanups = []

test_bad_calibration()

Test that asking for a bad calibration fails.

test_basic_attributes()

Test getting basic file attributes.

test_filename_grouping()

Test that filenames are grouped properly.

test_get_area_def(*adef*)

Test the area generation.

test_get_dataset()

Test gettting radiance data.

test_get_dataset_counts()

Test get counts data.

test_get_dataset_vis()

Test get visible calibrated data.

class satpy.tests.reader_tests.test_ami_l1b.**TestAMIL1bNetCDFBase**(*methodName='runTest'*)

Bases: TestCase

Common setup for NC_ABI_L1B tests.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*xr_*, *counts=None*)

Create a fake dataset using the given counts data.

class satpy.tests.reader_tests.test_ami_l1b.**TestAMIL1bNetCDFIRCal**(*methodName='runTest'*)

Bases: *TestAMIL1bNetCDFBase*

Test IR specific things about the AMI reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data for IR calibration tests.

test_default_calibrate()

Test default (pyspectral) IR calibration.

test_gsics_radiance_corr()

Test IR radiance adjustment using in-file GSICS coefs.

test_infile_calibrate()

Test IR calibration using in-file coefficients.

test_user_radiance_corr()

Test IR radiance adjustment using user-supplied coefs.

satpy.tests.reader_tests.test_amsr2_l1b module

Module for testing the satpy.readers.amsr2_l1b module.

class satpy.tests.reader_tests.test_amsr2_l1b.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_amsr2_l1b.**TestAMSR2L1BReader**(*methodName='runTest'*)

Bases: TestCase

Test AMSR2 L1B Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_init()

Test basic init with no extra parameters.

test_load_89ghz()

Test loading of 89GHz channels.

test_load_basic()

Test loading of basic channels.

yaml_file = 'amsr2_l1b.yaml'

satpy.tests.reader_tests.test_amsr2_l2 module

Unit tests for AMSR L2 reader.

class satpy.tests.reader_tests.test_amsr2_l2.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_amsr2_l2.**TestAMSR2L2Reader**(*methodName='runTest'*)

Bases: TestCase

Test AMSR2 L2 Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_init()

Test basic init with no extra parameters.

test_load_basic()

Test loading of basic channels.

yaml_file = 'amsr2_l2.yaml'

satpy.tests.reader_tests.test_amsr2_l2_gaasp module

Tests for the 'amsr2_l2_gaasp' reader.

class satpy.tests.reader_tests.test_amsr2_l2_gaasp.**TestGAASPReader**

Bases: object

Tests for the GAASP reader.

static _check_area(*data_id*, *data_arr*)

static _check_attrs(*data_arr*)

static _check_fill(*data_id*, *data_arr*)

setup_method()

Wrap pygrib to read fake data.

test_available_datasets(*filenames*, *expected_datasets*)

Test that variables are dynamically discovered.

test_basic_load(*filenames*, *loadable_ids*)

Test that variables are loaded properly.

test_reader_creation(*filenames*, *expected_loadables*)

Test basic initialization.

yaml_file = 'amsr2_l2_gaasp.yaml'

satpy.tests.reader_tests.test_amsr2_l2_gaasp.**_create_gridded_gaasp_dataset**(*filename*)

Represent files with gridded products.

satpy.tests.reader_tests.test_amsr2_l2_gaasp.**_create_one_res_gaasp_dataset**(*filename*)

Represent files with one resolution of variables in them (ex. SOIL).

satpy.tests.reader_tests.test_amsr2_l2_gaasp.**_create_two_res_gaasp_dataset**(*filename*)

Represent files with two resolution of variables in them (ex. OCEAN).

satpy.tests.reader_tests.test_amsr2_l2_gaasp.**_get_shared_global_attrs**(*filename*)

satpy.tests.reader_tests.test_amsr2_l2_gaasp.**fake_open_dataset**(*filename*, ***kwargs*)

Create a Dataset similar to reading an actual file with xarray.open_dataset.

satpy.tests.reader_tests.test_ascat_l2_soilmoisture_bufr module

Unittesting the ASCAT SCATTEROMETER SOIL MOISTURE BUFR reader.

class satpy.tests.reader_tests.test_ascat_l2_soilmoisture_bufr.**TesitAscatL2SoilmoistureBufr**(*methodName=*

Bases: TestCase

Test ASCAT Soil Mosture loader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create temporary file to perform tests with.

tearDown()

Remove the temporary directory created for a test.

test_scene()

Test scene creation.

test_scene_dataset_values()

Test loading data.

test_scene_load_available_datasets()

Test that all datasets are available.

satpy.tests.reader_tests.test_ascat_l2_soilmoisture_bufr.**create_message**()

Create fake message for testing.

satpy.tests.reader_tests.test_ascat_l2_soilmoisture_bufr.**save_test_data**(*path*)

Save the test file to the indicated directory.

satpy.tests.reader_tests.test_atms_l1b_nc module

The atms_l1b_nc reader tests package.

class satpy.tests.reader_tests.test_atms_l1b_nc.**TestAtsmsL1bNCFileHandler**

Bases: object

Test the AtmsL1bNCFileHandler reader.

test_antenna_temperature(*reader*, *atms_fake_dataset*)

Test antenna temperature.

test_attrs(*reader*, *param*, *expect*)

Test attributes.

test_drop_coords(*reader*)

Test drop coordinates.

test_end_time(*reader*)

Test end time.

test_get_dataset(*reader*)

Test get dataset.

test_merge_attributes(*reader*, *param*, *expect*)

Test merge attributes.

test_platform_name(*reader*)

Test platform name.

test_select_dataset(*reader*, *param*, *expect*)

Test select dataset.

test_sensor(*reader*)

Test sensor.

test_standardize_dims(*reader*, *dims*)

Test standardize dims.

test_start_time(*reader*)

Test start time.

satpy.tests.reader_tests.test_atms_l1b_nc.**atms_fake_dataset**()

Return fake ATMS dataset.

satpy.tests.reader_tests.test_atms_l1b_nc.**l1b_file**(*tmp_path*, *atms_fake_dataset*)

Return file path to level1b file.

satpy.tests.reader_tests.test_atms_l1b_nc.**reader**(*l1b_file*)

Return reader of ATMS level1b data.

satpy.tests.reader_tests.test_atms_sdr_hdf5 module

Module for testing the ATMS SDR HDF5 reader.

class satpy.tests.reader_tests.test_atms_sdr_hdf5.**FakeHDF5_ATMS_SDR_FileHandler**(*filename*, *filename_info*, *filetype_info*, *include_factors=True*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Create fake file handler.

static _add_basic_metadata_to_file_content(*file_content*, *filename_info*, *num_grans*)

_add_data_info_to_file_content(*file_content*, *filename*, *data_var_prefix*, *num_grans*)

static _add_geo_ref(*file_content*, *filename*)

static _add_geolocation_info_to_file_content(*file_content*, *filename*, *data_var_prefix*, *num_grans*)

_add_granule_specific_info_to_file_content(*file_content*, *dataset_group*, *num_granules*, *num_scans_per_granule*, *gran_group_prefix*)

static _convert_numpy_content_to_dataarray(*final_content*)

static _get_per_granule_lats()

static _get_per_granule_lons()

_num_of_bands = 22

_num_scans_per_gran = [12]

_num_test_granules = 1

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_atms_sdr_hdf5.**TestATMS_SDR_Reader**

Bases: object

Test ATMS SDR Reader.

_assert_bt_properties(*data_arr*, *num_scans=1*, *with_area=True*)

setup_method()

Wrap HDF5 file handler with our own fake handler.

teardown_method()

Stop wrapping the HDF5 file handler.

test_init()

Test basic init with no extra parameters.

test_init_start_end_time()

Test basic init with start and end times around the start/end times of the provided file.

test_load_all_bands(*files*, *expected*)

Load brightness temperatures for all 22 ATMS channels, with/without geolocation.

yaml_file = 'atms_sdr_hdf5.yaml'

satpy.tests.reader_tests.test_avhrr_l0_hrpt module

Tests for the hrpt reader.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**CalibratorPatcher**(*methodName='runTest'*)

Bases: *PygacPatcher*

Patch pygac.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp() → None

Patch pygac's calibration.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTChannel3**(*methodName='runTest'*)

Bases: *TestHRPTWithPatchedCalibratorAndFile*

Test case for reading calibrated brightness temperature from hrpt data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_channel_3a_counts()

Get the channel 4 bt.

_get_channel_3a_reflectance()

Get the channel 4 bt.

_get_channel_3b_bt()

Get the channel 4 bt.

test_channel_3a_masking()

Test that channel 3a is split correctly.

test_channel_3b_masking()

Test that channel 3b is split correctly.

test_uncalibrated_channel_3a_masking()

Test that channel 3a is split correctly.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTGetCalibratedBT**(*methodName='runTest'*)

Bases: *TestHRPTWithPatchedCalibratorAndFile*

Test case for reading calibrated brightness temperature from hrpt data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_channel_4_bt()

Get the channel 4 bt.

test_calibrated_bt_values()

Test the calibrated reflectance values.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTGetCalibratedReflectances**(*methodName='runTest'*)

Bases: *TestHRPTWithPatchedCalibratorAndFile*

Test case for reading calibrated reflectances from hrpt data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_channel_1_reflectance()

Get the channel 1 reflectance.

test_calibrated_reflectances_values()

Test the calibrated reflectance values.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTGetUncalibratedData**(*methodName='runTest'*)

Bases: *TestHRPTWithFile*

Test case for reading uncalibrated hrpt data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_channel_1_counts()

test_get_dataset_returns_a_dataarray()

Test that get_dataset returns a dataarray.

test_no_calibration_values_are_1()

Test that the values of non-calibrated data is 1.

test_platform_name()

Test that the platform name is correct.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTNavigation**(*methodName='runTest'*)

Bases: *TestHRPTWithFile*

Test case for computing HRPT navigation.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_prepare_mocks(*Orbital*, *SatelliteInterpolator*, *get_lonlatalt*)

Prepare the mocks.

setUp() → None

Set up the test case.

test_latitudes_are_returned(*Orbital*, *compute_pixels*, *get_lonlatalt*, *SatelliteInterpolator*)

Check that latitudes are returned properly.

test_longitudes_are_returned(*Orbital*, *compute_pixels*, *get_lonlatalt*, *SatelliteInterpolator*)

Check that latitudes are returned properly.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTReading**(*methodName='runTest'*)

Bases: *TestHRPTWithFile*

Test case for reading hrpt data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_reading()

Test that data is read.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTWithFile**(*methodName='runTest'*)

Bases: TestCase

Test base class with writing a fake file.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_dataset(*dataset_id*)

setUp() → None

Set up the test case.

tearDown() → None

Tear down the test case.

class satpy.tests.reader_tests.test_avhrr_l0_hrpt.**TestHRPTWithPatchedCalibratorAndFile**(*methodName='runTes*

Bases: *CalibratorPatcher*, *TestHRPTWithFile*

Test case with patched calibration routines and a synthetic file.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp() → None

Set up the test case.

tearDown()

Tear down the test case.

satpy.tests.reader_tests.test_avhrr_l0_hrpt.**fake_calibrate_solar**(*data*, **args*, ***kwargs*)

Fake calibration.

satpy.tests.reader_tests.test_avhrr_l0_hrpt.**fake_calibrate_thermal**(*data*, **args*, ***kwargs*)

Fake calibration.

satpy.tests.reader_tests.test_avhrr_l1b_gaclac module

Pygac interface.

class satpy.tests.reader_tests.test_avhrr_l1b_gaclac.**GACLACFilePatcher**(*methodName='runTest'*)

Bases: *PygacPatcher*

Patch pygac.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Patch GACLACFile.

class satpy.tests.reader_tests.test_avhrr_l1b_gaclac.**PygacPatcher**(*methodName='runTest'*)

Bases: TestCase

Patch pygac.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Patch pygac imports.

tearDown()

Unpatch the pygac imports.

class satpy.tests.reader_tests.test_avhrr_l1b_gaclac.**TestGACLACFile**(*methodName='runTest'*)

Bases: *GACLACFilePatcher*

Test the GACLAC file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_fh(*filename='NSS.GHRR.NG.D88002.S0614.E0807.B0670506.WI'*, ***kwargs*)

Create a file handler.

test__slice(*strip_invalid_lat*, *get_qual_flags*)

Test slicing.

test_get_angle()

Test getting the angle.

test_get_channel()

Test getting the channels.

test_get_dataset_angles(*get_angle*, **mocks*)

Test getting the angles.

test_get_dataset_latlon(**mocks*)

Test getting the latitudes and longitudes.

test_get_dataset_qual_flags(**mocks*)

Test getting the qualitiy flags.

test_get_dataset_slice(*get_channel*, *slc*, **mocks*)

Get a slice of a dataset.

test_init()

Test GACLACFile initialization.

test_read_raw_data()

Test raw data reading.

test_slice(*_slice*)

Test slicing.

test_strip_invalid_lat()

Test stripping invalid coordinates.

class satpy.tests.reader_tests.test_avhrr_l1b_gaclac.**TestGetDataset**(*methodName='runTest'*)

Bases: *GACLACFilePatcher*

Test the get_dataset method.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

static _check_get_channel_calls(*fh*, *get_channel*)

Check _get_channel() calls.

_classSetupFailed = False

_class_cleanups = []

static _create_expected(*name*)

static _create_file_handler(*reader*)

Mock reader and file handler.

static _get_dataset(*fh*)

setUp()

Set up the instance.

test_get_dataset_channels(*get_channel*, **mocks*)

Test getting the channel datasets.

test_get_dataset_no_tle(*get_channel*, **mocks*)

Test getting the channel datasets when no TLEs are present.

satpy.tests.reader_tests.test_avhrr_l1b_gaclac.**_get_fh_mocked**(*init_mock*, ***attrs*)

Create a mocked file handler with the given attributes.

satpy.tests.reader_tests.test_avhrr_l1b_gaclac.**_get_reader_mocked**(*along_track=3*)

Create a mocked reader.

satpy.tests.reader_tests.test_clavrx module

Module for testing the satpy.readers.clavrx module.

class satpy.tests.reader_tests.test_clavrx.**FakeHDF4FileHandlerGeo**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF4FileHandler*

Swap-in HDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_clavrx.**FakeHDF4FileHandlerPolar**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF4FileHandler*

Swap-in HDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_clavrx.**TestCLAVRXReaderGeo**(*methodName='runTest'*)

Bases: TestCase

Test CLAVR-X Reader with Geo files.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_load_all_new_donor()

Test loading all test datasets with new donor.

test_load_all_old_donor()

Test loading all test datasets with old donor.

test_no_nav_donor()

Test exception raised when no donor file is available.

yaml_file = 'clavrx.yaml'

class satpy.tests.reader_tests.test_clavrx.**TestCLAVRXReaderPolar**(*methodName='runTest'*)

Bases: TestCase

Test CLAVR-X Reader with Polar files.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_available_datasets()

Test available_datasets with fake variables from YAML.

test_init()

Test basic init with no extra parameters.

test_load_all()

Test loading all test datasets.

yaml_file = 'clavrx.yaml'

satpy.tests.reader_tests.test_clavrx_nc module

Module for testing the satpy.readers.clavrx module.

class satpy.tests.reader_tests.test_clavrx_nc.**TestCLAVRXReaderGeo**

Bases: object

Test CLAVR-X Reader with Geo files.

setup_method()

Read fake data.

test_available_datasets(*filenames*, *expected_datasets*)

Test that variables are dynamically discovered.

test_load_all_new_donor(*filenames*, *loadable_ids*)

Test loading all test datasets with new donor.

test_reader_creation(*filenames*, *expected_loadables*)

Test basic initialization.

yaml_file = 'clavrx.yaml'

satpy.tests.reader_tests.test_clavrx_nc.**fake_test_content**(*filename*, ***kwargs*)

Mimic reader input file content.

satpy.tests.reader_tests.test_cmsaf_claas module

Tests for the 'cmsaf-claas2_l2_nc' reader.

class satpy.tests.reader_tests.test_cmsaf_claas.**TestCLAAS2MultiFile**

Bases: object

Test reading multiple CLAAS-2 files.

multi_file_dataset(*multi_file_reader*)

Load datasets from multiple files.

multi_file_reader(*reader*, *fake_files*)

Create a multi-file reader.

test_combine_datasets(*multi_file_dataset*, *ds_name*, *expected*)

Test combination of datasets.

test_combine_timestamps(*multi_file_reader*, *start_time*)

Test combination of timestamps.

test_number_of_datasets(*multi_file_dataset*)

Test number of datasets.

class satpy.tests.reader_tests.test_cmsaf_claas.**TestCLAAS2SingleFile**

Bases: object

Test reading a single CLAAS2 file.

area_exp(*area_extent_exp*)

Get expected area definition.

area_extent_exp(*start_time*)

Get expected area extent.

file_handler(*fake_file*)

Return a CLAAS-2 file handler.

test_end_time(*file_handler*)

Test end time property.

test_get_area_def(*file_handler*, *area_exp*)

Test area definition.

test_get_dataset(*file_handler*, *ds_name*, *expected*)

Test dataset loading.

test_start_time(*file_handler*, *start_time*)

Test start time property.

satpy.tests.reader_tests.test_cmsaf_claas.**encoding**()

Dataset encoding.

satpy.tests.reader_tests.test_cmsaf_claas.**fake_dataset**(*start_time_str*)

Create a CLAAS-like test dataset.

satpy.tests.reader_tests.test_cmsaf_claas.**fake_file**(*fake_dataset*, *encoding*, *tmp_path*)

Write a fake dataset to file.

satpy.tests.reader_tests.test_cmsaf_claas.**fake_files**(*fake_dataset*, *encoding*, *tmp_path*)

Write the same fake dataset into two different files.

satpy.tests.reader_tests.test_cmsaf_claas.**reader**()

Return reader for CMSAF CLAAS-2.

satpy.tests.reader_tests.test_cmsaf_claas.**start_time**(*request*)

Get start time of the dataset.

satpy.tests.reader_tests.test_cmsaf_claas.**start_time_str**(*start_time*)

Get string representation of the start time.

satpy.tests.reader_tests.test_cmsaf_claas.**test_file_pattern**(*reader*)

Test file pattern matching.

satpy.tests.reader_tests.test_electrol_hrit module

The HRIT electrol reader tests package.

class satpy.tests.reader_tests.test_electrol_hrit.**TestHRITGOMSEpiFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRIT Epilogue FileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_init(*new_fh_init*, *fromfile*)

Set up the hrit file handler for testing.

class satpy.tests.reader_tests.test_electrol_hrit.**TestHRITGOMSFileHandler**(*methodName='runTest'*)

Bases: TestCase

A test of the ELECTRO-L main file handler functions.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_calibrate(**mocks*)

Test calibrate.

test_get_area_def(**mocks*)

Test get_area_def.

test_get_dataset(*calibrate_mock*, **mocks*)

Test get dataset.

class satpy.tests.reader_tests.test_electrol_hrit.**TestHRITGOMSProFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRIT Prologue FileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_calib = array([[50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50], ..., [50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50]], dtype=int32)

test_img_acq = {'Cel': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), 'StartDelay': array([9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019], dtype=int32), 'Status': array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=uint32), 'TagLength': array([24, 24, 24, 24, 24, 24, 24, 24, 24, 24], dtype=uint32), 'TagType': array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3], dtype=uint32)}

test_init(*new_fh_init*, *fromfile*)

Set up the hrit file handler for testing.

test_pro = {'ImageAcquisition': {'Cel': array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]), 'StartDelay': array([9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019, 9119019], dtype=int32), 'Status': array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2], dtype=uint32), 'TagLength': array([24, 24, 24, 24, 24, 24, 24, 24, 24, 24], dtype=uint32), 'TagType': array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3], dtype=uint32)}, 'ImageCalibration': array([[50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50], ..., [50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50], [50, 50, 50, ..., 50, 50, 50]], dtype=int32), 'SatelliteStatus': {'NominalLongitude': 1.3264, 'SatelliteCondition': 1, 'SatelliteID': 19002, 'SatelliteName': b'ELECTRO', 'TagLength': 292, 'TagType': 2, 'TimeOffset': 0.0}}

```
test_sat_status = {'NominalLongitude':  1.3264, 'SatelliteCondition':  1,
'SatelliteID': 19002, 'SatelliteName':  b'ELECTRO', 'TagLength':  292, 'TagType':
2, 'TimeOffset':  0.0}
```

class satpy.tests.reader_tests.test_electrol_hrit.**Testrecarray2dict**(*methodName='runTest'*)

Bases: TestCase

Test the function that converts numpy record arrays into dicts for use within SatPy.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fun()

Test record array.

satpy.tests.reader_tests.test_epic_l1b_h5 module

The epic_l1b_h5 reader tests package.

class satpy.tests.reader_tests.test_epic_l1b_h5.**TestEPICL1bReader**

Bases: object

Test the EPIC L1b HDF5 reader.

_setup_h5(*setup_hdf5_file*)

Initialise reader for the tests.

setup_method()

Set up the tests.

test_bad_calibration(*setup_hdf5_file*)

Test that error is raised if a bad calibration is used.

test_counts_calibration(*setup_hdf5_file*)

Test that data is correctly calibrated.

test_load_ancillary(*setup_hdf5_file*)

Test that ancillary datasets load correctly.

test_refl_calibration(*setup_hdf5_file*)

Test that data is correctly calibrated into reflectances.

test_times(*setup_hdf5_file*)

Test start and end times load properly.

satpy.tests.reader_tests.test_epic_l1b_h5.**make_fake_hdf_epic**(*fname*)

Make a fake HDF5 file for EPIC data testing.

satpy.tests.reader_tests.test_epic_l1b_h5.**setup_hdf5_file**(*tmp_path*)

Create temp hdf5 files.

satpy.tests.reader_tests.test_eps_l1b module

Test the eps l1b format.

class satpy.tests.reader_tests.test_eps_l1b.**BaseTestCaseEPSL1B**(*methodName='runTest'*)

Bases: TestCase

Base class for EPS l1b test case.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_create_structure()

class satpy.tests.reader_tests.test_eps_l1b.**TestEPSL1B**(*methodName='runTest'*)

Bases: *BaseTestCaseEPSL1B*

Test the filehandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the tests.

test_angles()

Test the navigation.

test_clould_flags()

Test getting the cloud flags.

test_dataset()

Test getting a dataset.

test_get_full_angles_twice(*mock__getitem__*)

Test get full angles twice.

test_navigation()

Test the navigation.

test_read_all()

Test initialization.

class satpy.tests.reader_tests.test_eps_l1b.**TestWrongSamplingEPSL1B**(*methodName='runTest'*)

Bases: *BaseTestCaseEPSL1B*

Test the filehandler on a corrupt file.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_inject_fixtures(*caplog*)

Inject caplog.

setUp()

Set up the tests.

test_get_dataset_fails_because_of_wrong_sample_rate()

Test that lons fail to be interpolate.

class satpy.tests.reader_tests.test_eps_l1b.**TestWrongScanlinesEPSL1B**(*methodName='runTest'*)

Bases: *BaseTestCaseEPSL1B*

Test the filehandler on a corrupt file.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_inject_fixtures(*caplog*)

Inject caplog.

setUp()

Set up the tests.

tearDown()

Tear down the tests.

test_get_dataset_longitude_shape_is_right()

Test that the shape of longitude is 1080.

test_read_all_assigns_int_scan_lines()

Test scanline assignment.

test_read_all_return_right_number_of_scan_lines()

Test scanline assignment.

test_read_all_warns_about_scan_lines()

Test scanline assignment.

satpy.tests.reader_tests.test_eps_l1b.**create_sections**(*structure*)

Create file sections.

satpy.tests.reader_tests.test_eum_base module

EUMETSAT base reader tests package.

class satpy.tests.reader_tests.test_eum_base.**TestGetServiceMode**(*methodName='runTest'*)

Bases: TestCase

Test the get_service_mode function.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_get_fci_service_mode_fdss()

Test fetching of FCI service mode information for FDSS.

test_get_fci_service_mode_rss()

Test fetching of FCI service mode information for RSS.

test_get_seviri_service_mode_fes()

Test fetching of SEVIRI service mode information for FES.

test_get_seviri_service_mode_iodc_E0415()

Test fetching of SEVIRI service mode information for IODC at 41.5 degrees East.

test_get_seviri_service_mode_iodc_E0455()

Test fetching of SEVIRI service mode information for IODC at 45.5 degrees East.

test_get_seviri_service_mode_rss()

Test fetching of SEVIRI service mode information for RSS.

test_get_unknown_instrument_service_mode()

Test fetching of service mode information for unknown input instrument.

test_get_unknown_lon_service_mode()

Test fetching of service mode information for unknown input longitude.

class satpy.tests.reader_tests.test_eum_base.**TestMakeTimeCdsDictionary**(*methodName='runTest'*)

Bases: TestCase

Test TestMakeTimeCdsDictionary.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fun()

Test function for TestMakeTimeCdsDictionary.

class satpy.tests.reader_tests.test_eum_base.**TestMakeTimeCdsRecarray**(*methodName='runTest'*)

Bases: TestCase

Test TestMakeTimeCdsRecarray.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fun()

Test function for TestMakeTimeCdsRecarray.

class satpy.tests.reader_tests.test_eum_base.**TestRecarray2Dict**(*methodName='runTest'*)

Bases: TestCase

Test TestRecarray2Dict.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_mpef_product_header()

Test function for TestRecarray2Dict and mpef product header.

test_timestamps()

Test function for TestRecarray2Dict.

satpy.tests.reader_tests.test_fci_l1c_nc module

Tests for the 'fci_l1c_nc' reader.

class satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerBase**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Class for faking the NetCDF4 Filehandler.

Get fake file content from 'get_test_content'.

_get_test_content_all_channels()

cached_file_content: Dict[str, DataArray] = {}

chan_patterns: Dict[str, Dict[str, List[int] | str]] = {}

get_test_content(*filename*, *filename_info*, *filetype_info*)

Get the content of the test data.

class satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerFDHSI**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeFCIFileHandlerBase*

Mock FDHSI data.

Get fake file content from 'get_test_content'.

chan_patterns: Dict[str, Dict[str, List[int] | str]] = {'ir_{:>02d}': {'channels': [38, 87, 97, 105, 123, 133], 'grid_type': '2km'}, 'nir_{:>02d}': {'channels': [13, 16, 22], 'grid_type': '1km'}, 'vis_{:>02d}': {'channels': [4, 5, 6, 8, 9], 'grid_type': '1km'}, 'wv_{:>02d}': {'channels': [63, 73], 'grid_type': '2km'}}

satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerFDHSI_fixture**()

Get a fixture for the fake FDHSI filehandler, including channel and file names.

class satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerHRFI**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeFCIFileHandlerBase*

Mock HRFI data.

Get fake file content from 'get_test_content'.

chan_patterns: Dict[str, Dict[str, List[int] | str]] = {'ir_{:>02d}_hr': {'channels': [38, 105], 'grid_type': '1km'}, 'nir_{:>02d}_hr': {'channels': [22], 'grid_type': '500m'}, 'vis_{:>02d}_hr': {'channels': [6], 'grid_type': '500m'}}

satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerHRFI_fixture**()

Get a fixture for the fake HRFI filehandler, including channel and file names.

class satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerWithBadData**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeFCIFileHandlerFDHSI*

Mock bad data.

Get fake file content from 'get_test_content'.

_get_test_content_all_channels()

class satpy.tests.reader_tests.test_fci_l1c_nc.**FakeFCIFileHandlerWithBadIDPFData**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeFCIFileHandlerFDHSI*

Mock bad data for IDPF TO-DO's.

Get fake file content from 'get_test_content'.

_get_test_content_all_channels()

class satpy.tests.reader_tests.test_fci_l1c_nc.**FakeH5Variable**(*data*, *dims=()*, *attrs=None*)

Bases: object

Class for faking h5netcdf.Variable class.

Initialize the class.

_set_meta()

property ndim

Get the number of dimensions.

property shape

Get the shape.

class satpy.tests.reader_tests.test_fci_l1c_nc.**TestFCIL1cNCReader**

Bases: object

Test FCI L1c NetCDF reader with nominal data.

expected_pos_info_for_filetype = {'fdhsi': {'1km': {'end_position_row': 200, 'grid_width': 11136, 'segment_height': 200, 'start_position_row': 1}, '2km': {'end_position_row': 100, 'grid_width': 5568, 'segment_height': 100, 'start_position_row': 1}}, 'hrfi': {'1km': {'end_position_row': 200, 'grid_width': 11136, 'segment_height': 200, 'start_position_row': 1}, '500m': {'end_position_row': 400, 'grid_width': 22272, 'segment_height': 400, 'start_position_row': 1}}}

fh_param_for_filetype = {'fdhsi': {'channels': {'solar': ['vis_04', 'vis_05', 'vis_06', 'vis_08', 'vis_09', 'nir_13', 'nir_16', 'nir_22'], 'solar_grid_type': ['1km', '1km', '1km', '1km', '1km', '1km', '1km', '1km'], 'terran': ['ir_38', 'wv_63', 'wv_73', 'ir_87', 'ir_97', 'ir_105', 'ir_123', 'ir_133'], 'terran_grid_type': ['2km', '2km', '2km', '2km', '2km', '2km', '2km', '2km']}, 'filenames': ['W_XX-EUMETSAT-Darmstadt,IMG+SAT, MTI1+FCI-1C-RRAD-FDHSI-FD--CHK-BODY--L2P-NC4E_C_EUMT_20170410114434_GTT_DEV_20170410113925_20170410 nc']}, 'hrfi': {'channels': {'solar': ['vis_06', 'nir_22'], 'solar_grid_type': ['500m', '500m'], 'terran': ['ir_38', 'ir_105'], 'terran_grid_type': ['1km', '1km']}, 'filenames': ['W_XX-EUMETSAT-Darmstadt,IMG+SAT, MTI1+FCI-1C-RRAD-HRFI-FD--CHK-BODY--L2P-NC4E_C_EUMT_20170410114434_GTT_DEV_20170410113925_20170410 nc']}}

test_area_definition_computation(*reader_configs*, *fh_param*, *expected_area*)

Test that the geolocation computation is correct.

test_excs(*reader_configs*, *fh_param*)

Test that exceptions are raised where expected.

test_file_pattern(*reader_configs*, *filenames*)

Test file pattern matching.

test_file_pattern_for_TRAIL_file(*reader_configs*, *filenames*)

Test file pattern matching for TRAIL files, which should not be picked up.

test_get_segment_position_info(*reader_configs*, *fh_param*, *expected_pos_info*)

Test the segment position info method.

test_load_aux_data(*reader_configs*, *fh_param*)

Test loading of auxiliary data.

test_load_bt(*reader_configs*, *caplog*, *fh_param*, *expected_res_n*)

Test loading with bt.

test_load_composite()

Test that composites are loadable.

test_load_counts(*reader_configs*, *fh_param*, *expected_res_n*)

Test loading with counts.

test_load_index_map(*reader_configs*, *fh_param*, *expected_res_n*)

Test loading of index_map.

test_load_quality_only(*reader_configs*, *fh_param*, *expected_res_n*)

Test that loading quality only works.

test_load_radiance(*reader_configs*, *fh_param*, *expected_res_n*)

Test loading with radiance.

test_load_reflectance(*reader_configs*, *fh_param*, *expected_res_n*)

Test loading with reflectance.

test_orbital_parameters_attr(*reader_configs*, *fh_param*)

Test the orbital parameter attribute.

test_platform_name(*reader_configs*, *fh_param*)

Test that platform name is exposed.

Test that the FCI reader exposes the platform name. Corresponds to GH issue 1014.

class satpy.tests.reader_tests.test_fci_l1c_nc.**TestFCIL1cNCReaderBadData**

Bases: object

Test the FCI L1c NetCDF Reader for bad data input.

test_handling_bad_data_ir(*reader_configs*, *caplog*)

Test handling of bad IR data.

test_handling_bad_data_vis(*reader_configs*, *caplog*)

Test handling of bad VIS data.

class satpy.tests.reader_tests.test_fci_l1c_nc.**TestFCIL1cNCReaderBadDataFromIDPF**

Bases: object

Test the FCI L1c NetCDF Reader for bad data input, specifically the IDPF issues.

test_bad_xy_coords(*reader_configs*)

Test that the geolocation computation is correct.

test_handling_bad_earthsun_distance(*reader_configs*)

Test handling of bad earth-sun distance data.

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_global_attributes**()

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_reader_with_filehandlers**(*filenames*, *reader_configs*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_calib_data_for_channel**(*data*, *ch_str*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_calib_for_channel_ir**(*data*, *meas_path*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_calib_for_channel_vis**(*data*, *meas*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_content_areadef**()

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_content_aux_data**()

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_content_for_channel**(*ch_str*, *grid_type*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_geolocation_for_channel**(*data*, *ch_str*, *grid_type*, *n_rows_cols*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_image_data_for_channel**(*data*, *ch_str*, *n_rows_cols*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_index_map_for_channel**(*data*, *ch_str*, *n_rows_cols*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_pixel_quality_for_channel**(*data*, *ch_str*, *n_rows_cols*)

satpy.tests.reader_tests.test_fci_l1c_nc.**_get_test_segment_position_for_channel**(*data*, *ch_str*, *n_rows_cols*)

satpy.tests.reader_tests.test_fci_l1c_nc.**clear_cache**(*reader*)

Clear the cache for file handlres in reader.

satpy.tests.reader_tests.test_fci_l1c_nc.**mocked_basefilehandler**(*filehandler*)

Mock patch the base class of the FCIL1cNCFileHandler with the content of our fake files (filehandler).

satpy.tests.reader_tests.test_fci_l1c_nc.**reader_configs**()

Return reader configs for FCI.

satpy.tests.reader_tests.test_fci_l2_nc module

The fci_cld_l2_nc reader tests package.

class satpy.tests.reader_tests.test_fci_l2_nc.**TestFciL2NCFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the FciL2NCFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test by creating a test file and opening it with the reader.

tearDown()

Remove the previously created test file.

test_all_basic()

Test all basic functionalities.

test_area_definition(*me_*, *gad_*)

Test the area definition computation.

test_dataset()

Test the correct execution of the get_dataset function with a valid file_key.

test_dataset_with_invalid_filekey()

Test the correct execution of the get_dataset function with an invalid file_key.

test_dataset_with_layer()

Check the correct execution of the get_dataset function with a valid file_key & layer.

test_dataset_with_scalar()

Test the execution of the get_dataset function for scalar values.

test_dataset_with_total_cot()

Test the correct execution of the get_dataset function for total COT (add contributions from two layers).

class satpy.tests.reader_tests.test_fci_l2_nc.**TestFciL2NCReadingByteData**(*methodName='runTest'*)

Bases: TestCase

Test the FciL2NCFileHandler when reading and extracting byte data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test by creating a test file and opening it with the reader.

tearDown()

Remove the previously created test file.

test_byte_extraction()

Test the execution of the get_dataset function.

class satpy.tests.reader_tests.test_fci_l2_nc.**TestFciL2NCSegmentFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the FciL2NCSegmentFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

static _get_unique_array(*iarr*, *jarr*)

setUp()

Set up the test by creating a test file and opening it with the reader.

tearDown()

Remove the previously created test file.

test_all_basic()

Test all basic functionalities.

test_dataset()

Test the correct execution of the get_dataset function with valid file_key.

test_dataset_slicing_catid()

Test the correct execution of the _slice_dataset function with 'category_id' set.

test_dataset_slicing_chid_catid()

Test the correct execution of the _slice_dataset function with 'channel_id' and 'category_id' set.

test_dataset_slicing_irid()

Test the correct execution of the _slice_dataset function with 'ir_channel_id' set.

test_dataset_slicing_visid_catid()

Test the correct execution of the _slice_dataset function with 'vis_channel_id' and 'category_id' set.

test_dataset_with_adef()

Test the correct execution of the get_dataset function with *with_area_definition=True*.

test_dataset_with_adef_and_wrongs_dims()

Test the correct execution of the get_dataset function with dims that don't match expected AreaDefinition.

test_dataset_with_invalid_filekey()

Test the correct execution of the get_dataset function with an invalid file_key.

test_dataset_with_scalar()

Test the execution of the get_dataset function for scalar values.

satpy.tests.reader_tests.test_fy4_base module

The fy4_base reader tests package.

class satpy.tests.reader_tests.test_fy4_base.**Test_FY4Base**

Bases: object

Tests for the FengYun4 base class for the components missed by AGRI/GHI tests.

setup_method()

Initialise the tests.

teardown_method()

Stop wrapping the HDF5 file handler.

test_badcalibration()

Test case where we pass a bad calibration type, radiance is not supported.

test_badplatform()

Test case where we pass a bad calibration type, radiance is not supported.

test_badsensor()

Test case where we pass a bad sensor name, must be GHI or AGRI.

satpy.tests.reader_tests.test_generic_image module

Unittests for generic image reader.

class satpy.tests.reader_tests.test_generic_image.**TestGenericImage**(*methodName='runTest'*)

Bases: TestCase

Test generic image reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create temporary images to test on.

tearDown()

Remove the temporary directory created for a test.

test_GenericImageFileHandler()

Test direct use of the reader.

test_GenericImageFileHandler_datasetid()

Test direct use of the reader.

test_GenericImageFileHandler_masking_only_integer()

Test direct use of the reader.

test_GenericImageFileHandler_nodata()

Test nodata handling with direct use of the reader.

test_geotiff_scene()

Test reading TIFF images via satpy.Scene().

test_geotiff_scene_nan()

Test reading TIFF images originally containing NaN values via satpy.Scene().

test_png_scene()

Test reading PNG images via satpy.Scene().

satpy.tests.reader_tests.test_geocat module

Module for testing the satpy.readers.geocat module.

class satpy.tests.reader_tests.test_geocat.**FakeNetCDF4FileHandler2**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_geocat.**TestGEOCATReader**(*methodName='runTest'*)

Bases: TestCase

Test GEOCAT Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap NetCDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_init_with_kwargs()

Test basic init with extra parameters.

test_load_all_goes17_hdf4()

Test loading all test datasets from GOES-17 HDF4 file.

test_load_all_himawari8()

Test loading all test datasets from H8 NetCDF file.

test_load_all_old_goes()

Test loading all test datasets from old GOES files.

yaml_file = 'geocat.yaml'

satpy.tests.reader_tests.test_geos_area module

Geostationary project utility module tests package.

class satpy.tests.reader_tests.test_geos_area.**TestGEOSProjectionUtil**(*methodName='runTest'*)

Bases: TestCase

Tests for the area utilities.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

make_pdict_ext(*typ*, *scan*)

Create a dictionary and extents to use in testing.

test_geos_area()

Test area extent calculation with N->S scan then S->N scan.

test_get_area_definition()

Test the retrieval of the area definition.

test_get_geos_area_naming()

Test the geos area naming function.

test_get_resolution_and_unit_strings_in_km()

Test the resolution and unit strings function for a km resolution.

test_get_resolution_and_unit_strings_in_m()

Test the resolution and unit strings function for a m resolution.

test_get_xy_from_linecol()

Test the scan angle calculation.

test_sampling_to_lfac_cfac()

Test conversion from angular sampling to line/column offset.

satpy.tests.reader_tests.test_gerb_l2_hr_h5 module

Unit tests for GERB L2 HR HDF5 reader.

satpy.tests.reader_tests.test_gerb_l2_hr_h5.**gerb_l2_hr_h5_dummy_file**(*tmp_path_factory*)

Create a dummy HDF5 file for the GERB L2 HR product.

satpy.tests.reader_tests.test_gerb_l2_hr_h5.**make_h5_null_string**(*length*)

Make a HDF5 type for a NULL terminated string of fixed length.

satpy.tests.reader_tests.test_gerb_l2_hr_h5.**test_dataset_load**(*gerb_l2_hr_h5_dummy_file*, *name*)

Test loading the solar flux component.

satpy.tests.reader_tests.test_gerb_l2_hr_h5.**write_h5_null_string_att**(*loc_id*, *name*, *s*)

Write a NULL terminated string attribute at loc_id.

satpy.tests.reader_tests.test_ghi_l1 module

The agri_l1 reader tests package.

class satpy.tests.reader_tests.test_ghi_l1.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_create_channel_data(*chs*, *cwls*, *file_type*)

_create_coeff_array(*nb_channels*)

_get_250m_data(*file_type*)

_get_2km_data(*file_type*)

_get_500m_data(*file_type*)

_get_geo_data(*file_type*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

make_test_data(*cwl*, *ch*, *prefix*, *dims*, *file_type*)

Make test data.

class satpy.tests.reader_tests.test_ghi_l1.**Test_HDF_GHI_L1_cal**

Bases: object

Test VIRR L1B Reader.

static _assert_which_channels_are_loaded(*available_datasets*, *band_names*, *resolution_to_test*)

_check_calibration_and_units(*band_names*, *result*)

static _check_keys_for_dsq(*available_datasets*, *resolution_to_test*)

static _check_units(*band_name*, *result*)

_create_reader_for_resolutions(**resolutions*)

setup_method()

Wrap HDF5 file handler with our own fake handler.

teardown_method()

Stop wrapping the HDF5 file handler.

test_ghi_all_bands_have_right_units()

Test all bands have the right units.

test_ghi_channels_are_loaded_with_right_resolution()

Test all channels are loaded with the right resolution.

test_ghi_counts_calibration()

Test loading data at counts calibration.

test_ghi_for_one_resolution(*resolution_to_test*)

Test loading data when only one resolution is available.

test_ghi_geo()

Test loading data for angles.

test_ghi_orbital_parameters_are_correct()

Test orbital parameters are set correctly.

yaml_file = 'ghi_l1.yaml'

satpy.tests.reader_tests.test_ghi_l1.**_create_filenames_from_resolutions**(**resolutions*)

Create filenames from the given resolutions.

satpy.tests.reader_tests.test_ghrsst_l2 module

Module for testing the satpy.readers.ghrsst_l2 module.

class satpy.tests.reader_tests.test_ghrsst_l2.**TestGHRSSTL2Reader**

Bases: object

Test Sentinel-3 SST L2 reader.

_create_tarfile_with_testdata(*mypath*)

Create a 'fake' testdata set in a tar file.

setup_method(*tmp_path*)

Create a fake osisaf ghrsst dataset.

test_get_dataset(*tmp_path*)

Test retrieval of datasets.

test_get_sensor(*tmp_path*)

Test retrieval of the sensor name from the netCDF file.

test_get_start_and_end_times(*tmp_path*)

Test retrieval of the sensor name from the netCDF file.

test_instantiate_single_netcdf_file(*tmp_path*)

Test initialization of file handlers - given a single netCDF file.

test_instantiate_tarfile(*tmp_path*)

Test initialization of file handlers - given a tar file as in the case of the SAFE format.

satpy.tests.reader_tests.test_glm_l2 module

The glm_l2 reader tests package.

class satpy.tests.reader_tests.test_glm_l2.**TestGLML2FileHandler**(*methodName='runTest'*)

Bases: TestCase

Tests for the GLM L2 reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*xr_*)

Create a fake file handler to test.

test_basic_attributes()

Test getting basic file attributes.

test_get_dataset()

Test the get_dataset method.

test_get_dataset_dqf()

Test the get_dataset method with special DQF var.

class satpy.tests.reader_tests.test_glm_l2.**TestGLML2Reader**(*methodName='runTest'*)

Bases: TestCase

Test high-level reading functionality of GLM L2 reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*xr_*)

Create a fake reader to test.

test_available_datasets()

Test that resolution is added to YAML configured variables.

yaml_file = 'glm_l2.yaml'

satpy.tests.reader_tests.test_glm_l2.**setup_fake_dataset**()

Create a fake dataset to avoid opening a file.

satpy.tests.reader_tests.test_goes_imager_hrit module

The hrit msg reader tests package.

class satpy.tests.reader_tests.test_goes_imager_hrit.**TestGVARFloat**(*methodName='runTest'*)

Bases: TestCase

GVAR float tester.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fun()

Test function.

class satpy.tests.reader_tests.test_goes_imager_hrit.**TestHRITGOESFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRITFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*new_fh_init*)

Set up the hrit file handler for testing.

test_get_area_def()

Test getting the area definition.

test_get_dataset(*base_get_dataset*)

Test get_dataset.

test_init()

Test the init.

class satpy.tests.reader_tests.test_goes_imager_hrit.**TestHRITGOESPrologueFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRITFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_init(*new_fh_init*, *fromfile*, *recarray2dict*)

Setup the hrit file handler for testing.

class satpy.tests.reader_tests.test_goes_imager_hrit.**TestMakeSGSTime**(*methodName='runTest'*)

Bases: TestCase

SGS Time tester.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fun()

Encode the test time.

satpy.tests.reader_tests.test_goes_imager_nc_eum module

Tests for the goes imager nc reader (EUMETSAT variant).

class satpy.tests.reader_tests.test_goes_imager_nc_eum.**GOESNCEUMFileHandlerRadianceTest**(*methodName='runT*

Bases: TestCase

Tests for the radiances.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

longMessage = True

setUp(*xr_*)

Set up the tests.

test_calibrate()

Test whether the correct calibration methods are called.

test_get_dataset_radiance()

Test getting the radiances.

test_get_sector()

Test sector identification.

class satpy.tests.reader_tests.test_goes_imager_nc_eum.**GOESNCEUMFileHandlerReflectanceTest**(*methodName='r*

Bases: TestCase

Testing the reflectances.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

longMessage = True

setUp(*xr_*)

Set up the tests.

test_get_dataset_reflectance()

Test getting the reflectance.

satpy.tests.reader_tests.test_goes_imager_nc_noaa module

Tests for the goes imager nc reader (NOAA CLASS variant).

class satpy.tests.reader_tests.test_goes_imager_nc_noaa.**GOESNCBaseFileHandlerTest**(*methodName='runTest'*)

Bases: TestCase

Testing the file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

longMessage = True

setUp(*xr_*)

Set up the tests.

test_calibrate_ir()

Test IR calibration.

test_calibrate_vis()

Test VIS calibration.

test_end_time()

Test dataset end time stamp.

test_get_nadir_pixel()

Test identification of the nadir pixel.

test_init()

Tests reader initialization.

test_ircounts2radiance()

Test conversion from IR counts to radiance.

test_start_time()

Test dataset start time stamp.

test_viscounts2radiance()

Test conversion from VIS counts to radiance.

class satpy.tests.reader_tests.test_goes_imager_nc_noaa.**GOESNCFileHandlerTest**(*methodName='runTest'*)

Bases: TestCase

Test the file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

longMessage = True

setUp(*xr_*)

Set up the tests.

test_calibrate()

Test whether the correct calibration methods are called.

test_get_dataset_coords()

Test whether coordinates returned by get_dataset() are correct.

test_get_dataset_counts()

Test whether counts returned by get_dataset() are correct.

test_get_dataset_invalid()

Test handling of invalid calibrations.

test_get_dataset_masks()

Test whether data and coordinates are masked consistently.

test_get_sector()

Test sector identification.

class satpy.tests.reader_tests.test_goes_imager_nc_noaa.**TestChannelIdentification**

Bases: object

Test identification of channel type.

test_invalid_channel()

Test handling of invalid channel type.

test_is_vis_channel(*channel_name*, *expected*)

Test vis channel identification.

class satpy.tests.reader_tests.test_goes_imager_nc_noaa.**TestMetadata**

Bases: object

Testcase for dataset metadata.

_apply_yaw_flip(*data_array*, *yaw_flip*)

_assert_earth_mask_equal(*metadata*, *expected*)

channel_id(*request*)

Set channel ID.

dataset(*lons_lats*, *channel_id*)

Create a fake dataset.

earth_mask(*yaw_flip*)

Get expected earth mask.

expected(*geometry*, *earth_mask*, *yaw_flip*)

Define expected metadata.

geometry(*channel_id*, *yaw_flip*)

Get expected geometry.

lons_lats(*yaw_flip*)

Get longitudes and latitudes.

mocked_file_handler(*dataset*)

Mock file handler to load the given fake dataset.

test_metadata(*mocked_file_handler*, *expected*)

Test dataset metadata.

yaw_flip(*request*)

Set yaw-flip flag.

satpy.tests.reader_tests.test_gpm_imerg module

Unittests for GPM IMERG reader.

class satpy.tests.reader_tests.test_gpm_imerg.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_get_geo_data(*num_rows*, *num_cols*)

_get_precip_data(*num_rows*, *num_cols*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_gpm_imerg.**TestHdf5IMERG**(*methodName='runTest'*)

Bases: TestCase

Test the GPM IMERG reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_load_data()

Test loading data.

yaml_file = 'gpm_imerg.yaml'

satpy.tests.reader_tests.test_grib module

Module for testing the satpy.readers.grib module.

class satpy.tests.reader_tests.test_grib.**FakeGRIB**(*messages=None*, *proj_params=None*, *latlons=None*)

Bases: object

Fake GRIB file returned by pygrib.open.

Init the grib file.

message(*msg_num*)

Get a message.

seek(*loc*)

Seek.

class satpy.tests.reader_tests.test_grib.**FakeMessage**(*values*, *proj_params=None*, *latlons=None*, ***attrs*)

Bases: object

Fake message returned by pygrib.open().message(x).

Init the message.

keys()

Get message keys.

latlons()

Get coordinates.

valid_key(*key*)

Validate key.

class satpy.tests.reader_tests.test_grib.**TestGRIBReader**

Bases: object

Test GRIB Reader.

static _get_fake_pygrib(*proj_params*, *lon_corners*, *lat_corners*)

_get_test_datasets(*dataids*, *fake_pygrib=None*)

setup_method()

Wrap pygrib to read fake data.

teardown_method()

Re-enable pygrib import.

test_area_def_crs(*proj_params*, *lon_corners*, *lat_corners*)

Check that the projection is accurate.

test_file_pattern()

Test matching of file patterns.

test_init()

Test basic init with no extra parameters.

test_jscanspositively(*proj_params*, *lon_corners*, *lat_corners*)
Check that data is flipped if the jScansPositively is present.

test_load_all(*proj_params*, *lon_corners*, *lat_corners*)
Test loading all test datasets.

test_missing_attributes(*proj_params*, *lon_corners*, *lat_corners*)
Check that the grib reader handles missing attributes in the grib file.

yaml_file = 'grib.yaml'

satpy.tests.reader_tests.test_grib.**_round_trip_projection_lonlat_check**(*area*)
Check that X/Y coordinates can be transformed multiple times.

Many GRIB files include non-standard projects that work for the initial transformation of X/Y coordinates to longitude/latitude, but may fail in the reverse transformation. For example, an eqc projection that goes from 0 longitude to 360 longitude. The X/Y coordinates may accurately go from the original X/Y metered space to the correct longitude/latitude, but transforming those coordinates back to X/Y space will produce the wrong result.

satpy.tests.reader_tests.test_grib.**fake_gribdata**()
Return some faked data for use as grib values.

satpy.tests.reader_tests.test_hdf4_utils module

Module for testing the satpy.readers.hdf4_utils module.

class satpy.tests.reader_tests.test_hdf4_utils.**FakeHDF4FileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDF4FileHandler*

Swap-in NetCDF4 File Handler for reader tests to use.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)
Mimic reader input file content.

Parameters

- **filename** (*str*) – input filename
- **filename_info** (*dict*) – Dict of metadata pulled from filename
- **filetype_info** (*dict*) – Dict of metadata from the reader's yaml config for this file type

Returns: dict of file content with keys like:

- 'dataset'
- '/attr/global_attr'
- 'dataset/attr/global_attr'
- 'dataset/shape'

class satpy.tests.reader_tests.test_hdf4_utils.**TestHDF4FileHandler**(*methodName='runTest'*)

Bases: TestCase

Test HDF4 File Handler Utility class.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create a test HDF4 file.

tearDown()

Remove the previously created test file.

test_all_basic()

Test everything about the HDF4 class.

satpy.tests.reader_tests.test_hdf5_utils module

Module for testing the satpy.readers.hdf5_utils module.

class satpy.tests.reader_tests.test_hdf5_utils.**FakeHDF5FileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *HDF5FileHandler*

Swap HDF5 File Handler for reader tests to use.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

Parameters

- **filename** (*str*) – input filename
- **filename_info** (*dict*) – Dict of metadata pulled from filename
- **filetype_info** (*dict*) – Dict of metadata from the reader's yaml config for this file type

Returns: dict of file content with keys like:

- 'dataset'
- '/attr/global_attr'
- 'dataset/attr/global_attr'
- 'dataset/shape'

class satpy.tests.reader_tests.test_hdf5_utils.**TestHDF5FileHandler**(*methodName='runTest'*)

Bases: TestCase

Test HDF5 File Handler Utility class.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create a test HDF5 file.

tearDown()

Remove the previously created test file.

test_all_basic()

Test everything about the HDF5 class.

satpy.tests.reader_tests.test_hdfeos_base module

Tests for the HDF-EOS base functionality.

class satpy.tests.reader_tests.test_hdfeos_base.**TestReadMDA**(*methodName='runTest'*)

Bases: TestCase

Test reading metadata.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_read_mda()

Test reading basic metadata.

test_read_mda_geo_resolution()

Test reading geo resolution.

satpy.tests.reader_tests.test_hrit_base module

The HRIT base reader tests package.

class satpy.tests.reader_tests.test_hrit_base.**TestHRITDecompress**(*methodName='runTest'*)

Bases: TestCase

Test the on-the-fly decompression.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_decompress(*popen*)

Test decompression works.

test_xrit_cmd()

Test running the xrit decompress command.

test_xrit_outfile()

Test the right decompression filename is used.

class satpy.tests.reader_tests.test_hrit_base.**TestHRITFileHandler**

Bases: object

Test the HRITFileHandler.

setup_method(*method*)

Set up the hrit file handler for testing.

test_get_area_def()

Test getting an area definition.

test_get_area_extent()

Test getting the area extent.

test_get_xy_from_linecol()

Test get_xy_from_linecol.

test_read_band_FSFile(*stub_hrit_file*)

Test reading a single band from an FSFile.

test_read_band_bzipped2_filepath(*stub_bzipped_hrit_file*)

Test reading a single band from a bzipped file.

test_read_band_filepath(*stub_hrit_file*)

Test reading a single band from a filepath.

test_read_band_gzip_stream(*stub_gzipped_hrit_file*)

Test reading a single band from a gzip stream.

test_start_end_time()

Test reading and converting start/end time.

class satpy.tests.reader_tests.test_hrit_base.**TestHRITFileHandlerCompressed**

Bases: object

Test the HRITFileHandler with compressed segments.

test_read_band_filepath(*stub_compressed_hrit_file*)

Test reading a single band from a filepath.

`satpy.tests.reader_tests.test_hrit_base.`**`create_stub_hrit`**(*filename, open_fun=<built-in function open>, meta={'GP_SC_ID': 324, 'annotation_header': b'H-000-MSG4__-MSG4________-VIS006___-000001___-202208180730-C_', 'cds_p_field': 64, 'cfac': -13642337, 'coff': 1856, 'compression_flag_for_data': 0, 'data_field_length': 17223680, 'data_field_representation': 3, 'file_type': 0, 'image_segment_line_quality': array([(1, (0, 0), 1, 1, 0), (1,*

Create a stub hrit file.

satpy.tests.reader_tests.test_hrit_base.**fake_decompress**(*infile*, *outdir='.'*)

Fake decompression.

satpy.tests.reader_tests.test_hrit_base.**new_get_hd**(*instance*, *hdr_info*)

Generate some metadata.

satpy.tests.reader_tests.test_hrit_base.**new_get_hd_compressed**(*instance*, *hdr_info*)

Generate some metadata.

satpy.tests.reader_tests.test_hrit_base.**stub_bzipped_hrit_file**(*tmp_path*)

Create a stub bzipped hrit file.

satpy.tests.reader_tests.test_hrit_base.**stub_compressed_hrit_file**(*tmp_path*)

Create a stub compressed hrit file.

satpy.tests.reader_tests.test_hrit_base.**stub_gzipped_hrit_file**(*tmp_path*)

Create a stub gzipped hrit file.

satpy.tests.reader_tests.test_hrit_base.**stub_hrit_file**(*tmp_path*)

Create a stub hrit file.

satpy.tests.reader_tests.test_hsaf_grib module

Module for testing the satpy.readers.grib module.

class satpy.tests.reader_tests.test_hsaf_grib.**FakeGRIB**(*messages=None*, *proj_params=None*, *latlons=None*)

Bases: object

Fake GRIB file returned by pygrib.open.

Init the fake grib file.

message(*msg_num*)

Fake message.

seek(*loc*)

Fake seek.

class satpy.tests.reader_tests.test_hsaf_grib.**FakeMessage**(*values*, *proj_params=None*, *latlons=None*, ***attrs*)

Bases: object

Fake message returned by pygrib.open().message(x).

Init the fake message.

latlons()

Get the latlons.

valid_key(*key*)

Check if key is valid.

class satpy.tests.reader_tests.test_hsaf_grib.**TestHSAFFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test HSAF Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap pygrib to read fake data.

tearDown()

Re-enable pygrib import.

test_get_area_def(*pg*)

Test the area definition setup, checks the size and extent.

test_get_dataset(*pg*)

Test reading the actual datasets from a grib file.

test_init(*pg*)

Test the init function, ensure that the correct dates and metadata are returned.

satpy.tests.reader_tests.test_hsaf_h5 module

Tests for the H-SAF H5 reader.

satpy.tests.reader_tests.test_hsaf_h5.**_get_scene_with_loaded_sc_datasets**(*filename*)

Return a scene with SC and SC_pal loaded.

satpy.tests.reader_tests.test_hsaf_h5.**sc_h5_file**(*tmp_path_factory*)

Create a fake HSAF SC HDF5 file.

satpy.tests.reader_tests.test_hsaf_h5.**test_hsaf_sc_areadef**(*sc_h5_file*)

Test the H-SAF SC area definition.

satpy.tests.reader_tests.test_hsaf_h5.**test_hsaf_sc_colormap_dataset**(*sc_h5_file*)

Test the H-SAF SC_pal dataset.

satpy.tests.reader_tests.test_hsaf_h5.**test_hsaf_sc_dataset**(*sc_h5_file*)

Test the H-SAF SC dataset.

satpy.tests.reader_tests.test_hsaf_h5.**test_hsaf_sc_datetime**(*sc_h5_file*)

Test the H-SAF reference time.

satpy.tests.reader_tests.test_hy2_scat_l2b_h5 module

Module for testing the satpy.readers.hy2_scat_l2b_h5 module.

class satpy.tests.reader_tests.test_hy2_scat_l2b_h5.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_get_all_ambiguities_data(*num_rows*, *num_cols*, *num_amb*)

_get_geo_data(*num_rows*, *num_cols*)

_get_geo_data_nsoas(*num_rows*, *num_cols*)

_get_global_attrs(*num_rows*, *num_cols*)

_get_selection_data(*num_rows*, *num_cols*)

_get_wvc_row_time(*num_rows*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_hy2_scat_l2b_h5.**TestHY2SCATL2BH5Reader**(*methodName='runTest'*)

Bases: TestCase

Test HY2 Scatterometer L2B H5 Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_load_data_all_ambiguities()

Test loading data.

test_load_data_row_times()

Test loading data.

test_load_data_selection()

Test loading data.

test_load_geo()

Test loading data.

test_load_geo_nsoas()

Test loading data from nsoas file.

test_properties()

Test platform_name.

test_reading_attrs()

Test loading data.

test_reading_attrs_nsoas()

Test loading data.

yaml_file = 'hy2_scat_l2b_h5.yaml'

satpy.tests.reader_tests.test_iasi_l2 module

Unit tests for IASI L2 reader.

class satpy.tests.reader_tests.test_iasi_l2.**TestIasiL2**(*methodName='runTest'*)

Bases: TestCase

Test IASI L2 reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

check_emissivity(*emis*)

Test reading emissivity dataset.

Helper function.

check_pressure(*pres*, *attrs=None*)

Test reading pressure dataset.

Helper function.

check_sensing_times(*times*)

Test reading sensing times.

Helper function.

setUp()

Create temporary data to test on.

tearDown()

Remove the temporary directory created for a test.

test_form_datetimes()

Test _form_datetimes() function.

test_get_dataset()

Test get_dataset() for different datasets.

test_init()

Test reader initialization.

test_read_dataset()

Test read_dataset() function.

test_read_geo()

Test read_geo() function.

test_scene()

Test scene creation.

test_scene_load_available_datasets()

Test that all datasets are available.

test_scene_load_emissivity()

Test loading emissivity data.

test_scene_load_pressure()

Test loading pressure data.

test_scene_load_sensing_times()

Test loading sensing times.

test_time_properties()

Test time properties.

satpy.tests.reader_tests.test_iasi_l2.**fake_iasi_l2_cdr_nc_dataset()**

Create minimally fake IASI L2 CDR NC dataset.

satpy.tests.reader_tests.test_iasi_l2.**fake_iasi_l2_cdr_nc_file**(*fake_iasi_l2_cdr_nc_dataset*, *tmp_path*)

Write a NetCDF file with minimal fake IASI L2 CDR NC data.

satpy.tests.reader_tests.test_iasi_l2.**save_test_data**(*path*)

Save the test to the indicated directory.

satpy.tests.reader_tests.test_iasi_l2.**test_iasi_l2_cdr_nc**(*fake_iasi_l2_cdr_nc_file*)

Test the IASI L2 CDR NC reader.

satpy.tests.reader_tests.test_iasi_l2_so2_bufr module

Unittesting the SEVIRI L2 BUFR reader.

class satpy.tests.reader_tests.test_iasi_l2_so2_bufr.**TestIasiL2So2Bufr**(*methodName='runTest'*)

Bases: TestCase

Test IASI l2 SO2 loader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create temporary file to perform tests with.

tearDown()

Remove the temporary directory created for a test.

test_scene()

Test scene creation.

test_scene_dataset_values()

Test loading data.

test_scene_load_available_datasets()

Test that all datasets are available.

satpy.tests.reader_tests.test_iasi_l2_so2_bufr.**save_test_data**(*path*)

Save the test file to the indicated directory.

satpy.tests.reader_tests.test_ici_l1b_nc module

The ici_l1b_nc reader tests package.

This version tests the reader for ICI test data as per PFS V3A.

class satpy.tests.reader_tests.test_ici_l1b_nc.**IciL1bFakeFileWriter**(*file_path*)

Bases: object

Writer class of fake ici level1b data.

Init.

static _write_attributes(*dataset*)

Write attributes.

static _write_measurement_data_group(*dataset*)

Write the measurement data group.

static _write_navigation_data_group(*dataset*)

Write the navigation data group.

static _write_quality_group(*dataset*)

Write the quality group.

write()

Write fake data to file.

class satpy.tests.reader_tests.test_ici_l1b_nc.**TestIciL1bNCFileHandler**

Bases: object

Test the IciL1bNCFileHandler reader.

test_calibrate_bt(*reader*)

Test calibrate brightness temperature.

test_calibrate_calls_calibrate_bt(*mocked_calibrate_bt*, *reader*)

Test calibrate calls calibrate_bt.

test_calibrate_does_not_call_calibrate_bt_if_not_needed(*mocked_calibrate*, *reader*)

Test calibrate does not call calibrate_bt if not needed.

test_calibrate_raises_for_unknown_calibration_method(*reader*)

Test perform calibration raises for unknown calibration method.

test_drop_coords(*reader*)

Test drop coordinates.

test_end_time(*reader*)

Test end time.

test_filter_variable(*reader*, *dims*, *data_info*, *expect*)

Test filter variable.

test_get_dataset_does_not_calibrate_if_not_desired(*mocked_calibrate*, *reader*, *dataset_info*)

Test get dataset does not calibrate if not desired.

test_get_dataset_handles_calibration(*reader*, *dataset_info*)

Test get dataset handles calibration.

test_get_dataset_orthorectifies_if_orthorect_data_defined(*reader*)

Test get dataset orthorectifies if orthorect data is defined.

test_get_dataset_return_none_if_data_not_exist(*reader*)

Tes get dataset return none if data does not exist.

test_get_global_attributes(*reader*)

Test get global attributes.

test_get_quality_attributes(*reader*)

Test get quality attributes.

test_get_third_dimension_name(*reader*)

Test get third dimension name.

test_get_third_dimension_name_return_none_for_2d_data(*reader*)

Test get third dimension name return none for 2d data.

test_interpolate_calls_interpolate_geo(*mock*, *reader*)

Test interpolate calls interpolate_geo.

test_interpolate_calls_interpolate_viewing_angles(*mock*, *reader*)

Test interpolate calls interpolate viewing_angles.

test_interpolate_geo(*reader*)

Test interpolate geographic coordinates.

test_interpolate_returns_none_if_dataset_not_exist(*reader*)

Test interpolate returns none if dataset not exist.

test_interpolate_viewing_angle(*reader*)

Test interpolate viewing angle.

test_latitude(*reader*)

Test latitude.

test_longitude(*reader*)

Test longitude.

test_manage_attributes(*mock*, *reader*)
Test manage attributes.

test_orthorectify(*reader*)
Test orthorectify.

test_platform_name(*reader*)
Test platform name.

test_sensor(*reader*)
Test sensor.

test_solar_azimuth(*reader*)
Test solar azimuth.

test_solar_zenith(*reader*)
Test solar zenith.

test_ssp_lon(*reader*)
Test sub satellite path longitude.

test_standardize_dims(*reader*, *dims*)
Test standardize dims.

test_start_time(*reader*)
Test start time.

satpy.tests.reader_tests.test_ici_l1b_nc.**dataset_info**()
Return dataset info.

satpy.tests.reader_tests.test_ici_l1b_nc.**fake_file**(*tmp_path*)
Return file path to level1b file.

satpy.tests.reader_tests.test_ici_l1b_nc.**reader**(*fake_file*)
Return reader of ici level1b data.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5 module

Tests for the Insat3D reader.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**_create_channels**(*channels*, *h5f*, *resolution*)

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**_create_lonlats**(*h5f*, *resolution*)

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**insat_filehandler**(*insat_filename*)
Instantiate a Filehandler.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**insat_filename**(*tmp_path_factory*)
Create a fake insat 3d l1b file.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**mask_array**(*array*)
Mask an array with nan instead of 0.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_filehandler_has_start_and_end_time**(*insat_filehandler*)
Test that the filehandler handles start and end time.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_filehandler_returns_area**(*insat_filehandler*)

Test that filehandle returns an area.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_filehandler_returns_coords**(*insat_filehandler*)

Test that lon and lat can be loaded.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_filehandler_returns_data_array**(*insat_filehandler*, *calibration*, *expected_values*)

Test that the filehandler can get dataarrays.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_filehandler_returns_masked_data_in_space**(*insat_fileh*

Test that the filehandler masks space pixels.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_backend_has_1km_channels**(*insat_filename*)

Test the insat3d backend.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_datatree_has_global_attributes**(*insat_filenam*

Test that the backend supports global attributes in the datatree.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_has_calibrated_arrays**(*insat_filename*, *resolution*, *name*, *shape*, *expected_values*, *expected_name*, *expected_units*)

Check that calibration happens as expected.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_has_dask_arrays**(*insat_filename*)

Test that the backend uses dask.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_has_global_attributes**(*insat_filename*, *resolution*)

Test that the backend supports global attributes.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_has_orbital_parameters**(*insat_filehandler*)

Test that the filehandler returns data with orbital parameter attributes.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_only_has_3_resolutions**(*insat_filename*)

Test that we only accept 1000, 4000, 8000.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_opens_datatree**(*insat_filename*, *resolution*)

Test that a datatree is produced.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_insat3d_returns_lonlat**(*insat_filename*, *resolution*)

Test that lons and lats are loaded.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_satpy_load_array**(*insat_filename*)

Test that satpy can load the VIS array.

satpy.tests.reader_tests.test_insat3d_img_l1b_h5.**test_satpy_load_two_arrays**(*insat_filename*)

Test that satpy can load the VIS array.

satpy.tests.reader_tests.test_li_l2_nc module

Unit tests on the LI L2 reader using the conventional mock constructed context.

class satpy.tests.reader_tests.test_li_l2_nc.**TestLIL2**

Bases: object

Main test class for the LI L2 reader.

_test_dataset_sector_variables(*settings*, *ds_desc*, *handler*)

Check the loading of the in sector variables.

_test_dataset_single_sector_variable(*names*, *desc*, *settings*, *handler*)

Check the validity of a given sector variable.

_test_dataset_single_variable(*vname*, *desc*, *settings*, *handler*)

Check the validity of a given variable.

_test_dataset_variable(*var_params*, *sname=''*)

Test the validity of a given (sector) variable.

_test_dataset_variables(*settings*, *ds_desc*, *handler*)

Check the loading of the non in sector variables.

create_fullname_key(*desc*, *var_path*, *vname*, *sname=''*)

Create full name key for sector/non-sector content retrieval.

fake_handler()

Wrap NetCDF4 FileHandler with our own fake handler.

generate_coords(*filetype_infos*, *file_type_name*, *variable_name*)

Generate file handler and mimic coordinate generator call.

get_variable_dataset(*dataset_info*, *dname*, *handler*)

Get the dataset of a given (sector) variable.

handler_with_area(*filetype_infos*, *product_name*)

Create handler with area definition.

static param_provider(*_filename*, *filename_info*, *_fileype_info*)

Provide parameters.

test_apply_accumulate_index_offset(*filetype_infos*)

Should accumulate index offsets.

test_available_datasets(*filetype_infos*)

Test available_datasets from li reader.

test_combine_info(*filetype_infos*)

Test overridden combine_info.

test_coordinates_projection(*filetype_infos*)

Should automatically generate lat/lon coords from projection data.

test_coords_generation(*filetype_infos*)

Compare daskified coords generation results with non-daskified.

test_dataset_loading(*filetype_infos*)

Test loading of all datasets from all products.

test_dataset_not_in_provided_dataset(*filetype_infos*)

Test loading of a dataset that is not provided.

test_filename_infos(*filetype_infos*)

Test settings retrieved from filename.

test_generate_coords_called_once(*filetype_infos*)

Test that the method is called only once.

test_generate_coords_inverse_proj(*filetype_infos*)

Test inverse_projection execution delayed until .values is called on the dataset.

test_generate_coords_not_called_on_non_accum_dataset(*filetype_infos*)

Test that the method is not called when getting non-accum dataset.

test_generate_coords_not_called_on_non_coord_dataset(*filetype_infos*)

Test that the method is not called when getting non-coord dataset.

test_generate_coords_on_accumulated_prods(*filetype_infos*)

Test daskified generation of coords.

test_generate_coords_on_lon_lat(*filetype_infos*)

Test getting lon/lat dataset on accumulated product.

test_get_area_def_acc_products(*filetype_infos*)

Test retrieval of area def for accumulated products.

test_get_area_def_non_acc_products(*filetype_infos*)

Test retrieval of area def for non-accumulated products.

test_get_first_valid_variable(*filetype_infos*)

Test get_first_valid_variable from li reader.

test_get_first_valid_variable_not_found(*filetype_infos*)

Test get_first_valid_variable from li reader if the variable is not found.

test_get_on_fci_grid_exc(*filetype_infos*)

Test the execution of the get_on_fci_grid function for an accumulated gridded variable.

test_get_on_fci_grid_exc_non_accum(*filetype_infos*)

Test the non-execution of the get_on_fci_grid function for a non-accumulated variable.

test_get_on_fci_grid_exc_non_grid(*filetype_infos*)

Test the non-execution of the get_on_fci_grid function for an accumulated non-gridded variable.

test_milliseconds_to_timedelta(*filetype_infos*)

Should covert milliseconds to timedelta.

test_report_datetimes(*filetype_infos*)

Should report time variables as numpy datetime64 type and time durations as timedelta64.

test_swath_coordinates(*filetype_infos*)

Test that swath coordinates are used correctly to assign coordinates to some datasets.

test_unregistered_dataset_loading(*filetype_infos*)

Test loading of an unregistered dataset.

test_var_path_exists(*filetype_infos*)

Test variable_path_exists from li reader.

test_variable_scaling(*filetype_infos*)

Test automatic rescaling with offset and scale attributes.

test_with_area_def(*filetype_infos*)

Test accumulated products data array with area definition.

test_with_area_def_pixel_placement(*filetype_infos*)

Test the placements of pixel value with area definition.

test_with_area_def_vars_with_no_pattern(*filetype_infos*)

Test accumulated products variable with no patterns and with area definition.

test_without_area_def(*filetype_infos*)

Test accumulated products data array without area definition.

satpy.tests.reader_tests.test_li_l2_nc.**std_filetype_infos**()

Return standard filetype info for LI L2.

satpy.tests.reader_tests.test_meris_nc module

Module for testing the satpy.readers.meris_nc_sen3 module.

class satpy.tests.reader_tests.test_meris_nc.**TestBitFlags**(*methodName='runTest'*)

Bases: TestCase

Test the bitflag reading.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_bitflags()

Test the BitFlags class.

class satpy.tests.reader_tests.test_meris_nc.**TestMERISReader**(*methodName='runTest'*)

Bases: TestCase

Test various meris_nc_sen3 filehandlers.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_get_dataset(*mocked_dataset*)

Test reading datasets.

test_instantiate(*mocked_dataset*)

Test initialization of file handlers.

test_meris_angles(*mocked_dataset*)

Test reading datasets.

test_meris_meteo(*mocked_dataset*)

Test reading datasets.

test_open_file_objects(*mocked_open_dataset*)

Test initialization of file handlers.

satpy.tests.reader_tests.test_mersi_l1b module

Tests for the 'mersi2_l1b' reader.

class satpy.tests.reader_tests.test_mersi_l1b.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_add_band_data_file_content()

_add_geo_data_file_content()

_add_tbb_coefficients(*global_attrs*)

property _geo_prefix_for_file_type

_get_data_file_content()

property _num_cols_for_file_type

property _rows_per_scan

_set_sensor_attrs(*global_attrs*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

num_cols = 2048

num_scans = 2

class satpy.tests.reader_tests.test_mersi_l1b.**MERSIL1BTester**

Bases: object

Test MERSI2 L1B Reader.

setup_method()

Wrap HDF5 file handler with our own fake handler.

teardown_method()

Stop wrapping the HDF5 file handler.

class satpy.tests.reader_tests.test_mersi_l1b.**TestMERSI2L1B**

Bases: *MERSIL1BTester*

Test the FY3D MERSI2 L1B reader.

filenames_1000m = ['tf2019071182739.FY3D-X_MERSI_1000M_L1B.HDF', 'tf2019071182739.FY3D-X_MERSI_GEO1K_L1B.HDF']

filenames_250m = ['tf2019071182739.FY3D-X_MERSI_0250M_L1B.HDF', 'tf2019071182739.FY3D-X_MERSI_GEOQK_L1B.HDF']

filenames_all = ['tf2019071182739.FY3D-X_MERSI_1000M_L1B.HDF', 'tf2019071182739.FY3D-X_MERSI_GEO1K_L1B.HDF', 'tf2019071182739.FY3D-X_MERSI_0250M_L1B.HDF', 'tf2019071182739.FY3D-X_MERSI_GEOQK_L1B.HDF']

test_1km_resolutions()

Test loading data when only 1km resolutions are available.

test_250_resolutions()

Test loading data when only 250m resolutions are available.

test_all_resolutions()

Test loading data when all resolutions are available.

test_counts_calib()

Test loading data at counts calibration.

test_rad_calib()

Test loading data at radiance calibration.

yaml_file = 'mersi2_l1b.yaml'

class satpy.tests.reader_tests.test_mersi_l1b.**TestMERSILLL1B**

Bases: *MERSIL1BTester*

Test the FY3E MERSI-LL L1B reader.

filenames_1000m = ['FY3E_MERSI_GRAN_L1_20230410_1910_1000M_V0.HDF', 'FY3E_MERSI_GRAN_L1_20230410_1910_GEO1K_V0.HDF']

filenames_250m = ['FY3E_MERSI_GRAN_L1_20230410_1910_0250M_V0.HDF', 'FY3E_MERSI_GRAN_L1_20230410_1910_GEOQK_V0.HDF']

filenames_all = ['FY3E_MERSI_GRAN_L1_20230410_1910_1000M_V0.HDF', 'FY3E_MERSI_GRAN_L1_20230410_1910_GEO1K_V0.HDF', 'FY3E_MERSI_GRAN_L1_20230410_1910_0250M_V0.HDF', 'FY3E_MERSI_GRAN_L1_20230410_1910_GEOQK_V0.HDF']

test_1km_resolutions()

Test loading data when only 1km resolutions are available.

test_250_resolutions()

Test loading data when only 250m resolutions are available.

test_all_resolutions()

Test loading data when all resolutions are available.

test_rad_calib()

Test loading data at radiance calibration.

yaml_file = 'mersi_ll_l1b.yaml'

satpy.tests.reader_tests.test_mersi_l1b.**_get_1km_data**(*num_scans*, *rows_per_scan*, *num_cols*)

satpy.tests.reader_tests.test_mersi_l1b.**_get_250m_data**(*num_scans*, *rows_per_scan*, *num_cols*)

satpy.tests.reader_tests.test_mersi_l1b.**_get_250m_ll_data**(*num_scans*, *rows_per_scan*, *num_cols*)

satpy.tests.reader_tests.test_mersi_l1b.**_get_calibration**(*num_scans*)

satpy.tests.reader_tests.test_mersi_l1b.**_get_geo_data**(*num_scans*, *rows_per_scan*, *num_cols*, *prefix*)

satpy.tests.reader_tests.test_mersi_l1b.**_test_helper**(*res*)

Remove test code duplication.

satpy.tests.reader_tests.test_mersi_l1b.**make_test_data**(*dims*)

Make test data.

satpy.tests.reader_tests.test_mimic_TPW2_lowres module

Module for testing the satpy.readers.tropomi_l2 module.

class satpy.tests.reader_tests.test_mimic_TPW2_lowres.**FakeNetCDF4FileHandlerMimicLow**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=Fa*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content for lower resolution files.

class satpy.tests.reader_tests.test_mimic_TPW2_lowres.**TestMimicTPW2Reader**(*methodName='runTest'*)

Bases: TestCase

Test Mimic Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap NetCDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_init()

Test basic initialization of this reader.

test_load_mimic_float()

Load TPW mimic float data.

test_load_mimic_timedelta()

Load TPW mimic timedelta data (data latency variables).

test_load_mimic_ubyte()

Load TPW mimic sensor grids.

yaml_file = 'mimicTPW2_comp.yaml'

satpy.tests.reader_tests.test_mimic_TPW2_nc module

Module for testing the satpy.readers.tropomi_l2 module.

class satpy.tests.reader_tests.test_mimic_TPW2_nc.**FakeNetCDF4FileHandlerMimic**(*filename, filename_info, filetype_info, auto_maskandscale=False, xarray_kwargs=None, cache_var_size=0, cache_handle=False, extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename, filename_info, filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_mimic_TPW2_nc.**TestMimicTPW2Reader**(*methodName='runTest'*)

Bases: TestCase

Test Mimic Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()
Wrap NetCDF4 file handler with our own fake handler.

tearDown()
Stop wrapping the NetCDF4 file handler.

test_init()
Test basic initialization of this reader.

test_load_mimic()
Load Mimic data.

yaml_file = 'mimicTPW2_comp.yaml'

satpy.tests.reader_tests.test_mirs module

Module for testing the satpy.readers.tropomi_l2 module.

class satpy.tests.reader_tests.test_mirs.**TestMirsL2_NcReader**
Bases: object

Test mirs Reader.

static _check_area(*data_arr*)

static _check_attrs(*data_arr*, *platform_name*)

static _check_fill(*data_arr*)

static _check_fill_value(*data_arr*, *test_fill_value*)

static _check_valid_range(*data_arr*, *test_valid_range*)

setup_method()
Read fake data.

test_available_datasets(*filenames*, *expected_datasets*)
Test that variables are dynamically discovered.

test_basic_load(*filenames*, *loadable_ids*, *platform_name*, *reader_kw*)
Test that variables are loaded properly.

test_reader_creation(*filenames*, *expected_loadables*)
Test basic initialization.

yaml_file = 'mirs.yaml'

satpy.tests.reader_tests.test_mirs.**_get_datasets_with_attributes**(***kwargs*)
Represent files with two resolution of variables in them (ex. OCEAN).

satpy.tests.reader_tests.test_mirs.**_get_datasets_with_less_attributes()**
Represent files with two resolution of variables in them (ex. OCEAN).

satpy.tests.reader_tests.test_mirs.**fake_coeff_from_fn**(*fn*)
Create Fake Coefficients.

satpy.tests.reader_tests.test_mirs.**fake_open_dataset**(*filename*, ***kwargs*)
Create a Dataset similar to reading an actual file with xarray.open_dataset.

satpy.tests.reader_tests.test_msi_safe module

Module for testing the satpy.readers.msi_safe module.

class satpy.tests.reader_tests.test_msi_safe.**TestMTDXML**

Bases: object

Test the SAFE MTD XML file handler.

setup_method()

Set up the test case.

test_old_xml_calibration()

Test the calibration of older data formats (no offset).

test_satellite_zenith_array()

Test reading the satellite zenith array.

test_xml_calibration()

Test the calibration with radiometric offset.

test_xml_calibration_to_radiance()

Test the calibration with a different offset.

test_xml_calibration_unmasked_saturated()

Test the calibration with radiometric offset but unmasked saturated pixels.

test_xml_calibration_with_different_offset()

Test the calibration with a different offset.

test_xml_navigation()

Test the navigation.

class satpy.tests.reader_tests.test_msi_safe.**TestSAFEMSIL1C**

Bases: object

Test case for image reading (jp2k).

setup_method()

Set up the test.

test_calibration_and_masking(*mask_saturated*, *calibration*, *expected*)

Test that saturated is masked with inf when requested and that calibration is performed.

satpy.tests.reader_tests.test_msu_gsa_l1b module

Tests for the 'msu_gsa_l1b' reader.

class satpy.tests.reader_tests.test_msu_gsa_l1b.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_get_data(*num_scans*, *num_cols*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_msu_gsa_l1b.**TestMSUGSABReader**

Bases: object

Test MSU GS/A L1B Reader.

setup_method()

Wrap HDF5 file handler with our own fake handler.

teardown_method()

Stop wrapping the HDF5 file handler.

test_irbt()

Test retrieval in brightness temperature.

test_nocounts()

Test we can't get IR or VIS data as counts.

test_vis_cal()

Test that we can retrieve VIS data as both radiance and reflectance.

yaml_file = 'msu_gsa_l1b.yaml'

satpy.tests.reader_tests.test_mviri_l1b_fiduceo_nc module

Unit tests for the FIDUCEO MVIRI FCDR Reader.

class satpy.tests.reader_tests.test_mviri_l1b_fiduceo_nc.**TestDatasetWrapper**

Bases: object

Unit tests for DatasetWrapper class.

test_reassign_coords()

Test reassigning of coordinates.

For some reason xarray does not always assign (y, x) coordinates to the high resolution datasets, although they have dimensions (y, x) and coordinates y and x exist. A dataset with these properties seems impossible to create (neither dropping, resetting or deleting coordinates seems to work). Instead use mock as a workaround.

class satpy.tests.reader_tests.test_mviri_l1b_fiduceo_nc.**TestFiduceoMviriFileHandlers**

Bases: object

Unit tests for FIDUCEO MVIRI file handlers.

test_angle_cache(*interp_tiepoints*, *file_handler*)

Test caching of angle datasets.

test_bad_quality_warning(*file_handler*)

Test warning about bad VIS quality.

test_calib_exceptions(*file_handler*)

Test calibration exceptions.

test_file_pattern(*reader*)

Test file pattern matching.

test_get_area_definition(*file_handler*, *name*, *resolution*, *area_exp*)
Test getting area definitions.

test_get_dataset(*file_handler*, *name*, *calibration*, *resolution*, *expected*)
Test getting datasets.

test_get_dataset_corrupt(*file_handler*)
Test getting datasets with known corruptions.

test_init(*file_handler*)
Test file handler initialization.

test_time_cache(*interp_acq_time*, *file_handler*)
Test caching of acquisition times.

satpy.tests.reader_tests.test_mviri_l1b_fiduceo_nc.**fixture_fake_dataset**()
Create fake dataset.

satpy.tests.reader_tests.test_mviri_l1b_fiduceo_nc.**fixture_file_handler**(*fake_dataset*, *request*)
Create mocked file handler.

satpy.tests.reader_tests.test_mviri_l1b_fiduceo_nc.**fixture_reader**()
Return MVIRI FIDUCEO FCDR reader.

satpy.tests.reader_tests.test_mws_l1b_nc module

The mws_l1b_nc reader tests.

This module tests the reading of the MWS l1b netCDF format data as per version v4B issued 22 November 2021.

class satpy.tests.reader_tests.test_mws_l1b_nc.**MWSL1BFakeFileWriter**(*file_path*)
Bases: object

Writer class of fake mws level-1b data.

Init.

static _create_scan_dimensions(*dataset*)
Create the scan/fovs dimensions.

static _write_attributes(*dataset*)
Write attributes.

static _write_calibration_data_group(*dataset*)
Write the calibration data group.

static _write_measurement_data_group(*dataset*)
Write the measurement data group.

static _write_navigation_data_group(*dataset*)
Write the navigation data group.

static _write_quality_group(*dataset*)
Write the quality group.

static _write_status_group(*dataset*)
Write the status group.

write()

Write fake data to file.

class satpy.tests.reader_tests.test_mws_l1b_nc.**TestMwsL1bNCFileHandler**

Bases: object

Test the MWSL1BFile reader.

static test_drop_coords(*reader*)

Test drop coordinates.

test_end_time(*reader*)

Test acquiring the end time.

test_get_dataset_aux_data_expected_data_missing(*caplog*, *reader*)

Test get auxillary dataset which is not present but supposed to be in file.

test_get_dataset_aux_data_not_supported(*reader*)

Test get auxillary dataset not supported.

test_get_dataset_get_channeldata_bts(*reader*)

Test getting channel data.

test_get_dataset_get_channeldata_counts(*reader*)

Test getting channel data.

test_get_dataset_logs_debug_message(*caplog*, *fake_file*, *reader*)

Test get dataset return none if data does not exist.

test_get_dataset_return_none_if_data_not_exist(*reader*)

Test get dataset return none if data does not exist.

test_get_global_attributes(*reader*)

Test get global attributes.

test_get_navigation_longitudes(*caplog*, *fake_file*, *reader*)

Test get the longitudes.

test_manage_attributes(*mock*, *reader*)

Test manage attributes.

test_platform_name(*reader*)

Test getting the platform name.

test_sensor(*reader*)

Test sensor.

test_standardize_dims(*reader*, *dims*)

Test standardize dims.

test_start_time(*reader*)

Test acquiring the start time.

test_sub_satellite_latitude_end(*reader*)

Test getting the latitude of sub-satellite point at end of the product.

test_sub_satellite_latitude_start(*reader*)

Test getting the latitude of sub-satellite point at start of the product.

test_sub_satellite_longitude_end(*reader*)

Test getting the longitude of sub-satellite point at end of the product.

test_sub_satellite_longitude_start(*reader*)

Test getting the longitude of sub-satellite point at start of the product.

satpy.tests.reader_tests.test_mws_l1b_nc.**fake_file**(*tmp_path*)

Return file path to level-1b file.

satpy.tests.reader_tests.test_mws_l1b_nc.**reader**(*fake_file*)

Return reader of mws level-1b data.

satpy.tests.reader_tests.test_mws_l1b_nc.**test_get_channel_index_from_name**(*name*, *index*)

Test getting the MWS channel index from the channel name.

satpy.tests.reader_tests.test_mws_l1b_nc.**test_get_channel_index_from_name_throw_exception**()

Test that an excpetion is thrown when getting the MWS channel index from an unsupported name.

satpy.tests.reader_tests.test_netcdf_utils module

Module for testing the satpy.readers.netcdf_utils module.

class satpy.tests.reader_tests.test_netcdf_utils.**FakeNetCDF4FileHandler**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *NetCDF4FileHandler*

Swap-in NetCDF4 File Handler for reader tests to use.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

Parameters

- **filename** (*str*) – input filename
- **filename_info** (*dict*) – Dict of metadata pulled from filename
- **filetype_info** (*dict*) – Dict of metadata from the reader's yaml config for this file type

Returns: dict of file content with keys like:

- 'dataset'
- '/attr/global_attr'
- 'dataset/attr/global_attr'
- 'dataset/shape'
- 'dataset/dimensions'
- '/dimension/my_dim'

class satpy.tests.reader_tests.test_netcdf_utils.**TestNetCDF4FileHandler**(*methodName='runTest'*)

Bases: TestCase

Test NetCDF4 File Handler Utility class.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create a test NetCDF4 file.

tearDown()

Remove the previously created test file.

test_all_basic()

Test everything about the NetCDF4 class.

test_caching()

Test that caching works as intended.

test_filenotfound()

Test that error is raised when file not found.

test_get_and_cache_npxr_data_is_cached()

Test that the data are cached when get_and_cache_npxr() is called.

test_get_and_cache_npxr_is_xr()

Test that get_and_cache_npxr() returns xr.DataArray.

test_listed_variables()

Test that only listed variables/attributes area collected.

test_listed_variables_with_composing()

Test that composing for listed variables is performed.

class satpy.tests.reader_tests.test_netcdf_utils.**TestNetCDF4FsspecFileHandler**

Bases: object

Test the remote reading class.

test_default_to_netcdf4_lib()

Test that the NetCDF4 backend is used by default.

test_use_h5netcdf_for_file_not_accessible_locally()

Test that h5netcdf is used for files that are not accesible locally.

satpy.tests.reader_tests.test_nucaps module

Module for testing the satpy.readers.nucaps module.

class satpy.tests.reader_tests.test_nucaps.**FakeNetCDF4FileHandler2**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_nucaps.**TestNUCAPSReader**(*methodName='runTest'*)

Bases: TestCase

Test NUCAPS Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap NetCDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_init_with_kwargs()

Test basic init with extra parameters.

test_load_individual_pressure_levels_min_max()

Test loading individual Temperature with min/max level specified.

test_load_individual_pressure_levels_single()

Test loading individual Temperature with specific levels.

test_load_individual_pressure_levels_true()

Test loading Temperature with individual pressure datasets.

test_load_multiple_files_pressure()

Test loading Temperature from multiple input files.

test_load_nonpressure_based()

Test loading all channels that aren't based on pressure.

test_load_pressure_based()

Test loading all channels based on pressure.

test_load_pressure_levels_min_max()

Test loading Temperature with min/max level specified.

test_load_pressure_levels_single()

Test loading a specific Temperature level.

test_load_pressure_levels_single_and_pressure_levels()

Test loading a specific Temperature level and pressure levels.

test_load_pressure_levels_true()

Test loading Temperature with all pressure levels.

yaml_file = 'nucaps.yaml'

class satpy.tests.reader_tests.test_nucaps.**TestNUCAPSScienceEDRReader**(*methodName='runTest'*)

Bases: TestCase

Test NUCAPS Science EDR Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap NetCDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_load_individual_pressure_levels_min_max()

Test loading individual Temperature with min/max level specified.

test_load_individual_pressure_levels_single()

Test loading individual Temperature with specific levels.

test_load_individual_pressure_levels_true()

Test loading Temperature with individual pressure datasets.

test_load_nonpressure_based()

Test loading all channels that aren't based on pressure.

test_load_pressure_based()

Test loading all channels based on pressure.

test_load_pressure_levels_min_max()

Test loading Temperature with min/max level specified.

test_load_pressure_levels_single()

Test loading a specific Temperature level.

test_load_pressure_levels_single_and_pressure_levels()

Test loading a specific Temperature level and pressure levels.

test_load_pressure_levels_true()

Test loading Temperature with all pressure levels.

yaml_file = 'nucaps.yaml'

satpy.tests.reader_tests.test_nwcsaf_msg module

Unittests for NWC SAF MSG (2013) reader.

class satpy.tests.reader_tests.test_nwcsaf_msg.**TestH5NWCSAF**(*methodName='runTest'*)

Bases: TestCase

Test the nwcsaf msg reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the tests.

tearDown()

Destroy.

test_get_area_def()

Get the area definition.

test_get_dataset()

Retrieve datasets from a NWCSAF msgv2013 hdf5 file.

satpy.tests.reader_tests.test_nwcsaf_nc module

Unittests for NWC SAF reader.

class satpy.tests.reader_tests.test_nwcsaf_nc.**TestNcNWCSAFFileKeyPrefix**

Bases: object

Test the NcNWCSAF reader when using a file key prefix.

test_get_dataset_scales_and_offsets_palette_meanings_using_other_dataset(*nwcsaf_pps_cmic_filehandler*)

Test that get_dataset() returns scaled palette_meanings using another dataset as scaling source.

test_get_dataset_uses_file_key_prefix(*nwcsaf_pps_cmic_filehandler*)

Test that get_dataset() uses a file_key_prefix.

class satpy.tests.reader_tests.test_nwcsaf_nc.**TestNcNWCSAFGeo**

Bases: object

Test the NcNWCSAF reader for Geo products.

test_end_time(*nwcsaf_geo_ct_filehandler*)

Test the end time property.

test_get_area_def(*nwcsaf_geo_ct_filehandler*)

Test that get_area_def() returns proper area.

test_get_area_def_km(*nwcsaf_old_geo_ct_filehandler*)

Test that get_area_def() returns proper area when the projection is in km.

test_orbital_parameters_are_correct(*nwcsaf_geo_ct_filehandler*)

Test that orbital parameters are present in the dataset attributes.

test_scale_dataset_attr_removal(*nwcsaf_geo_ct_filehandler*)

Test the scaling of the dataset and removal of obsolete attributes.

test_scale_dataset_floating(*nwcsaf_geo_ct_filehandler*, *attrs*, *expected*)

Test the scaling of the dataset with floating point values.

test_scale_dataset_floating_nwcsaf_geo_ctth(*nwcsaf_geo_ct_filehandler*)

Test the scaling of the dataset with floating point values for CTTH NWCSAF/Geo v2016/v2018.

test_sensor_name_platform(*nwcsaf_geo_ct_filehandler*, *platform*, *instrument*)

Test that the correct sensor name is being set.

test_sensor_name_sat_id(*nwcsaf_geo_ct_filehandler*, *platform*, *instrument*)

Test that the correct sensor name is being set.

test_start_time(*nwcsaf_geo_ct_filehandler*)

Test the start time property.

test_times_are_in_dataset_attributes(*nwcsaf_geo_ct_filehandler*)

Check that start/end times are in the attributes of datasets.

class satpy.tests.reader_tests.test_nwcsaf_nc.**TestNcNWCSAFPPS**

Bases: object

Test the NcNWCSAF reader for PPS products.

test_drop_xycoords(*nwcsaf_pps_cmic_filehandler*)

Test the drop of x and y coords.

test_end_time(*nwcsaf_pps_cmic_filehandler*)

Test the start time property.

test_get_dataset_can_handle_file_key_list(*nwcsaf_pps_cmic_filehandler*, *nwcsaf_pps_cpp_filehandler*)

Test that get_dataset() can handle a list of file_keys.

test_get_dataset_raises_when_dataset_missing(*nwcsaf_pps_cpp_filehandler*)

Test that get_dataset() raises an error when the requested dataset is missing.

test_get_dataset_scales_and_offsets(*nwcsaf_pps_cpp_filehandler*)

Test that get_dataset() returns scaled and offseted data.

test_get_dataset_scales_and_offsets_palette_meanings_using_other_dataset(*nwcsaf_pps_cpp_filehandler*)

Test that get_dataset() returns scaled palette_meanings with another dataset as scaling source.

test_get_dataset_uses_file_key_if_present(*nwcsaf_pps_cmic_filehandler*, *nwcsaf_pps_cpp_filehandler*)

Test that get_dataset() uses a file_key if present.

test_get_palette_fill_value_color_added(*nwcsaf_pps_ctth_filehandler*)

Test that get_dataset() returns scaled palette_meanings with fill_value_color added.

test_start_time(*nwcsaf_pps_cmic_filehandler*)

Test the start time property.

satpy.tests.reader_tests.test_nwcsaf_nc.**_check_area_def**(*area_definition*)

satpy.tests.reader_tests.test_nwcsaf_nc.**create_cmic_file**(*path*, *filetype*, *attrs={'gdal_projection': '+proj=geos +a=6378137.000 +b=6356752.300 +lon_0=0.000000 +h=35785863.000', 'gdal_xgeo_low_right': 5566500.0, 'gdal_xgeo_up_left': -5569500.0, 'gdal_ygeo_low_right': 2653500.0, 'gdal_ygeo_up_left': 5437500.0, 'satellite_identifier': 'MSG4', 'source': 'NWC/GEO version v2021.1', 'sub-satellite_longitude': 0.0, 'time_coverage_end': '2023-01-18T10:42:22Z', 'time_coverage_start': '2023-01-18T10:39:17Z'}*)

Create a cmic file.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_cot_pal_variable**(*nc_file*, *var_name*)

Create a palette variable.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_cot_variable**(*nc_file*, *var_name*)

Create a COT variable.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_cre_variables**(*nc_file*, *var_name*)

Create a CRE variable.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_ctth_alti_pal_variable_with_fill_value_color**(*nc_file*, *var_name*)

Create a palette variable.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_ctth_file**(*path*, *attrs={'gdal_projection': '+proj=geos +a=6378137.000 +b=6356752.300 +lon_0=0.000000 +h=35785863.000', 'gdal_xgeo_low_right': 5566500.0, 'gdal_xgeo_up_left': -5569500.0, 'gdal_ygeo_low_right': 2653500.0, 'gdal_ygeo_up_left': 5437500.0, 'satellite_identifier': 'MSG4', 'source': 'NWC/GEO version v2021.1', 'sub-satellite_longitude': 0.0, 'time_coverage_end': '2023-01-18T10:42:22Z', 'time_coverage_start': '2023-01-18T10:39:17Z'}*)

Create a cmic file.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_ctth_variables**(*nc_file*, *var_name*)

Create a CRE variable.

satpy.tests.reader_tests.test_nwcsaf_nc.**create_nwcsaf_geo_ct_file**(*directory*, *attrs={'gdal_projection': '+proj=geos +a=6378137.000 +b=6356752.300 +lon_0=0.000000 +h=35785863.000', 'gdal_xgeo_low_right': 5566500.0, 'gdal_xgeo_up_left': -5569500.0, 'gdal_ygeo_low_right': 2653500.0, 'gdal_ygeo_up_left': 5437500.0, 'nominal_product_time': '2023-01-18T10:30:00Z', 'satellite_identifier': 'MSG4', 'source': 'NWC/GEO version v2021.1', 'sub-satellite_longitude': 0.0, 'time_coverage_end': '2023-01-18T10:42:22Z', 'time_coverage_start': '2023-01-18T10:39:17Z'}*)

Create a CT file.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_geo_ct_filehandler**(*nwcsaf_geo_ct_filename*)

Create a CT filehandler.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_geo_ct_filename**(*tmp_path_factory*)

Create a CT file and return the filename.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_old_geo_ct_filehandler**(*nwcsaf_old_geo_ct_filename*)

Create a CT filehandler.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_old_geo_ct_filename**(*tmp_path_factory*)

Create a CT file and return the filename.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_pps_cmic_filehandler**(*nwcsaf_pps_cmic_filename*)

Create a CMIC filehandler.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_pps_cmic_filename**(*tmp_path_factory*)

Create a CMIC file.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_pps_cpp_filehandler**(*nwcsaf_pps_cpp_filename*)

Create a CPP filehandler.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_pps_cpp_filename**(*tmp_path_factory*)

Create a CPP file.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_pps_ctth_filehandler**(*nwcsaf_pps_ctth_filename*)

Create a CMIC filehandler.

satpy.tests.reader_tests.test_nwcsaf_nc.**nwcsaf_pps_ctth_filename**(*tmp_path_factory*)

Create a CTTH file.

satpy.tests.reader_tests.test_oceancolorcci_l3_nc module

Module for testing the satpy.readers.oceancolorcci_l3_nc module.

class satpy.tests.reader_tests.test_oceancolorcci_l3_nc.**TestOCCCIReader**

Bases: object

Test the Ocean Color reader.

_create_reader_for_resolutions(*filename*)

area_exp()

Get expected area definition.

setup_method()

Set up the reader tests.

test_bad_fname(*fake_dataset*, *fake_file_dict*)

Test case where an incorrect composite period is given.

test_correct_dimnames(*fake_file_dict*)

Check that the loaded dimension names are correct.

test_end_time(*fake_file_dict*)

Test end time property.

test_get_area_def(*area_exp*, *fake_file_dict*)

Test area definition.

test_get_dataset_1d_kprods(*fake_dataset*, *fake_file_dict*)

Test dataset loading.

test_get_dataset_5d_allprods(*fake_dataset*, *fake_file_dict*)

Test dataset loading.

test_get_dataset_8d_iopprods(*fake_dataset*, *fake_file_dict*)

Test dataset loading.

test_get_dataset_monthly_allprods(*fake_dataset*, *fake_file_dict*)

Test dataset loading.

test_start_time(*fake_file_dict*)

Test start time property.

satpy.tests.reader_tests.test_oceancolorcci_l3_nc.**fake_dataset**()

Create a CLAAS-like test dataset.

satpy.tests.reader_tests.test_oceancolorcci_l3_nc.**fake_file_dict**(*fake_dataset*, *tmp_path*)

Write a fake dataset to file.

satpy.tests.reader_tests.test_olci_nc module

Module for testing the satpy.readers.olci_nc module.

class satpy.tests.reader_tests.test_olci_nc.**TestBitFlags**(*methodName='runTest'*)

Bases: TestCase

Test the bitflag reading.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_bitflags()

Test the BitFlags class.

class satpy.tests.reader_tests.test_olci_nc.**TestOLCIReader**(*methodName='runTest'*)

Bases: TestCase

Test various olci_nc filehandlers.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_chl_nn(*mocked_dataset*)

Test unlogging the chl_nn product.

test_get_mask(*mocked_dataset*)

Test reading datasets.

test_get_mask_with_alternative_items(*mocked_dataset*)

Test reading datasets.

test_instantiate(*mocked_dataset*)

Test initialization of file handlers.

test_olci_angles(*mocked_dataset*)

Test reading datasets.

test_olci_meteo(*mocked_dataset*)

Test reading datasets.

test_open_file_objects(*mocked_open_dataset*)

Test initialization of file handlers.

satpy.tests.reader_tests.test_omps_edr module

Module for testing the satpy.readers.omps_edr module.

class satpy.tests.reader_tests.test_omps_edr.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_omps_edr.**TestOMPSEDRReader**(*methodName='runTest'*)

Bases: TestCase

Test OMPS EDR Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_basic_load_so2()

Test basic load of so2 datasets.

test_basic_load_to3()

Test basic load of to3 datasets.

test_init()

Test basic init with no extra parameters.

test_load_so2_DIMENSION_LIST(*mock_h5py_file*, *mock_hdf5_utils_get_reference*)

Test load of so2 datasets with DIMENSION_LIST.

yaml_file = 'omps_edr.yaml'

satpy.tests.reader_tests.test_osisaf_l3 module

Module for testing the satpy.readers.osisaf_l3 module.

class satpy.tests.reader_tests.test_osisaf_l3.**OSISAFL3ReaderTests**

Bases: object

Test OSI-SAF level 3 netCDF reader ice files.

setup_method(*tester='ice'*)

Create a fake dataset.

test_get_area_def_bad(*tmp_path*)

Test getting the area definition for the polar stereographic grid.

test_get_dataset(*tmp_path*)

Test retrieval of datasets.

test_get_start_and_end_times(*tmp_path*)

Test retrieval of the sensor name from the netCDF file.

test_instantiate_single_netcdf_file(*tmp_path*)

Test initialization of file handlers - given a single netCDF file.

class satpy.tests.reader_tests.test_osisaf_l3.**TestOSISAFL3ReaderFluxGeo**

Bases: *OSISAFL3ReaderTests*

Test OSI-SAF level 3 netCDF reader flux files on lat/lon grid (GEO sensors).

setup_method()

Set up the tests.

test_get_area_def_grid(*tmp_path*)

Test getting the area definition for the lat/lon grid.

class satpy.tests.reader_tests.test_osisaf_l3.**TestOSISAFL3ReaderFluxStere**

Bases: *OSISAFL3ReaderTests*

Test OSI-SAF level 3 netCDF reader flux files on stereographic grid.

setup_method()

Set up the tests.

test_get_area_def_stere(*tmp_path*)

Test getting the area definition for the polar stereographic grid.

class satpy.tests.reader_tests.test_osisaf_l3.**TestOSISAFL3ReaderICE**

Bases: *OSISAFL3ReaderTests*

Test OSI-SAF level 3 netCDF reader ice files.

setup_method()

Set up the tests.

test_get_area_def_ease(*tmp_path*)

Test getting the area definition for the EASE grid.

test_get_area_def_stere(*tmp_path*)

Test getting the area definition for the polar stereographic grid.

class satpy.tests.reader_tests.test_osisaf_l3.**TestOSISAFL3ReaderSST**

Bases: *OSISAFL3ReaderTests*

Test OSI-SAF level 3 netCDF reader surface temperature files.

setup_method()

Set up the tests.

test_get_area_def_stere(*tmp_path*)

Test getting the area definition for the polar stereographic grid.

satpy.tests.reader_tests.test_safe_sar_l2_ocn module

Module for testing the satpy.readers.safe_sar_l2_ocn module.

class satpy.tests.reader_tests.test_safe_sar_l2_ocn.**TestSAFENC**(*methodName='runTest'*)

Bases: TestCase

Test various SAFE SAR L2 OCN file handlers.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*xr_*)

Set up the tests.

test_get_dataset()

Test getting a dataset.

test_init()

Test reader initialization.

satpy.tests.reader_tests.test_sar_c_safe module

Module for testing the satpy.readers.sar-c_safe module.

class satpy.tests.reader_tests.test_sar_c_safe.**Calibration**(*value*)

Bases: Enum

Calibration levels.

beta_nought = 3

dn = 4

gamma = 1

sigma_nought = 2

class satpy.tests.reader_tests.test_sar_c_safe.**TestSAFEGRD**(*methodName='runTest'*)

Bases: TestCase

Test the SAFE GRD file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*mocked_rio_open*)

Set up the test case.

test_instantiate()

Test initialization of file handlers.

test_read_calibrated_dB(*mocked_xarray_open*)

Test the calibration routines.

test_read_calibrated_natural(*mocked_xarray_open*)

Test the calibration routines.

test_read_lon_lats()

Test reading lons and lats.

class satpy.tests.reader_tests.test_sar_c_safe.**TestSAFEXMLAnnotation**(*methodName='runTest'*)

Bases: TestCase

Test the SAFE XML Annotation file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_incidence_angle()

Test reading the incidence angle.

class satpy.tests.reader_tests.test_sar_c_safe.**TestSAFEXMLCalibration**(*methodName='runTest'*)

Bases: TestCase

Test the SAFE XML Calibration file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_beta_calibration_array()

Test reading the beta calibration array.

test_dn_calibration_array()

Test reading the dn calibration array.

test_gamma_calibration_array()

Test reading the gamma calibration array.

test_get_calibration_constant()

Test getting the calibration constant.

test_get_calibration_dataset()

Test using get_dataset for the calibration.

test_get_calibration_dataset_has_right_chunk_size()

Test using get_dataset for the calibration yields array with right chunksize.

test_sigma_calibration_array()

Test reading the sigma calibration array.

class satpy.tests.reader_tests.test_sar_c_safe.**TestSAFEXMLNoise**(*methodName='runTest'*)

Bases: TestCase

Test the SAFE XML Noise file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_azimuth_noise_array()

Test reading the azimuth-noise array.

test_azimuth_noise_array_with_holes()

Test reading the azimuth-noise array.

test_get_noise_dataset()

Test using get_dataset for the noise.

test_get_noise_dataset_has_right_chunk_size()

Test using get_dataset for the noise has right chunk size in result.

test_range_noise_array()

Test reading the range-noise array.

satpy.tests.reader_tests.test_satpy_cf_nc module

Tests for the CF reader.

class satpy.tests.reader_tests.test_satpy_cf_nc.**TestCFReader**

Bases: object

Test case for CF reader.

test_dataid_attrs_equal_contains_not_matching_key(*cf_scene*, *nc_filename*)

Check that get_dataset returns valid dataset when dataid have key(s) not existing in data.

test_dataid_attrs_equal_matching_dataset(*cf_scene*, *nc_filename*)

Check that get_dataset returns valid dataset when keys matches.

test_dataid_attrs_equal_not_matching_dataset(*cf_scene*, *nc_filename*)

Check that get_dataset returns None when key(s) are not matching.

test_fix_modifier_attr()

Check that fix modifier can handle empty list as modifier attribute.

test_orbital_parameters(*cf_scene*, *nc_filename*)

Test that the orbital parameters in attributes are handled correctly.

test_read_prefixed_channels(*cf_scene*, *nc_filename*)

Check channels starting with digit is prefixed and read back correctly.

test_read_prefixed_channels_by_user(*cf_scene*, *nc_filename*)

Check channels starting with digit is prefixed by user and read back correctly.

test_read_prefixed_channels_by_user2(*cf_scene*, *nc_filename*)

Check channels starting with digit is prefixed by user when saving and read back correctly without prefix.

test_read_prefixed_channels_by_user_include_prefix(*cf_scene*, *nc_filename*)

Check channels starting with digit is prefixed by user and include original name when saving.

test_read_prefixed_channels_by_user_no_prefix(*cf_scene*, *nc_filename*)

Check channels starting with digit is not prefixed by user.

test_read_prefixed_channels_include_orig_name(*cf_scene*, *nc_filename*)

Check channels starting with digit and includeed orig name is prefixed and read back correctly.

test_write_and_read_from_two_files(*nc_filename*, *nc_filename_i*)

Save two datasets with different resolution and read the solar_zenith_angle again.

test_write_and_read_with_area_definition(*cf_scene*, *nc_filename*)

Save a dataset with an area definition to file with cf_writer and read the data again.

test_write_and_read_with_swath_definition(*cf_scene*, *nc_filename*)

Save a dataset with a swath definition to file with cf_writer and read the data again.

satpy.tests.reader_tests.test_satpy_cf_nc.**_create_test_netcdf**(*filename*, *resolution=742*)

satpy.tests.reader_tests.test_satpy_cf_nc.**cf_scene**()

Create a cf scene.

satpy.tests.reader_tests.test_satpy_cf_nc.**nc_filename**(*tmp_path*)

Create an nc filename for viirs m band.

satpy.tests.reader_tests.test_satpy_cf_nc.**nc_filename_i**(*tmp_path*)

Create an nc filename for viirs i band.

satpy.tests.reader_tests.test_scmi module

The scmi_abi_l1b reader tests package.

class satpy.tests.reader_tests.test_scmi.**FakeDataset**(*info*, *attrs*, *dims=None*)

Bases: object

Fake dataset.

Init the dataset.

close()

Close the dataset.

rename(**args*, ***kwargs*)

Rename the dataset.

class satpy.tests.reader_tests.test_scmi.**TestSCMIFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the SCMIFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*xr_*)

Set up for test.

test_basic_attributes()

Test getting basic file attributes.

test_data_load()

Test data loading.

class satpy.tests.reader_tests.test_scmi.**TestSCMIFileHandlerArea**(*methodName='runTest'*)

Bases: TestCase

Test the SCMIFileHandler's area creation.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

create_reader(*proj_name*, *proj_attrs*, *xr_*)

Create a fake reader.

test_get_area_def_bad(*adef*)

Test the area generation for bad projection.

test_get_area_def_geos(*adef*)

Test the area generation for geos projection.

test_get_area_def_lcc(*adef*)

Test the area generation for lcc projection.

test_get_area_def_merc(*adef*)

Test the area generation for merc projection.

test_get_area_def_stere(*adef*)

Test the area generation for stere projection.

satpy.tests.reader_tests.test_seadas_l2 module

Tests for the 'seadas_l2' reader.

class satpy.tests.reader_tests.test_seadas_l2.**TestSEADAS**

Bases: object

Test the SEADAS L2 file reader.

test_available_reader()

Test that SEADAS L2 reader is available.

test_load_chlor_a(*input_files*, *exp_plat*, *exp_sensor*, *exp_rps*, *apply_quality_flags*)

Test that we can load 'chlor_a'.

test_scene_available_datasets(*input_files*)

Test that datasets are available.

satpy.tests.reader_tests.test_seadas_l2.**_add_variable_to_hdf4_file**(*h*, *var_name*, *var_info*)

satpy.tests.reader_tests.test_seadas_l2.**_add_variable_to_netcdf_file**(*nc*, *var_name*, *var_info*)

satpy.tests.reader_tests.test_seadas_l2.**_create_seadas_chlor_a_hdf4_file**(*full_path*, *mission*, *sensor*)

satpy.tests.reader_tests.test_seadas_l2.**_create_seadas_chlor_a_netcdf_file**(*full_path*, *mission*, *sensor*)

satpy.tests.reader_tests.test_seadas_l2.**seadas_l2_modis_chlor_a**(*tmp_path_factory*)

Create MODIS SEADAS file.

satpy.tests.reader_tests.test_seadas_l2.**seadas_l2_modis_chlor_a_netcdf**(*tmp_path_factory*)

Create MODIS SEADAS NetCDF file.

satpy.tests.reader_tests.test_seadas_l2.**seadas_l2_viirs_j01_chlor_a**(*tmp_path_factory*)

Create VIIRS JPSS-01 SEADAS file.

satpy.tests.reader_tests.test_seadas_l2.**seadas_l2_viirs_npp_chlor_a**(*tmp_path_factory*)

Create VIIRS NPP SEADAS file.

satpy.tests.reader_tests.test_seviri_base module

Test the MSG common (native and hrit format) functionionalities.

class satpy.tests.reader_tests.test_seviri_base.**SeviriBaseTest**(*methodName='runTest'*)

Bases: TestCase

Test SEVIRI base.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

observation_end_time()

Get scan end timestamp for testing.

observation_start_time()

Get scan start timestamp for testing.

test_chebyshev()

Test the chebyshev function.

test_dec10216()

Test the dec10216 function.

test_get_cds_time()

Test the get_cds_time function.

static test_get_padding_area_float()

Test padding area generator for floats.

static test_get_padding_area_int()

Test padding area generator for integers.

static test_pad_data_horizontally()

Test the horizontal hrv padding.

test_pad_data_horizontally_bad_shape()

Test the error handling for the horizontal hrv padding.

static test_pad_data_vertically()

Test the vertical hrv padding.

test_pad_data_vertically_bad_shape()

Test the error handling for the vertical hrv padding.

test_round_nom_time()

Test the rouding of start/end_time.

class satpy.tests.reader_tests.test_seviri_base.**TestMeirinkSlope**

Bases: object

Unit tests for the slope of Meirink calibration.

test_get_meirink_slope_2020(*platform_id*, *time*, *expected*)

Test the value of the slope of the Meirink calibration.

test_get_meirink_slope_epoch(*platform_id*, *channel_name*)

Test the value of the slope of the Meirink calibration on 2000-01-01.

class satpy.tests.reader_tests.test_seviri_base.**TestOrbitPolynomialFinder**

Bases: object

Unit tests for orbit polynomial finder.

test_get_orbit_polynomial(*orbit_polynomials*, *time*, *orbit_polynomial_exp*)

Test getting the satellite locator.

test_get_orbit_polynomial_exceptions(*orbit_polynomials*, *time*)

Test exceptions thrown while getting the satellite locator.

class satpy.tests.reader_tests.test_seviri_base.**TestSatellitePosition**

Bases: object

Test locating the satellite.

orbit_polynomial()

Get an orbit polynomial for testing.

test_eval_polynomial(*orbit_polynomial*, *time*)

Test getting the position in cartesian coordinates.

test_get_satpos(*orbit_polynomial*, *time*)

Test getting the position in geodetic coordinates.

time()

Get scan timestamp for testing.

satpy.tests.reader_tests.test_seviri_base.**chebyshev4**(*c*, *x*, *domain*)

Evaluate 4th order Chebyshev polynomial.

satpy.tests.reader_tests.test_seviri_l1b_calibration module

Unittesting the native msg reader.

class satpy.tests.reader_tests.test_seviri_l1b_calibration.**TestFileHandlerCalibrationBase**

Bases: object

Base class for file handler calibration tests.

_get_expected(*channel*, *calibration*, *calib_mode*, *use_ext_coefs*)

counts()

Provide fake image counts.

```
expected = {'HRV': {'counts': {'NOMINAL': <xarray.DataArray (y: 2, x: 2)>
array([[ 0, 10], [100, 255]]) Dimensions without coordinates: y, x}, 'radiance':
{'EXTERNAL': <xarray.DataArray (y: 2, x: 2)> array([[ nan, 45.], [ 495., 1270.]])
Dimensions without coordinates: y, x, 'GSICS': <xarray.DataArray (y: 2, x: 2)>
array([[ nan, 108.], [1188., 3048.]]) Dimensions without coordinates: y, x,
'NOMINAL': <xarray.DataArray (y: 2, x: 2)> array([[ nan, 108.], [1188., 3048.]])
Dimensions without coordinates: y, x}, 'reflectance': {'EXTERNAL':
<xarray.DataArray (y: 2, x: 2)> array([[ nan, 173.02817], [1903.31 , 4883.2397 ]])
Dimensions without coordinates: y, x, 'NOMINAL': <xarray.DataArray (y: 2, x: 2)>
array([[ nan, 415.26767], [ 4567.944 , 11719.775 ]]) Dimensions without coordinates:
y, x}}, 'IR_108': {'brightness_temperature': {'EXTERNAL': <xarray.DataArray (y:
2, x: 2)> array([[ nan, 335.14236], [ 758.6249 , 1262.7567 ]]) Dimensions without
coordinates: y, x, 'GSICS': <xarray.DataArray (y: 2, x: 2)> array([[ nan,
189.20985], [285.53293, 356.06668]]) Dimensions without coordinates: y, x,
'NOMINAL': <xarray.DataArray (y: 2, x: 2)> array([[ nan, 279.82318], [543.2585 ,
812.77167]]) Dimensions without coordinates: y, x}, 'counts': {'NOMINAL':
<xarray.DataArray (y: 2, x: 2)> array([[ 0, 10], [100, 255]]) Dimensions without
coordinates: y, x}, 'radiance': {'EXTERNAL': <xarray.DataArray (y: 2, x: 2)>
array([[ nan, 180.], [1980., 5080.]]) Dimensions without coordinates: y, x,
'GSICS': <xarray.DataArray (y: 2, x: 2)> array([[ nan, 8.19], [ 89.19, 228.69]])
Dimensions without coordinates: y, x, 'NOMINAL': <xarray.DataArray (y: 2, x: 2)>
array([[ nan, 81.], [ 891., 2286.]]) Dimensions without coordinates: y, x}},
'VIS006': {'counts': {'NOMINAL': <xarray.DataArray (y: 2, x: 2)> array([[ 0,
10], [100, 255]]) Dimensions without coordinates: y, x}, 'radiance': {'EXTERNAL':
<xarray.DataArray (y: 2, x: 2)> array([[ nan, 90.], [ 990., 2540.]]) Dimensions
without coordinates: y, x, 'GSICS': <xarray.DataArray (y: 2, x: 2)> array([[ nan,
9.], [ 99., 254.]]) Dimensions without coordinates: y, x, 'NOMINAL':
<xarray.DataArray (y: 2, x: 2)> array([[ nan, 9.], [ 99., 254.]]) Dimensions
without coordinates: y, x}, 'reflectance': {'EXTERNAL': <xarray.DataArray (y: 2,
x: 2)> array([[ nan, 418.89853], [ 4607.8843 , 11822.249 ]]) Dimensions without
coordinates: y, x, 'NOMINAL': <xarray.DataArray (y: 2, x: 2)> array([[ nan,
41.88985], [ 460.7884 , 1182.2247 ]]) Dimensions without coordinates: y, x}}}

external_coefs = {'HRV': {'gain': 5, 'offset': -5}, 'IR_108': {'gain': 20,
'offset': -20}, 'VIS006': {'gain': 10, 'offset': -10}}

gains_gsics = [0, 0, 0, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 0]

gains_nominal = array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

offsets_gsics = [0, 0, 0, -0.4, -0.5, -0.6, -0.7, -0.8, -0.9, -1.0, -1.1, 0]

offsets_nominal = array([ -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12])

platform_id = 324

radiance_types = array([2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])

scan_time = datetime.datetime(2020, 1, 1, 0, 0)

spectral_channel_ids = {'HRV': 12, 'IR_108': 9, 'VIS006': 1}
```

class satpy.tests.reader_tests.test_seviri_l1b_calibration.**TestSEVIRICalibrationAlgorithm**(*methodName='ru*

Bases: TestCase

Unit Tests for SEVIRI calibration algorithm.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the SEVIRI Calibration algorithm for testing.

test_convert_to_radiance()

Test the conversion from counts to radiances.

test_ir_calibrate()

Test conversion from radiance to brightness temperature.

test_vis_calibrate()

Test conversion from radiance to reflectance.

class satpy.tests.reader_tests.test_seviri_l1b_calibration.**TestSeviriCalibrationHandler**

Bases: object

Unit tests for SEVIRI calibration handler.

_get_calibration_handler(*calib_mode='NOMINAL'*, *ext_coefs=None*)

Provide a calibration handler.

test_calibrate_exceptions()

Test exceptions raised by the calibration handler.

test_get_gain_offset(*calib_mode*, *ext_coefs*, *expected*)

Test selection of gain and offset.

test_init()

Test initialization of the calibration handler.

satpy.tests.reader_tests.test_seviri_l1b_hrit module

The HRIT msg reader tests package.

class satpy.tests.reader_tests.test_seviri_l1b_hrit.**TestHRITMSGBase**(*methodName='runTest'*)

Bases: TestCase

Baseclass for SEVIRI HRIT reader tests.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

assert_attrs_equal(*attrs*, *attrs_exp*)

Assert equality of dataset attributes.

class satpy.tests.reader_tests.test_seviri_l1b_hrit.**TestHRITMSGCalibration**

Bases: *TestFileHandlerCalibrationBase*

Unit tests for calibration.

file_handler()

Create a mocked file handler.

test_calibrate(*file_handler*, *counts*, *channel*, *calibration*, *calib_mode*, *use_ext_coefs*)

Test the calibration.

test_mask_bad_quality(*file_handler*)

Test the masking of bad quality scan lines.

class satpy.tests.reader_tests.test_seviri_l1b_hrit.**TestHRITMSGEpilogueFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRIT epilogue file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*init*, **mocks*)

Set up the test case.

test_extra_kwargs(*init*, **mocks*)

Test whether the epilogue file handler accepts extra keyword arguments.

test_reduce(*reduce_mda*)

Test metadata reduction.

class satpy.tests.reader_tests.test_seviri_l1b_hrit.**TestHRITMSGFileHandler**(*methodName='runTest'*)

Bases: *TestHRITMSGBase*

Test the HRITFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_fake_data()

setUp()

Set up the hrit file handler for testing.

test_get_area_def()

Test getting the area def.

test_get_dataset(*calibrate*, *parent_get_dataset*)

Test getting the dataset.

test_get_dataset_with_raw_metadata(*calibrate*, *parent_get_dataset*)

Test getting the dataset.

test_get_dataset_without_masking_bad_scan_lines(*calibrate*, *parent_get_dataset*)

Test getting the dataset.

test_get_raw_mda()

Test provision of raw metadata.

test_read_band(*memmap*)

Test reading a band.

test_satpos_no_valid_orbit_polynomial()

Test satellite position if there is no valid orbit polynomial.

class satpy.tests.reader_tests.test_seviri_l1b_hrit.**TestHRITMSGFileHandlerHRV**(*methodName='runTest'*)

Bases: *TestHRITMSGBase*

Test the HRITFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the hrit file handler for testing HRV.

test_get_area_def()

Test getting the area def.

test_get_dataset(*calibrate*, *parent_get_dataset*)

Test getting the hrv dataset.

test_get_dataset_non_fill(*calibrate*, *parent_get_dataset*)

Test getting a non-filled hrv dataset.

test_read_hrv_band(*memmap*)

Test reading the hrv band.

class satpy.tests.reader_tests.test_seviri_l1b_hrit.**TestHRITMSGPrologueFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the HRIT prologue file handler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(**mocks*)

Set up the test case.

test_extra_kwargs(*init*, **mocks*)

Test whether the prologue file handler accepts extra keyword arguments.

test_reduce(*reduce_mda*)

Test metadata reduction.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup module

Setup for SEVIRI HRIT reader tests.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_acq_time_cds**(*start_time*, *nlines*)

Get fake scanline acquisition times.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_acq_time_exp**(*start_time*, *nlines*)

Get expected scanline acquisition times.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_attrs_exp**(*projection_longitude=0.0*)

Get expected dataset attributes.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_fake_dataset_info**()

Create fake dataset info.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_fake_epilogue**()

Create a fake HRIT epilogue.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_fake_file_handler**(*observation_start_time*, *nlines*, *ncols*, *projection_longitude=0*, *orbit_polynomials={'EndTime': array([[datetime.datetime(2006, 1, 1, 12, 0), datetime.datetime(2006, 1, 1, 18, 0), datetime.datetime(2006, 1, 2, 0, 0), datetime.datetime(1958, 1, 1, 0, 0)]], dtype=object), 'StartTime': array([[datetime.datetime(2006, 1, 1, 6, 0), datetime.datetime(2006, 1, 1, 12, 0), datetime.datetime(2006, 1, 1, 18, 0), datetime.datetime(1958, 1, 1, 0, 0)]], dtype=object), 'X': [array([0., 0., 0., 0., 0., 0., 0., 0.]), [84160.7082, 2.9431926, 0.986748617, -0.270135453, -0.038436465, 0.00848718433, 0.000770548174, -0.000144262718], array([0., 0., 0., 0., 0., 0., 0., 0.])], 'Y': [array([0., 0., 0., 0., 0., 0., 0., 0.]), [-5211.70255, 5.12998948, -1.33370453, -0.309634144, 0.0618232793, 0.00750505681, -0.00135131011, -0.000112054405], array([0., 0., 0., 0., 0., 0., 0., 0.])], 'Z': [array([0., 0., 0., 0., 0., 0., 0., 0.]), [-651.293855, 145.830459, 56.13794, -3.90970565, -0.738137565, 0.0306131644, 0.00382892428, -0.000112739309],*

Create a mocked SEVIRI HRIT file handler.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_fake_filename_info**(*start_time*)

Create fake filename information.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_fake_mda**(*nlines*, *ncols*, *start_time*)

Create fake metadata.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_fake_prologue**(*projection_longitude*, *orbit_polynomials*)

Create a fake HRIT prologue.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**get_new_read_prologue**(*prologue*)

Create mocked read_prologue() method.

satpy.tests.reader_tests.test_seviri_l1b_hrit_setup.**new_get_hd**(*instance*, *hdr_info*)

Generate some metadata.

satpy.tests.reader_tests.test_seviri_l1b_icare module

Tests for the SEVIRI L1b HDF4 from ICARE reader.

class satpy.tests.reader_tests.test_seviri_l1b_icare.**FakeHDF4FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF4FileHandler*

Swap in HDF4 file handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filename_type*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_seviri_l1b_icare.**TestSEVIRIICAREReader**(*methodName='runTest'*)

Bases: TestCase

Test SEVIRI L1b HDF4 from ICARE Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

compare_areas(*v*)

Compare produced AreaDefinition with expected.

setUp()

Wrap HDF4 file handler with own fake file handler.

tearDown()

Stop wrapping the HDF4 file handler.

test_area_def_hires()

Test loading all datasets from an area of interest file.

test_area_def_lores()

Test loading all datasets from an area of interest file.

test_bad_bandname()

Check reader raises an error if a band bandname is passed.

test_init()

Test basic init with no extra parameters.

test_load_dataset_ir()

Test loading all datasets from a full swath file.

test_load_dataset_vis()

Test loading all datasets from a full swath file.

test_nocompute()

Test that dask does not compute anything in the reader itself.

test_sensor_names()

Check satellite name conversion is correct, including error case.

yaml_file = 'seviri_l1b_icare.yaml'

satpy.tests.reader_tests.test_seviri_l1b_native module

Unittesting the Native SEVIRI reader.

class satpy.tests.reader_tests.test_seviri_l1b_native.**TestNativeMSGCalibration**

Bases: *TestFileHandlerCalibrationBase*

Unit tests for calibration.

file_handler()

Create a mocked file handler.

test_calibrate(*file_handler*, *counts*, *channel*, *calibration*, *calib_mode*, *use_ext_coefs*)

Test the calibration.

class satpy.tests.reader_tests.test_seviri_l1b_native.**TestNativeMSGDataset**

Bases: object

Tests for getting the dataset.

static _exp_data_array()

static _fake_data()

static _fake_header()

file_handler()

Create a file handler for testing.

test_get_dataset(*file_handler*)

Test getting the dataset.

test_get_dataset_with_raw_metadata(*file_handler*)

Test provision of raw metadata.

test_repeat_cycle_duration(*file_handler*)

Test repeat cycle handling for FD or ReduscedScan.

test_satpos_no_valid_orbit_polynomial(*file_handler*)

Test satellite position if there is no valid orbit polynomial.

test_time(*file_handler*)

Test start/end nominal/observation time handling.

class satpy.tests.reader_tests.test_seviri_l1b_native.**TestNativeMSGFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the NativeMSGFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_get_available_channels()

Test the derivation of the available channel list.

class satpy.tests.reader_tests.test_seviri_l1b_native.**TestNativeMSGFilenames**

Bases: object

Test identification of Native format filenames.

reader()

Return reader for SEVIRI Native format.

test_file_pattern(*reader*)

Test file pattern matching.

class satpy.tests.reader_tests.test_seviri_l1b_native.**TestNativeMSGPadder**(*methodName='runTest'*)

Bases: TestCase

Test Padder of the native l1b seviri reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

static prepare_padder(*test_dict*)

Initialize Padder and pad test data.

test_padder_fes_hrv()

Test padder for FES HRV data.

test_padder_rss_roi()

Test padder for RSS and ROI data (applies to both VISIR and HRV).

satpy.tests.reader_tests.test_seviri_l1b_native.**create_test_header**(*earth_model*, *dataset_id*, *is_full_disk*, *is_rapid_scan*, *good_qual='OK'*)

Create test header for SEVIRI L1.5 product.

Header includes mandatory attributes for NativeMSGFileHandler.get_area_extent

satpy.tests.reader_tests.test_seviri_l1b_native.**create_test_trailer**(*is_rapid_scan*)

Create test trailer for SEVIRI L1.5 product.

Trailer includes mandatory attributes for NativeMSGFileHandler.get_area_extent

satpy.tests.reader_tests.test_seviri_l1b_native.**prepare_area_definitions**(*test_dict*)

Prepare calculated and expected area definitions for equal checking.

satpy.tests.reader_tests.test_seviri_l1b_native.**prepare_is_roi**(*test_dict*)

Prepare calculated and expected check for region of interest data for equal checking.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_area_definitions**(*actual*, *expected*)

Test area definitions with only one area.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_has_archive_header**(*starts_with*, *expected*)

Test if the file includes an ASCII archive header.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_header_type**(*file_content*, *exp_header_size*)

Test identification of the file header type.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_header_warning**()

Test warning is raised for NOK quality flag.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_is_roi**(*actual*, *expected*)

Test if given area is of area-of-interest.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_read_header**()

Test that reading header returns the header correctly converted to a dictionary.

satpy.tests.reader_tests.test_seviri_l1b_native.**test_stacked_area_definitions**(*actual*, *expected*)

Test area definitions with stacked areas.

satpy.tests.reader_tests.test_seviri_l1b_nc module

The HRIT msg reader tests package.

class satpy.tests.reader_tests.test_seviri_l1b_nc.**TestNCSEVIRIFileHandler**

Bases: *TestFileHandlerCalibrationBase*

Unit tests for SEVIRI netCDF reader.

_get_fake_dataset(*counts*, *h5netcdf*)

Create a fake dataset.

Parameters

- **counts** (*xr.DataArray*) – Array with data.
- **h5netcdf** (*boolean*) – If True an array attribute will be created which is common for the h5netcdf backend in xarray for scalar values.

file_handler(*counts*, *h5netcdf*)
Create a mocked file handler.

h5netcdf()
Fixture for xr backend choice.

test_calibrate(*file_handler*, *channel*, *calibration*, *use_ext_coefs*)
Test the calibration.

test_get_dataset(*file_handler*, *channel*, *calibration*, *mask_bad_quality_scan_lines*)
Test getting the dataset.

test_h5netcdf_pecularity(*file_handler*, *h5netcdf*)
Test conversion of attributes when xarray is used with h5netcdf backend.

test_mask_bad_quality(*file_handler*)
Test masking of bad quality scan lines.

test_repeat_cycle_duration(*file_handler*)
Test repeat cycle handling for FD or ReduscedScan.

test_satpos_no_valid_orbit_polynomial(*file_handler*)
Test satellite position if there is no valid orbit polynomial.

test_time(*file_handler*)
Test start/end nominal/observation time handling.

satpy.tests.reader_tests.test_seviri_l1b_nc.**to_cds_time**(*time*)
Convert datetime to (days, msecs) since 1958-01-01.

satpy.tests.reader_tests.test_seviri_l2_bufr module

Unittesting the SEVIRI L2 BUFR reader.

class satpy.tests.reader_tests.test_seviri_l2_bufr.**SeviriL2AMVBufrData**(*filename*)
Bases: object

Mock SEVIRI L2 AMV BUFR data.

Initialize by mocking test data for testing the SEVIRI L2 BUFR reader.

class satpy.tests.reader_tests.test_seviri_l2_bufr.**SeviriL2BufrData**(*filename*, *with_adef=False*, *rect_lon='default'*)
Bases: object

Mock SEVIRI L2 BUFR data.

Initialize by mocking test data for testing the SEVIRI L2 BUFR reader.

get_data(*dataset_info*)
Read data from mock file.

class satpy.tests.reader_tests.test_seviri_l2_bufr.**TestSeviriL2AMVBufrReader**
Bases: object

Test SEVIRI L2 BUFR Reader for AMV data.

static test_amv_with_area_def()
Test that AMV data can not be loaded with an area definition.

class satpy.tests.reader_tests.test_seviri_l2_bufr.**TestSeviriL2BufrReader**

Bases: object

Test SEVIRI L2 BUFR Reader.

pytestmark = [Mark(name='parametrize', args=('input_file', ['ASRBUFRProd_20191106130000Z_00_OMPEFS02_MET09_FES_E0000', 'MSG2-SEVI-MSGASRE-0101-0101-20191106130000.000000000Z-20191106131702-1362128.bfr', 'MSG2-SEVI-MSGASRE-0101-0101-20191106101500.000000000Z-20191106103218-1362148']), kwargs={})]

static test_attributes_with_area_definition(*input_file*)

Test correctness of dataset attributes with data loaded with a AreaDefinition.

static test_attributes_with_swath_definition(*input_file*)

Test correctness of dataset attributes with data loaded with a SwathDefinition (default behaviour).

test_data_with_area_definition(*input_file*)

Test data loaded with AreaDefinition.

test_data_with_rect_lon(*input_file*)

Test data loaded with AreaDefinition and user defined rectification longitude.

static test_data_with_swath_definition(*input_file*)

Test data loaded with SwathDefinition (default behaviour).

static test_lonslats(*input_file*)

Test reading of longitude and latitude data with SEVIRI L2 BUFR reader.

satpy.tests.reader_tests.test_seviri_l2_grib module

SEVIRI L2 GRIB-reader test package.

class satpy.tests.reader_tests.test_seviri_l2_grib.**Test_SeviriL2GribFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the SeviriL2GribFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*ec_*)

Set up the test by creating a mocked eccodes library.

test_data_reading(*da_*, *xr_*)

Test the reading of data from the product.

satpy.tests.reader_tests.test_slstr_l1b module

Module for testing the satpy.readers.nc_slstr module.

class satpy.tests.reader_tests.test_slstr_l1b.**TestSLSTRCalibration**(*methodName='runTest'*)

Bases: *TestSLSTRL1B*

Test the implementation of the calibration factors.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_cal_rad()

Test the radiance to reflectance converter.

test_radiance_calibration(*xr_*)

Test radiance calibration steps.

test_reflectance_calibration(*da_*, *xr_*)

Test reflectance calibration.

class satpy.tests.reader_tests.test_slstr_l1b.**TestSLSTRL1B**(*methodName='runTest'*)

Bases: TestCase

Common setup for SLSTR_L1B tests.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*xr_*)

Create a fake dataset using the given radiance data.

class satpy.tests.reader_tests.test_slstr_l1b.**TestSLSTRReader**(*methodName='runTest'*)

Bases: *TestSLSTRL1B*

Test various nc_slstr file handlers.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

class FakeSpl

Bases: object

Fake return function for SPL interpolation.

static ev(*foo_x*, *foo_y*)

Fake function to return interpolated data.

_classSetupFailed = False

_class_cleanups = []

test_instantiate(*bvs_*, *xr_*)

Test initialization of file handlers.

satpy.tests.reader_tests.test_slstr_l1b.**make_dataid**(***items*)

Make a data id.

satpy.tests.reader_tests.test_smos_l2_wind module

Module for testing the satpy.readers.smos_l2_wind module.

class satpy.tests.reader_tests.test_smos_l2_wind.**FakeNetCDF4FileHandlerSMOSL2WIND**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_smos_l2_wind.**TestSMOSL2WINDReader**(*methodName='runTest'*)

Bases: TestCase

Test SMOS L2 WINDReader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap NetCDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_adjust_lon()

Load adjust longitude dataset.

test_init()

Test basic initialization of this reader.

test_load_lat()

Load lat dataset.

test_load_lon()

Load lon dataset.

test_load_wind_speed()

Load wind_speed dataset.

test_roll_dataset()

Load roll of dataset along the lon coordinate.

yaml_file = 'smos_l2_wind.yaml'

satpy.tests.reader_tests.test_tropomi_l2 module

Module for testing the satpy.readers.tropomi_l2 module.

class satpy.tests.reader_tests.test_tropomi_l2.**FakeNetCDF4FileHandlerTL2**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

_convert_data_content_to_dataarrays(*file_content*)

Convert data content to xarray's dataarrays.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_tropomi_l2.**TestTROPOMIL2Reader**(*methodName='runTest'*)

Bases: TestCase

Test TROPOMI L2 Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap NetCDF4 file handler with our own fake handler.

tearDown()

Stop wrapping the NetCDF4 file handler.

test_init()

Test basic initialization of this reader.

test_load_bounds()

Load bounds dataset.

test_load_no2()

Load NO2 dataset.

test_load_so2()

Load SO2 dataset.

yaml_file = 'tropomi_l2.yaml'

satpy.tests.reader_tests.test_utils module

Testing of helper functions.

class satpy.tests.reader_tests.test_utils.**TestHelpers**(*methodName='runTest'*)

Bases: TestCase

Test the area helpers.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_apply_rad_correction()

Test radiance correction technique using user-supplied coefs.

test_generic_open_BZ2File(*bz2_mock*)

Test the generic_open method with bz2 filename input.

test_generic_open_FSFile_MemoryFileSystem()

Test the generic_open method with FSFile in MemoryFileSystem.

test_generic_open_filename(*open_mock*)

Test the generic_open method with filename (str).

test_geostationary_mask()

Test geostationary mask.

test_get_earth_radius()

Test earth radius computation.

test_get_geostationary_angle_extent()

Get max geostationary angles.

test_get_geostationary_bbox()

Get the geostationary bbox.

test_get_user_calibration_factors()

Test the retrieval of user-supplied calibration factors.

test_lonlat_from_geos()

Get lonlats from geos.

test_np2str()

Test the np2str function.

test_pro_reading_gets_unzipped_file(*fake_unzip_file*, *fake_remove*)

Test the bz2 file unzipping context manager.

test_reduce_mda()

Test metadata size reduction.

test_sub_area(*adef*)

Sub area slicing.

test_unzip_FSFile(*bz2_mock*)

Test the FSFile bz2 file unzipping techniques.

test_unzip_file(*mock_popen*, *mock_bz2*)

Test the bz2 file unzipping techniques.

class satpy.tests.reader_tests.test_utils.**TestSunEarthDistanceCorrection**

Bases: object

Tests for applying Sun-Earth distance correction to reflectance.

setup_method()

Create input / output arrays for the tests.

test_apply_sunearth_corr()

Test the correction of reflectances with sun-earth distance.

test_get_utc_time()

Test the retrieval of scene time from a dataset.

test_remove_sunearth_corr()

Test the removal of the sun-earth distance correction.

satpy.tests.reader_tests.test_utils.**test_generic_open_binary**(*tmp_path*, *data*, *filename*, *mode*)

Test the bz2 file unzipping context manager using dummy binary data.

satpy.tests.reader_tests.test_vaisala_gld360 module

Unittesting the Vaisala GLD360 reader.

class satpy.tests.reader_tests.test_vaisala_gld360.**TestVaisalaGLD360TextFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the VaisalaGLD360TextFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_vaisala_gld360()

Test basic functionality for vaisala file handler.

satpy.tests.reader_tests.test_vii_base_nc module

The vii_base_nc reader tests package.

class satpy.tests.reader_tests.test_vii_base_nc.**TestViiNCBaseFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the ViiNCBaseFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp(*pgi_*)

Set up the test.

tearDown()

Remove the previously created test file.

test_dataset(*po_*, *pi_*, *pc_*)

Test the execution of the get_dataset function.

test_file_reading()

Test the file product reading.

test_functions(*tpgi_*, *tpi_*)

Test the functions.

test_standardize_dims()

Test the standardize dims function.

satpy.tests.reader_tests.test_vii_l1b_nc module

The vii_l1b_nc reader tests package.

This version tests the readers for VII test data V2 as per PFS V4A.

class satpy.tests.reader_tests.test_vii_l1b_nc.**TestViiL1bNCFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the ViiL1bNCFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test.

tearDown()

Remove the previously created test file.

test_calibration_functions()

Test the calibration functions.

test_functions()

Test the functions.

satpy.tests.reader_tests.test_vii_l2_nc module

The vii_2_nc reader tests package.

class satpy.tests.reader_tests.test_vii_l2_nc.**TestViiL2NCFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the ViiL2NCFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test.

tearDown()

Remove the previously created test file.

test_functions()

Test the functions.

satpy.tests.reader_tests.test_vii_utils module

The vii_utils reader tests package.

class satpy.tests.reader_tests.test_vii_utils.**TestViiUtils**(*methodName='runTest'*)

Bases: TestCase

Test the vii_utils module.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_constants()

Test the constant values.

satpy.tests.reader_tests.test_vii_wv_nc module

The vii_l2_nc reader tests package for VII/METimage water vapour products.

class satpy.tests.reader_tests.test_vii_wv_nc.**TestViiL2NCFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the ViiL2NCFileHandler reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test.

tearDown()

Remove the previously created test file.

test_functions()

Test the functions.

satpy.tests.reader_tests.test_viirs_atms_utils module

Test common VIIRS/ATMS SDR reader functions.

satpy.tests.reader_tests.test_viirs_atms_utils.**test_get_file_units**(*caplog*)

Test get the file-units from the dataset info.

satpy.tests.reader_tests.test_viirs_atms_utils.**test_get_scale_factors_for_units_reflectances**(*caplog*)

Test get scale factors for units, when variable is supposed to be a reflectance.

satpy.tests.reader_tests.test_viirs_atms_utils.**test_get_scale_factors_for_units_tbs**(*caplog*)

Test get scale factors for units, when variable is supposed to be a brightness temperature.

satpy.tests.reader_tests.test_viirs_atms_utils.**test_get_scale_factors_for_units_unsupported_units**()

Test get scale factors for units, when units are not supported.

satpy.tests.reader_tests.test_viirs_compact module

Module for testing the satpy.readers.viirs_compact module.

class satpy.tests.reader_tests.test_viirs_compact.**TestCompact**

Bases: object

Test class for reading compact viirs format.

_dataset_iterator()

_setup_method(*fake_dnb_file*)

Create a fake file from scratch.

teardown_method()

Destroy.

test_distributed()

Check that distributed computations work.

test_get_dataset()

Retrieve datasets from a DNB file.

satpy.tests.reader_tests.test_viirs_compact.**fake_dnb**()

Create fake DNB content.

satpy.tests.reader_tests.test_viirs_compact.**fake_dnb_file**(*fake_dnb*, *tmp_path*)

Create an hdf5 file in viirs_compact format with DNB data in it.

satpy.tests.reader_tests.test_viirs_edr module

Module for testing the satpy.readers.viirs_l2_jrr module.

Note: This is adapted from the test_slstr_l2.py code.

class satpy.tests.reader_tests.test_viirs_edr.**TestVIIRSJRRReader**

Bases: object

Test the VIIRS JRR L2 reader.

test_availability_veg_idx(*data_file*, *exp_available*)

Test that vegetation indexes aren't available when they aren't present.

test_get_dataset_generic(*var_names*, *data_file*)

Test datasets from cloud height files.

test_get_dataset_surf_refl(*data_files*)

Test retrieval of datasets.

test_get_dataset_surf_refl_with_veg_idx(*data_files*, *filter_veg*)

Test retrieval of vegetation indices from surface reflectance files.

test_get_platformname(*surface_reflectance_file*, *filename_platform*, *exp_shortname*)

Test finding start and end times of granules.

satpy.tests.reader_tests.test_viirs_edr.**_array_checks**(*data_arr: xr.DataArray*, *dtype: npt.Dtype = <class 'numpy.float32'>*, *multiple_files: bool = False*) → None

satpy.tests.reader_tests.test_viirs_edr.**_check_continuous_data_arr**(*data_arr: DataArray*) → None

satpy.tests.reader_tests.test_viirs_edr.**_check_surf_refl_data_arr**(*data_arr: xr.DataArray*, *dtype: npt.DType = <class 'numpy.float32'>*, *multiple_files: bool = False*) → None

satpy.tests.reader_tests.test_viirs_edr.**_check_surf_refl_qf_data_arr**(*data_arr: DataArray*, *multiple_files: bool*) → None

satpy.tests.reader_tests.test_viirs_edr.**_check_vi_data_arr**(*data_arr: DataArray*, *is_filtered: bool*, *multiple_files: bool*) → None

satpy.tests.reader_tests.test_viirs_edr.**_create_continuous_variables**(*var_names: Iterable[str]*) → dict[str, DataArray]

satpy.tests.reader_tests.test_viirs_edr.**_create_fake_dataset**(*vars_dict: dict[str, DataArray]*) → Dataset

satpy.tests.reader_tests.test_viirs_edr.**_create_fake_file**(*tmp_path_factory: TempPathFactory*, *filename: str*, *data_arrs: dict[str, DataArray]*) → Path

satpy.tests.reader_tests.test_viirs_edr.**_create_lst_variables**() → dict[str, DataArray]

satpy.tests.reader_tests.test_viirs_edr.**_create_surf_refl_variables**() → dict[str, DataArray]

satpy.tests.reader_tests.test_viirs_edr.**_create_surface_reflectance_file**(*tmp_path_factory: TempPathFactory*, *start_time: datetime*, *include_veg_indices: bool = False*) → Path

satpy.tests.reader_tests.test_viirs_edr.**_create_veg_index_variables**() → dict[str, DataArray]

satpy.tests.reader_tests.test_viirs_edr.**_is_mband_res**(*data_arr: DataArray*) → bool

satpy.tests.reader_tests.test_viirs_edr.**_shared_metadata_checks**(*data_arr: DataArray*) → None

satpy.tests.reader_tests.test_viirs_edr.**aod_file**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake AOD VIIRs EDR file.

satpy.tests.reader_tests.test_viirs_edr.**cloud_height_file**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake CloudHeight VIIRS EDR file.

satpy.tests.reader_tests.test_viirs_edr.**lst_file**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake VLST EDR file.

satpy.tests.reader_tests.test_viirs_edr.**multiple_surface_reflectance_files**(*surface_reflectance_file*, *surface_reflectance_file2*) → list[Path]

Get two multiple surface reflectance files.

satpy.tests.reader_tests.test_viirs_edr.**multiple_surface_reflectance_files_with_veg_indices**(*surface_reflect*
sur-
face_reflectanc
→
list[Path]

Get two multiple surface reflectance files with vegetation indexes included.

satpy.tests.reader_tests.test_viirs_edr.**surface_reflectance_file**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake surface reflectance EDR file.

satpy.tests.reader_tests.test_viirs_edr.**surface_reflectance_file2**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake surface reflectance EDR file.

satpy.tests.reader_tests.test_viirs_edr.**surface_reflectance_with_veg_indices_file**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake surface reflectance EDR file with vegetation indexes included.

satpy.tests.reader_tests.test_viirs_edr.**surface_reflectance_with_veg_indices_file2**(*tmp_path_factory: TempPathFactory*) → Path

Generate fake surface reflectance EDR file with vegetation indexes included.

satpy.tests.reader_tests.test_viirs_edr.**test_available_datasets**(*aod_file*)

Test that available datasets doesn't claim non-filetype datasets.

For example, if a YAML-configured dataset's file type is not loaded then the available status is *None* and should remain *None*. This means no file type knows what to do with this dataset. If it is *False* then that means that a file type knows of the dataset, but that the variable is not available in the file. In the below test this isn't the case so the YAML-configured dataset should be provided once and have a *None* availability.

satpy.tests.reader_tests.test_viirs_edr_active_fires module

VIIRS Active Fires Tests.

This module implements tests for VIIRS Active Fires NetCDF and ASCII file readers.

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**FakeImgFiresNetCDF4FileHandler**(*filename, filename_info, filetype_info, auto_maskandsca xarray_kwargs=Non cache_var_size=0 cache_handle=Fa extra_file_content=*

Bases: *FakeNetCDF4FileHandler*

Swap in CDF4 file handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename, filename_info, filename_type*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**FakeImgFiresTextFileHandler**(*filename*, *file-name_info*, *file-type_info*, ***kwargs*)

Bases: *BaseFileHandler*

Fake file handler for text files at image resolution.

Get fake file content from 'get_test_content'.

get_test_content()

Create fake test file content.

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**FakeModFiresNetCDF4FileHandler**(*filename*, *file-name_info*, *file-type_info*, *auto_maskandsca xar-ray_kwargs=Non cache_var_size=(cache_handle=Fa ex-tra_file_content=*

Bases: *FakeNetCDF4FileHandler*

Swap in CDF4 file handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filename_type*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**FakeModFiresTextFileHandler**(*filename*, *file-name_info*, *file-type_info*, ***kwargs*)

Bases: *BaseFileHandler*

Fake file handler for text files at moderate resolution.

Get fake file content from 'get_test_content'.

get_test_content()

Create fake test file content.

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**TestImgVIIRSActiveFiresNetCDF4**(*methodName='ru*

Bases: TestCase

Test VIIRS Fires Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap CDF4 file handler with own fake file handler.

tearDown()

Stop wrapping the CDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_load_dataset()

Test loading all datasets.

yaml_file = 'viirs_edr_active_fires.yaml'

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**TestImgVIIRSActiveFiresText**(*methodName='runTe*

Bases: TestCase

Test VIIRS Fires Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap file handler with own fake file handler.

tearDown()

Stop wrapping the text file handler.

test_init(*mock_obj*)

Test basic init with no extra parameters.

test_load_dataset(*mock_obj*)

Test loading all datasets.

yaml_file = 'viirs_edr_active_fires.yaml'

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**TestModVIIRSActiveFiresNetCDF4**(*methodName='ru*

Bases: TestCase

Test VIIRS Fires Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap CDF4 file handler with own fake file handler.

tearDown()

Stop wrapping the CDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_load_dataset()

Test loading all datasets.

yaml_file = 'viirs_edr_active_fires.yaml'

class satpy.tests.reader_tests.test_viirs_edr_active_fires.**TestModVIIRSActiveFiresText**(*methodName='runTe*

Bases: TestCase

Test VIIRS Fires Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap file handler with own fake file handler.

tearDown()

Stop wrapping the text file handler.

test_init(*mock_obj*)

Test basic init with no extra parameters.

test_load_dataset(*csv_mock*)

Test loading all datasets.

yaml_file = 'viirs_edr_active_fires.yaml'

satpy.tests.reader_tests.test_viirs_edr_flood module

Tests for the VIIRS EDR Flood reader.

class satpy.tests.reader_tests.test_viirs_edr_flood.**FakeHDF4FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF4FileHandler*

Swap in HDF4 file handler.

Get fake file content from 'get_test_content'.

get_test_content(*filename*, *filename_info*, *filename_type*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_viirs_edr_flood.**TestVIIRSEDRFloodReader**(*methodName='runTest'*)

Bases: TestCase

Test VIIRS EDR Flood Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF4 file handler with own fake file handler.

tearDown()

Stop wrapping the HDF4 file handler.

test_init()

Test basic init with no extra parameters.

test_load_dataset()

Test loading all datasets from a full swath file.

test_load_dataset_aoi()

Test loading all datasets from an area of interest file.

yaml_file = 'viirs_edr_flood.yaml'

satpy.tests.reader_tests.test_viirs_l1b module

Module for testing the satpy.readers.viirs_l1b module.

class satpy.tests.reader_tests.test_viirs_l1b.**FakeNetCDF4FileHandlerDay**(*filename, filename_info, filetype_info, auto_maskandscale=False, xarray_kwargs=None, cache_var_size=0, cache_handle=False, extra_file_content=None*)

Bases: *FakeNetCDF4FileHandler*

Swap-in NetCDF4 File Handler.

Get fake file content from 'get_test_content'.

I_BANDS = ['I01', 'I02', 'I03', 'I04', 'I05']

I_BT_BANDS = ['I04', 'I05']

I_REFL_BANDS = ['I01', 'I02', 'I03']

M_BANDS = ['M01', 'M02', 'M03', 'M04', 'M05', 'M06', 'M07', 'M08', 'M09', 'M10', 'M11', 'M12', 'M13', 'M14', 'M15', 'M16']

M_BT_BANDS = ['M12', 'M13', 'M14', 'M15', 'M16']

M_REFL_BANDS = ['M01', 'M02', 'M03', 'M04', 'M05', 'M06', 'M07', 'M08', 'M09', 'M10', 'M11']

_fill_contents_with_default_data(*file_content*, *file_type*)

Fill file contents with default data.

static _set_dataset_specific_metadata(*file_content*)

Set dataset-specific metadata.

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_viirs_l1b.**FakeNetCDF4FileHandlerNight**(*filename*, *filename_info*, *filetype_info*, *auto_maskandscale=False*, *xarray_kwargs=None*, *cache_var_size=0*, *cache_handle=False*, *extra_file_content=None*)

Bases: *FakeNetCDF4FileHandlerDay*

Same as the day file handler, but some day-only bands are missing.

This matches what happens in real world files where reflectance bands are removed in night data to save space.

Get fake file content from 'get_test_content'.

I_BANDS = ['I04', 'I05']

M_BANDS = ['M12', 'M13', 'M14', 'M15', 'M16']

class satpy.tests.reader_tests.test_viirs_l1b.**TestVIIRSL1BReaderDay**

Bases: object

Test VIIRS L1B Reader.

fake_cls

alias of *FakeNetCDF4FileHandlerDay*

has_reflectance_bands = True

setup_method()

Wrap NetCDF4 file handler with our own fake handler.

teardown_method()

Stop wrapping the NetCDF4 file handler.

test_available_datasets_m_bands()

Test available datasets for M band files.

test_init()

Test basic init with no extra parameters.

test_load_dnb_angles()

Test loading all DNB angle datasets.

test_load_dnb_radiance()

Test loading the main DNB dataset.

test_load_every_m_band_bt()

Test loading all M band brightness temperatures.

test_load_every_m_band_rad()

Test loading all M bands as radiances.

test_load_every_m_band_refl()

Test loading all M band reflectances.

test_load_i_band_angles()

Test loading all M bands as radiances.

yaml_file = 'viirs_l1b.yaml'

class satpy.tests.reader_tests.test_viirs_l1b.**TestVIIRSL1BReaderDayNight**

Bases: *TestVIIRSL1BReaderDay*

Test VIIRS L1b with night data.

Night data files don't have reflectance bands in them.

fake_cls

alias of *FakeNetCDF4FileHandlerNight*

has_reflectance_bands = False

satpy.tests.reader_tests.test_viirs_sdr module

Module for testing the satpy.readers.viirs_sdr module.

class satpy.tests.reader_tests.test_viirs_sdr.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, *include_factors=True*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Create fake file handler.

static _add_basic_metadata_to_file_content(*file_content*, *filename_info*, *num_grans*)

_add_data_info_to_file_content(*file_content*, *filename*, *data_var_prefix*, *num_grans*)

static _add_geo_ref(*file_content*, *filename*)

static _add_geolocation_info_to_file_content(*file_content*, *filename*, *data_var_prefix*, *num_grans*)

_add_granule_specific_info_to_file_content(*file_content*, *dataset_group*, *num_granules*, *num_scans_per_granule*, *gran_group_prefix*)

static _convert_numpy_content_to_dataarray(*final_content*)

static _get_per_granule_lats()

static _get_per_granule_lons()

_num_scans_per_gran = [48]

_num_test_granules = 1

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

class satpy.tests.reader_tests.test_viirs_sdr.**FakeHDF5FileHandlerAggr**(*filename*, *filename_info*, *filetype_info*, *include_factors=True*)

Bases: *FakeHDF5FileHandler2*

Swap-in HDF5 File Handler with 4 VIIRS Granules per file.

Create fake file handler.

_num_scans_per_gran = [48, 48, 48, 48]

_num_test_granules = 4

class satpy.tests.reader_tests.test_viirs_sdr.**FakeShortHDF5FileHandlerAggr**(*filename*, *filename_info*, *filetype_info*, *include_factors=True*)

Bases: *FakeHDF5FileHandler2*

Fake file that has less scans than usual in a couple granules.

Create fake file handler.

_num_scans_per_gran = [47, 48, 47]

_num_test_granules = 3

class satpy.tests.reader_tests.test_viirs_sdr.**TestAggrVIIRSSDRReader**(*methodName='runTest'*)

Bases: TestCase

Test VIIRS SDR Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_bounding_box()

Test bounding box.

yaml_file = 'viirs_sdr.yaml'

class satpy.tests.reader_tests.test_viirs_sdr.**TestShortAggrVIIRSSDRReader**(*methodName='runTest'*)

Bases: TestCase

Test VIIRS SDR Reader with a file that has truncated granules.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_load_truncated_band()

Test loading a single truncated band.

yaml_file = 'viirs_sdr.yaml'

class satpy.tests.reader_tests.test_viirs_sdr.**TestVIIRSSDRReader**(*methodName='runTest'*)

Bases: TestCase

Test VIIRS SDR Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_assert_bt_properties(*data_arr*, *num_scans=16*, *with_area=True*)

_assert_dnb_radiance_properties(*data_arr*, *with_area=True*)

_assert_reflectance_properties(*data_arr*, *num_scans=16*, *with_area=True*)

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_init()

Test basic init with no extra parameters.

test_init_end_time_beyond()

Test basic init with end_time before the provided files.

test_init_start_end_time()

Test basic init with end_time before the provided files.

test_init_start_time_beyond()

Test basic init with start_time after the provided files.

test_init_start_time_is_nodate()

Test basic init with start_time being set to the no-date 1/1-1958.

test_load_all_i_bts()

Load all I band brightness temperatures.

test_load_all_i_radiances()

Load all I band radiances.

test_load_all_i_reflectances_provided_geo()

Load all I band reflectances with geo files provided.

test_load_all_m_bts()

Load all M band brightness temperatures.

test_load_all_m_radiances()

Load all M band radiances.

test_load_all_m_reflectances_find_geo()

Load all M band reflectances with geo files not specified but existing.

test_load_all_m_reflectances_no_geo()

Load all M band reflectances with no geo files provided.

test_load_all_m_reflectances_provided_geo()

Load all M band reflectances with geo files provided.

test_load_all_m_reflectances_use_nontc()

Load all M band reflectances but use non-TC geolocation.

test_load_all_m_reflectances_use_nontc2()

Load all M band reflectances but use non-TC geolocation because TC isn't available.

test_load_dnb()

Load DNB dataset.

test_load_dnb_no_factors()

Load DNB dataset with no provided scale factors.

test_load_dnb_sza_no_factors()

Load DNB solar zenith angle with no scaling factors.

The angles in VIIRS SDRs should never have scaling factors so we test it that way.

test_load_i_no_files()

Load I01 when only DNB files are provided.

yaml_file = 'viirs_sdr.yaml'

satpy.tests.reader_tests.test_viirs_sdr.**_touch_geo_file**(*prefix*)

satpy.tests.reader_tests.test_viirs_sdr.**touch_geo_files**(**prefixes*)

Create and then remove VIIRS SDR geolocation files.

satpy.tests.reader_tests.test_viirs_vgac_l1c_nc module

The viirs_vgac_l1b_nc reader tests package.

This version tests the readers for VIIIRS VGAC data preliminary version.

class satpy.tests.reader_tests.test_viirs_vgac_l1c_nc.**TestVGACREader**

Bases: object

Test the VGACFileHandler reader.

test_read_vgac(*nc_filename*)

Test reading reflectances and BT.

satpy.tests.reader_tests.test_viirs_vgac_l1c_nc.**nc_filename**(*tmp_path*)

Create an nc test data file and return its filename.

satpy.tests.reader_tests.test_virr_l1b module

Test for readers/virr_l1b.py.

class satpy.tests.reader_tests.test_virr_l1b.**FakeHDF5FileHandler2**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *FakeHDF5FileHandler*

Swap-in HDF5 File Handler.

Get fake file content from 'get_test_content'.

_make_file(*platform_id*, *geolocation_prefix*, *l1b_prefix*, *ECWN*, *Emissive_units*)

get_test_content(*filename*, *filename_info*, *filetype_info*)

Mimic reader input file content.

make_test_data(*dims*)

Create fake test data.

class satpy.tests.reader_tests.test_virr_l1b.**TestVIRRL1BReader**(*methodName='runTest'*)

Bases: TestCase

Test VIRR L1B Reader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_band_helper(*attributes*, *units*, *calibration*, *standard_name*, *file_type*, *band_index_size*, *resolution*)

_classSetupFailed = False

_class_cleanups = []

_fy3_helper(*platform_name*, *reader*, *Emissive_units*)

Load channels and test accurate metadata.

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_fy3b_file()

Test that FY3B files are recognized.

test_fy3c_file()

Test that FY3C files are recognized.

yaml_file = 'virr_l1b.yaml'

satpy.tests.reader_tests.utils module

Utilities for reader tests.

satpy.tests.reader_tests.utils.**_is_jit_method**(*obj*)

satpy.tests.reader_tests.utils.**default_attr_processor**(*root*, *attr*)

Do not change the attribute.

satpy.tests.reader_tests.utils.**fill_h5**(*root*, *contents*, *attr_processor=<function default_attr_processor>*)

Fill hdf5 file with the given contents.

Parameters

- **root** – hdf5 file rott
- **contents** – Contents to be written into the file
- **attr_processor** – A method for modifying attributes before they are written to the file.

satpy.tests.reader_tests.utils.**get_jit_methods**(*module*)

Get all jit-compiled methods in a module.

satpy.tests.reader_tests.utils.**skip_numba_unstable_if_missing**()

Determine if numba-based tests should be skipped during unstable CI tests.

If numba fails to import it could be because numba is not compatible with a newer version of numpy. This is very likely to happen in the unstable/experimental CI environment. This function returns `True` if numba-based tests should be skipped if `numba` could not be imported *and* we're in the unstable environment. We determine if we're in this CI environment by looking for the `UNSTABLE="1"` environment variable.

Module contents

The reader tests package.

satpy.tests.scene_tests package

Submodules

satpy.tests.scene_tests.test_conversions module

Unit tests for Scene conversion functionality.

class satpy.tests.scene_tests.test_conversions.**TestSceneConversions**

Bases: object

Test Scene conversion to geoviews, xarray, etc.

test_geoviews_basic_with_area()

Test converting a Scene to geoviews with an AreaDefinition.

test_geoviews_basic_with_swath()

Test converting a Scene to geoviews with a SwathDefinition.

test_to_xarray_dataset_with_empty_scene()

Test converting empty Scene to xarray dataset.

class satpy.tests.scene_tests.test_conversions.**TestSceneSerialization**

Bases: object

Test the Scene serialization.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_serialization_with_readers_and_data_arr()

Test that dask can serialize a Scene with readers.

class satpy.tests.scene_tests.test_conversions.**TestToXarrayConversion**

Bases: object

Test Scene.to_xarray() conversion.

multi_area_scn()

Define Scene with multiple area.

single_area_scn()

Define Scene with single area.

test_dataset_string_accepted(*single_area_scn*)

Test accept dataset string.

test_include_lonlats_false(*single_area_scn*)

Test exclude lonlats.

test_include_lonlats_true(*single_area_scn*)

Test include lonlats.

test_to_xarray_with_multiple_area_scene(*multi_area_scn*)

Test converting muiltple area Scene to xarray.

test_with_empty_scene()

Test converting empty Scene to xarray.

test_with_single_area_scene_type(*single_area_scn*)

Test converting single area Scene to xarray dataset.

test_wrong_dataset_key(*single_area_scn*)

Test raise error if unexisting dataset.

satpy.tests.scene_tests.test_data_access module

Unit tests for data access methods and properties of the Scene class.

class satpy.tests.scene_tests.test_data_access.**TestComputePersist**

Bases: object

Test methods that compute the internal data in some way.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_chunk_pass_through()

Test pass through of xarray chunk.

test_compute_pass_through()

Test pass through of xarray compute.

test_persist_pass_through()

Test pass through of xarray persist.

class satpy.tests.scene_tests.test_data_access.**TestDataAccessMethods**

Bases: object

Test the scene class.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_bad_setitem()

Test setting an item wrongly.

test_contains()

Test contains.

test_delitem()

Test deleting an item.

test_getitem()

Test __getitem__ with names only.

test_getitem_modifiers()

Test __getitem__ with names and modifiers.

test_getitem_slices()

Test __getitem__ with slices.

test_iter()

Test iteration over the scene.

test_iter_by_area_swath()

Test iterating by area on a swath.

test_sensor_names_added_datasets(*include_reader*, *added_sensor*, *exp_sensors*)

Test that Scene sensor_names handles contained sensors properly.

test_sensor_names_readers(*reader*, *filenames*, *exp_sensors*)

Test that Scene sensor_names handles different cases properly.

test_setitem()

Test setting an item.

class satpy.tests.scene_tests.test_data_access.**TestFinestCoarsestArea**

Bases: object

Test the Scene logic for finding the finest and coarsest area.

test_coarsest_finest_area_different_shape(*coarse_area*, *fine_area*)

Test 'coarsest_area' and 'finest_area' methods for upright areas.

test_coarsest_finest_area_same_shape(*area_def*, *shifted_area*)

Test that two areas with the same shape are consistently returned.

If two geometries (ex. two AreaDefinitions or two SwathDefinitions) have the same resolution (shape) but different coordinates, which one has the finer resolution would ultimately be determined by the semi-random ordering of the internal container of the Scene (a dict) if only pixel resolution was compared. This test makes sure that it is always the same object returned.

satpy.tests.scene_tests.test_data_access.**_create_coarest_finest_data_array**(*shape*, *area_def*, *attrs=None*)

satpy.tests.scene_tests.test_data_access.**_create_coarsest_finest_area_def**(*shape*, *extents*)

satpy.tests.scene_tests.test_data_access.**_create_coarsest_finest_swath_def**(*shape*, *extents*, *name_suffix*)

satpy.tests.scene_tests.test_init module

Unit tests for Scene creation.

class satpy.tests.scene_tests.test_init.**TestScene**

Bases: object

Test the scene class.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_create_multiple_reader_different_kwargs(*include_test_etc*)

Test passing different kwargs to different readers.

test_create_reader_instances_with_filenames()

Test creating a reader providing filenames.

test_create_reader_instances_with_reader()

Test createring a reader instance providing the reader name.

test_create_reader_instances_with_reader_kwargs()

Test creating a reader instance with reader kwargs.

test_init()

Test scene initialization.

test_init_alone()

Test simple initialization.

test_init_no_files()

Test that providing an empty list of filenames fails.

test_init_preserve_reader_kwargs()

Test that the initialization preserves the kwargs.

test_init_str_filename()

Test initializing with a single string as filenames.

test_init_with_empty_filenames()

Test initialization with empty filename list.

test_init_with_fsfile()

Test initialisation with FSFile objects.

test_start_end_times()

Test start and end times for a scene.

test_storage_options_from_reader_kwargs_no_options()

Test getting storage options from reader kwargs.

Case where there are no options given.

test_storage_options_from_reader_kwargs_per_reader()

Test getting storage options from reader kwargs.

Case where each reader have their own storage options.

test_storage_options_from_reader_kwargs_per_reader_and_global()

Test getting storage options from reader kwargs.

Case where each reader have their own storage options and there are global options to merge.

test_storage_options_from_reader_kwargs_single_dict(*reader_kwargs*)

Test getting storage options from reader kwargs.

Case where a single dict is given for all readers with some common storage options.

test_storage_options_from_reader_kwargs_single_dict_no_options()

Test getting storage options from reader kwargs for remote files.

Case where a single dict is given for all readers without storage options.

satpy.tests.scene_tests.test_load module

Unit tests for loading-related functionality in scene.py.

class satpy.tests.scene_tests.test_load.**TestBadLoading**

Bases: object

Test the Scene object's *.load* method with bad inputs.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_load_no_exist()

Test loading a dataset that doesn't exist.

test_load_str()

Test passing a string to Scene.load.

class satpy.tests.scene_tests.test_load.**TestLoadingComposites**

Bases: object

Test the Scene object's *.load* method for composites.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_load_comp11_and_23()

Test loading two composites that depend on similar wavelengths.

test_load_comp15()

Test loading a composite whose prerequisites can't be loaded.

Note that the prereq exists in the reader, but fails in loading.

test_load_comp17()

Test loading a composite that depends on a composite that won't load.

test_load_comp18()

Test loading a composite that depends on an incompatible area modified dataset.

test_load_comp18_2()

Test loading a composite that depends on an incompatible area modified dataset.

Specifically a modified dataset where the modifier has optional dependencies.

test_load_comp19()

Test loading a composite that shares a dep with a dependency.

More importantly test that loading a dependency that depends on the same dependency as this composite (a sibling dependency) and that sibling dependency includes a modifier. This test makes sure that the Node in the dependency tree is the exact same node.

test_load_comp8()

Test loading a composite that has a non-existent prereq.

test_load_dataset_after_composite()

Test load composite followed by other datasets.

test_load_dataset_after_composite2()

Test load complex composite followed by other datasets.

test_load_modified()

Test loading a modified dataset.

test_load_modified_with_load_kwarg()

Test loading a modified dataset using the `Scene.load` keyword argument.

test_load_multiple_comps()

Test loading multiple composites.

test_load_multiple_comps_separate()

Test loading multiple composites, one at a time.

test_load_multiple_modified()

Test loading multiple modified datasets.

test_load_multiple_resolutions()

Test loading a dataset has multiple resolutions available with different resolutions.

test_load_same_subcomposite()

Test loading a composite and one of it's subcomposites at the same time.

test_load_too_many()

Test dependency tree if too many reader keys match.

test_load_when_sensor_none_in_preloaded_dataarrays()

Test Scene loading when existing loaded arrays have sensor set to None.

Some readers or composites (ex. static images) don't have a sensor and developers choose to set it to *None*. This test makes sure this doesn't break loading.

test_modified_with_wl_dep()

Test modifying a dataset with a modifier with modified deps.

More importantly test that loading the modifiers dependency at the same time as the original modified dataset that the dependency tree nodes are unique and that DataIDs.

test_no_generate_comp10()

Test generating a composite after loading.

test_single_composite_loading(*comp_name*, *exp_id_or_name*)

Test that certain composites can be loaded individually.

class satpy.tests.scene_tests.test_load.**TestLoadingReaderDatasets**

Bases: object

Test the Scene object's *.load* method for datasets coming from a reader.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_load_ds1_load_twice()

Test loading one dataset with no loaded compositors.

test_load_ds1_no_comps()

Test loading one dataset with no loaded compositors.

test_load_ds1_unknown_modifier()

Test loading one dataset with no loaded compositors.

test_load_ds4_cal()

Test loading a dataset that has two calibration variations.

test_load_ds5_multiple_resolution_loads()

Test loading a dataset with multiple resolutions available as separate loads.

test_load_ds5_variations(*input_filenames*, *load_kwargs*, *exp_resolution*)

Test loading a dataset has multiple resolutions available.

test_load_ds6_wl()

Test loading a dataset by wavelength.

test_load_ds9_fail_load()

Test loading a dataset that will fail during load.

test_load_no_exist2()

Test loading a dataset that doesn't exist then another load.

class satpy.tests.scene_tests.test_load.**TestSceneAllAvailableDatasets**

Bases: object

Test the Scene's handling of various dependencies.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_all_dataset_names_no_readers()

Test all dataset names with no reader.

test_all_datasets_multiple_reader()

Test all datasets for multiple readers.

test_all_datasets_no_readers()

Test all datasets with no reader.

test_all_datasets_one_reader()

Test all datasets for one reader.

test_available_composite_ids_missing_available()

Test available_composite_ids when a composites dep is missing.

test_available_composites_known_versus_all()

Test available_composite_ids when some datasets aren't available.

test_available_comps_no_deps()

Test Scene available composites when composites don't have a dependency.

test_available_dataset_names_no_readers()

Test the available dataset names without a reader.

test_available_dataset_no_readers()

Test the available datasets without a reader.

test_available_datasets_one_reader()

Test the available datasets for one reader.

test_available_when_sensor_none_in_preloaded_dataarrays()

Test Scene available composites when existing loaded arrays have sensor set to None.

Some readers or composites (ex. static images) don't have a sensor and developers choose to set it to *None*. This test makes sure this doesn't break available composite IDs.

satpy.tests.scene_tests.test_load.**_data_array_none_sensor**(*name: str*) → DataArray

Create a DataArray with sensor set to None.

satpy.tests.scene_tests.test_load.**_scene_with_data_array_none_sensor()**

satpy.tests.scene_tests.test_resampling module

Unit tests for resampling and crop-related functionality in scene.py.

class satpy.tests.scene_tests.test_resampling.**TestSceneAggregation**

Bases: object

Test the scene's aggregate method.

_check_aggregation_results(*expected_aggregated_shape*, *scene1*, *scene2*, *x_size*, *y_size*)

static _create_test_data(*x_size*, *y_size*)

test_aggregate()

Test the aggregate method.

test_aggregate_with_boundary()

Test aggregation with boundary argument.

test_custom_aggregate()

Test the aggregate method with custom function.

class satpy.tests.scene_tests.test_resampling.**TestSceneCrop**

Bases: object

Test creating new Scenes by cropping an existing Scene.

test_crop()

Test the crop method.

test_crop_epsg_crs()

Test the crop method when source area uses an EPSG code.

test_crop_rgb()

Test the crop method on multi-dimensional data.

class satpy.tests.scene_tests.test_resampling.**TestSceneResampling**

Bases: object

Test resampling a Scene to another Scene object.

_fake_resample_dataset(*dataset*, *dest_area*, ***kwargs*)

Return copy of dataset pretending it was resampled.

_fake_resample_dataset_force_20x20(*dataset*, *dest_area*, ***kwargs*)

Return copy of dataset pretending it was resampled to (20, 20) shape.

pytestmark = [Mark(name='usefixtures', args=('include_test_etc',), kwargs={})]

test_comp_loading_after_resampling_existing_sensor()

Test requesting a composite after resampling.

test_comp_loading_after_resampling_new_sensor()

Test requesting a composite after resampling when the sensor composites weren't loaded before.

test_comp_loading_multisensor_composite_created_user()

Test that multisensor composite can be created manually.

Test that if the user has created datasets "manually", that multi-sensor composites provided can still be read.

test_comps_need_resampling_optional_mod_deps()

Test that a composite with complex dependencies.

This is specifically testing the case where a compositor depends on multiple resolution prerequisites which themselves are composites. These sub-composites depend on data with a modifier that only has optional dependencies. This is a very specific use case and is the simplest way to present the problem (so far).

The general issue is that the Scene loading creates the "ds13" dataset which already has one modifier on it. The "comp27" composite requires resampling so its 4 prerequisites + the requested "ds13" (from the reader which includes mod1 modifier) remain. If the DependencyTree is not copied properly in this situation then the new Scene object will have the composite dependencies without resolution in its dep tree, but have the DataIDs with the resolution in the dataset dictionary. This all results in the Scene trying to regenerate composite dependencies that aren't needed which fail.

test_no_generate_comp10(*rs*)

Test generating a composite after loading.

test_resample_ancillary()

Test that the Scene reducing data does not affect final output.

test_resample_multi_ancillary()

Test that multiple ancillary variables are retained after resampling.

This test corresponds to GH#2329

test_resample_reduce_data()

Test that the Scene reducing data does not affect final output.

test_resample_reduce_data_toggle(*rs*)

Test that the Scene can be reduced or not reduced during resampling.

test_resample_scene_copy(*rs*, *datasets*)

Test that the Scene is properly copied during resampling.

The Scene that is created as a copy of the original Scene should not be able to affect the original Scene object.

test_resample_scene_preserves_requested_dependencies(*rs*)

Test that the Scene is properly copied during resampling.

The Scene that is created as a copy of the original Scene should not be able to affect the original Scene object.

satpy.tests.scene_tests.test_saving module

Unit tests for saving-related functionality in scene.py.

class satpy.tests.scene_tests.test_saving.**TestSceneSaving**

Bases: object

Test the Scene's saving method.

test_save_dataset_default(*tmp_path*)

Save a dataset using 'save_dataset'.

test_save_datasets_bad_writer(*tmp_path*)

Save a dataset using 'save_datasets' and a bad writer.

test_save_datasets_by_ext(*tmp_path*)

Save a dataset using 'save_datasets' with 'filename'.

test_save_datasets_default(*tmp_path*)

Save a dataset using 'save_datasets'.

test_save_datasets_missing_wishlist(*tmp_path*)

Calling 'save_datasets' with no valid datasets.

Module contents

Tests of the Scene class.

satpy.tests.writer_tests package

Submodules

satpy.tests.writer_tests.test_awips_tiled module

Tests for the AWIPS Tiled writer.

class satpy.tests.writer_tests.test_awips_tiled.**TestAWIPSTiledWriter**

Bases: object

Test basic functionality of AWIPS Tiled writer.

static _get_glm_glob_filename(*extra_kwargs*)

test_basic_lettered_tiles(*tmp_path*)

Test creating a lettered grid.

test_basic_lettered_tiles_diff_projection(*tmp_path*)

Test creating a lettered grid from data with differing projection..

test_basic_numbered_1_tile(*extra_attrs*, *expected_filename*, *use_save_dataset*, *caplog*, *tmp_path*)

Test creating a single numbered tile.

test_basic_numbered_tiles(*tile_count*, *tile_size*, *tmp_path*)

Test creating a multiple numbered tiles.

test_basic_numbered_tiles_rgb(*tmp_path*)

Test creating a multiple numbered tiles with RGB.

test_init(*tmp_path*)

Test basic init method of writer.

test_lettered_tiles_bad_filename(*tmp_path*)

Test creating a lettered grid with a bad filename.

test_lettered_tiles_no_fit(*tmp_path*)

Test creating a lettered grid with no data overlapping the grid.

test_lettered_tiles_no_valid_data(*tmp_path*)

Test creating a lettered grid with no valid data.

test_lettered_tiles_sector_ref(*tmp_path*)

Test creating a lettered grid using the sector as reference.

test_lettered_tiles_update_existing(*tmp_path*)

Test updating lettered tiles with additional data.

test_multivar_numbered_tiles_glm(*sector*, *extra_kwargs*, *tmp_path*)

Test creating a tiles with multiple variables.

test_units_length_warning(*tmp_path*)

Test long 'units' warnings are raised.

satpy.tests.writer_tests.test_awips_tiled.**_check_production_location**(*ds*)

satpy.tests.writer_tests.test_awips_tiled.**_check_required_common_attributes**(*ds*)

Check common properties of the created AWIPS tiles for validity.

satpy.tests.writer_tests.test_awips_tiled.**_check_scaled_x_coordinate_variable**(*ds*, *masked_ds*)

satpy.tests.writer_tests.test_awips_tiled.**_check_scaled_y_coordinate_variable**(*ds*, *masked_ds*)

satpy.tests.writer_tests.test_awips_tiled.**_get_test_area**(*shape=(200, 100)*, *crs=None*, *extents=None*)

satpy.tests.writer_tests.test_awips_tiled.**_get_test_data**(*shape=(200, 100)*, *chunks=50*)

satpy.tests.writer_tests.test_awips_tiled.**_get_test_lcc_data**(*dask_arr*, *area_def*, *extra_attrs=None*)

satpy.tests.writer_tests.test_awips_tiled.**check_required_properties**(*unmasked_ds*, *masked_ds*)

Check various aspects of coordinates and attributes for correctness.

satpy.tests.writer_tests.test_cf module

Tests for the CF writer.

class satpy.tests.writer_tests.test_cf.**TempFile**(*suffix='.nc'*)

Bases: object

A temporary filename class.

Initialize.

class satpy.tests.writer_tests.test_cf.**TestCFWriter**

Bases: object

Test case for CF writer.

test_ancillary_variables()

Test ancillary_variables cited each other.

test_bounds()

Test setting time bounds.

test_bounds_minimum()

Test minimum bounds.

test_bounds_missing_time_info()

Test time bounds generation in case of missing time.

test_global_attr_default_history_and_Conventions()

Test saving global attributes history and Conventions.

test_global_attr_history_and_Conventions()

Test saving global attributes history and Conventions.

test_groups()

Test creating a file with groups.

test_header_attrs()

Check global attributes are set.

test_init()

Test initializing the CFWriter class.

test_load_module_with_old_pyproj()

Test that cf_writer can still be loaded with pyproj 1.9.6.

test_save_array()

Test saving an array to netcdf/cf.

test_save_array_coords()

Test saving array with coordinates.

test_save_dataset_a_digit()

Test saving an array to netcdf/cf where dataset name starting with a digit.

test_save_dataset_a_digit_no_prefix_include_attr()

Test saving an array to netcdf/cf dataset name starting with a digit with no prefix include orig name.

test_save_dataset_a_digit_prefix()

Test saving an array to netcdf/cf where dataset name starting with a digit with prefix.

test_save_dataset_a_digit_prefix_include_attr()

Test saving an array to netcdf/cf where dataset name starting with a digit with prefix include orig name.

test_single_time_value()

Test setting a single time value.

test_time_coordinate_on_a_swath()

Test that time dimension is not added on swath data with time already as a coordinate.

test_unlimited_dims_kwarg()

Test specification of unlimited dimensions.

class satpy.tests.writer_tests.test_cf.**TestEncodingAttribute**

Bases: *TestNetcdfEncodingKwargs*

Test CF writer with 'encoding' dataset attribute.

scene_with_encoding(*scene*, *encoding*)

Create scene with a dataset providing the 'encoding' attribute.

test_encoding_attribute(*scene_with_encoding*, *filename*, *expected*)

Test 'encoding' dataset attribute.

class satpy.tests.writer_tests.test_cf.**TestNetcdfEncodingKwargs**

Bases: object

Test netCDF compression encodings.

_assert_encoding_as_expected(*filename*, *expected*)

complevel_exp(*compression_on*)

Get expected compression level.

compression_on(*request*)

Get compression options.

encoding(*compression_on*)

Get encoding.

expected(*complevel_exp*)

Get expectated file contents.

filename(*tmp_path*)

Get output filename.

scene()

Create a fake scene.

test_encoding_kwarg(*scene*, *encoding*, *filename*, *expected*)

Test 'encoding' keyword argument.

test_no_warning_if_backends_match(*scene*, *filename*, *monkeypatch*)

Make sure no warning is issued if backends match.

test_warning_if_backends_dont_match(*scene*, *filename*, *monkeypatch*)

Test warning if backends don't match.

satpy.tests.writer_tests.test_cf.**_get_compression_params**(*complevel*)

satpy.tests.writer_tests.test_cf.**_should_use_compression_keyword**()

satpy.tests.writer_tests.test_geotiff module

Tests for the geotiff writer.

class satpy.tests.writer_tests.test_geotiff.**TestGeoTIFFWriter**

Bases: object

Test the GeoTIFF Writer class.

test_colormap_write(*tmp_path*)

Test writing an image with a colormap.

test_dtype_for_enhance_false(*tmp_path*)

Test that dtype of dataset is used if parameters enhance=False and dtype=None.

test_dtype_for_enhance_false_and_given_dtype(*tmp_path*)

Test that dtype of dataset is used if enhance=False and dtype=uint8.

test_fill_value_from_config(*tmp_path*)

Test fill_value coming from the writer config.

test_float_write(*tmp_path*)

Test that geotiffs can be written as floats.

NOTE: Does not actually check that the output is floats.

test_float_write_with_unit_conversion(*tmp_path*)

Test that geotiffs can be written as floats and convert units.

test_init()

Test creating the writer with no arguments.

test_scale_offset(*input_func*, *save_kwargs*, *tmp_path*)

Test tags being added.

test_simple_delayed_write(*tmp_path*)

Test writing can be delayed.

test_simple_write(*input_func*, *tmp_path*)

Test basic writer operation.

test_tags(*tmp_path*)

Test tags being added.

test_tiled_value_from_config(*tmp_path*)

Test tiled value coming from the writer config.

satpy.tests.writer_tests.test_geotiff.**_get_test_datasets_2d**()

Create a single 2D test dataset.

satpy.tests.writer_tests.test_geotiff.**_get_test_datasets_2d_nonlinear_enhancement**()

satpy.tests.writer_tests.test_geotiff.**_get_test_datasets_3d**()

Create a single 3D test dataset.

satpy.tests.writer_tests.test_mitiff module

Tests for the mitiff writer.

Based on the test for geotiff writer

class satpy.tests.writer_tests.test_mitiff.**TestMITIFFWriter**(*methodName='runTest'*)

Bases: TestCase

Test the MITIFF Writer class.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_test_dataset(*bands=3*)

Create a single test dataset.

_get_test_dataset_calibration(*bands=6*)

Create a single test dataset.

_get_test_dataset_calibration_one_dataset(*bands=1*)

Create a single test dataset.

_get_test_dataset_three_bands_prereq(*bands=3*)

Create a single test dataset.

_get_test_dataset_three_bands_two_prereq(*bands=3*)

Create a single test dataset.

_get_test_dataset_with_bad_values(*bands=3*)

Create a single test dataset.

_get_test_datasets()

Create a datasets list.

_get_test_datasets_sensor_set()

Create a datasets list.

_get_test_one_dataset()

Create a single test dataset.

_get_test_one_dataset_sensor_set()

Create a single test dataset.

_imagedescription_from_mitiff(*filename*)

_read_back_mitiff_and_check(*filename*, *expected*, *test_shape=(100, 200)*)

setUp()

Create temporary directory to save files to.

tearDown()

Remove the temporary directory created for a test.

test_convert_proj4_string()

Test conversion of geolocations.

test_get_test_dataset_three_bands_prereq()

Test basic writer operation with 3 bands with DataQuery prerequisites with missing name.

test_init()

Test creating the writer with no arguments.

test_save_dataset_palette()

Test writer operation as palette.

test_save_dataset_with_bad_value()

Test writer operation with bad values.

test_save_dataset_with_calibration()

Test writer operation with calibration.

test_save_dataset_with_calibration_error_one_dataset()

Test saving if mitiff as dataset with only one channel with invalid calibration.

test_save_dataset_with_calibration_one_dataset()

Test saving if mitiff as dataset with only one channel.

test_save_dataset_with_missing_palette()

Test saving if mitiff missing palette.

test_save_datasets()

Test basic writer operation save_datasets.

test_save_datasets_sensor_set()

Test basic writer operation save_datasets.

test_save_one_dataset()

Test basic writer operation with one dataset ie. no bands.

test_save_one_dataset_sensor_set()

Test basic writer operation with one dataset ie. no bands.

test_simple_write()

Test basic writer operation.

test_simple_write_two_bands()

Test basic writer operation with 3 bands from 2 prerequisites.

satpy.tests.writer_tests.test_ninjogeotiff module

Tests for writing GeoTIFF files with NinJoTIFF tags.

satpy.tests.writer_tests.test_ninjogeotiff.**_get_fake_da**(*lo*, *hi*, *shp*, *dtype='f4'*)

Generate dask array with synthetic data.

This is more or less a 2d linspace: it'll return a 2-d dask array of shape `shp`, lowest value is `lo`, highest value is `hi`.

satpy.tests.writer_tests.test_ninjogeotiff.**_patch_datetime_now**(*monkeypatch*)

Get a fake datetime.datetime.now().

satpy.tests.writer_tests.test_ninjogeotiff.**ntg1**(*test_image_small_mid_atlantic_L*)

Create instance of NinJoTagGenerator class.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg2**(*test_image_large_asia_RGB*)

Create instance of NinJoTagGenerator class.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg3**(*test_image_small_arctic_P*)

Create instance of NinJoTagGenerator class.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg_cmyk**(*test_image_cmyk_antarctic*)

Create NinJoTagGenerator instance with CMYK image.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg_latlon**(*test_image_latlon*)

Create NinJoTagGenerator with latlon-area image.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg_no_fill_value**(*test_image_small_mid_atlantic_L*)

Create instance of NinJoTagGenerator class.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg_northpole**(*test_image_northpole*)

Create NinJoTagGenerator with north pole image.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg_rgba**(*test_image_rgba_merc*)

Create NinJoTagGenerator instance with RGBA image.

satpy.tests.writer_tests.test_ninjogeotiff.**ntg_weird**(*test_image_weird*)

Create NinJoTagGenerator instance with weird image.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_epsg4326**()

Test with EPSG4326 (latlong) area, which has no CRS coordinate operation.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_merc**()

Create a mercator area.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_northpole**()

Create a 20x10 test area centered exactly on the north pole.

This has no well-defined central meridian so needs separate testing.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_small_eqc_wgs84**()

Create 50x100 test equirectangular area centered on (50, 90), wgs84.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_tiny_antarctic**()

Create a 20x10 test stereographic area centered near the south pole, wgs84.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_tiny_eqc_sphere**()

Create 10x00 test equirectangular area centered on (40, -30), spherical geoid, m.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_tiny_stereographic_wgs84**()

Create a 20x10 test stereographic area centered near the north pole, wgs84.

satpy.tests.writer_tests.test_ninjogeotiff.**test_area_weird**()

Create a weird area (interrupted goode homolosine) to test error handling.

satpy.tests.writer_tests.test_ninjogeotiff.**test_calc_single_tag_by_name**(*ntg1*, *ntg2*, *ntg3*)

Test calculating single tag from dataset.

satpy.tests.writer_tests.test_ninjogeotiff.**test_create_unknown_tags**(*test_image_small_arctic_P*)

Test that unknown tags raise ValueError.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_all_tags**(*ntg1*, *ntg3*, *ntg_latlon*, *ntg_northpole*, *caplog*)

Test getting all tags from dataset.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_central_meridian**(*ntg1*, *ntg2*, *ntg3*, *ntg_latlon*, *ntg_northpole*)

Test calculating the central meridian.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_color_depth**(*ntg1*, *ntg2*, *ntg3*, *ntg_weird*, *ntg_rgba*, *ntg_cmyk*)

Test extracting the color depth.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_creation_date_id**(*ntg1*, *ntg2*, *ntg3*)

Test getting the creation date ID.

This is the time at which the file was created.

This test believes it is run at 2033-5-18 05:33:20Z.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_date_id**(*ntg1*, *ntg2*, *ntg3*)

Test getting the date ID.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_earth_radius_large**(*ntg1*, *ntg2*, *ntg3*)

Test getting the Earth semi-major axis.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_earth_radius_small**(*ntg1*, *ntg2*, *ntg3*)

Test getting the Earth semi-minor axis.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_filename**(*ntg1*, *ntg2*, *ntg3*)

Test getting the filename.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_max_gray_value_L**(*ntg1*)

Test getting max gray value for mode L.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_max_gray_value_P**(*ntg3*)

Test getting max gray value for mode P.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_max_gray_value_RGB**(*ntg2*)

Test max gray value for RGB.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_meridian_east**(*ntg1*, *ntg2*, *ntg3*)

Test getting east meridian.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_meridian_west**(*ntg1*, *ntg2*, *ntg3*)

Test getting west meridian.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_min_gray_value_L**(*ntg1*)

Test getting min gray value for mode L.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_min_gray_value_P**(*ntg3*)

Test getting min gray value for mode P.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_min_gray_value_RGB**(*ntg2*)

Test getting min gray value for RGB.

Note that min/max gray value is mandatory in NinJo even for RGBs?

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_projection**(*ntg1*, *ntg2*, *ntg3*, *ntg_weird*, *ntg_rgba*, *ntg_cmyk*, *ntg_latlon*)

Test getting projection string.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_ref_lat_1**(*ntg1*, *ntg2*, *ntg3*, *ntg_weird*, *ntg_latlon*)

Test getting reference latitude 1.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_ref_lat_2**(*ntg1*, *ntg2*, *ntg3*)

Test getting reference latitude 2.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_transparent_pixel**(*ntg1*, *ntg2*, *ntg3*, *ntg_no_fill_value*)

Test getting fill value.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_xmax**(*ntg1*, *ntg2*, *ntg3*)

Test getting maximum x.

satpy.tests.writer_tests.test_ninjogeotiff.**test_get_ymax**(*ntg1*, *ntg2*, *ntg3*)

Test getting maximum y.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_cmyk_antarctic**(*test_area_tiny_antarctic*)

Get a small test image in mode CMYK on south pole.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_large_asia_RGB**(*test_area_small_eqc_wgs84*)

Get a large-ish test image in mode RGB, over Asia.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_latlon**(*test_area_epsg4326*)

Get image with latlon areadefinition.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_northpole**(*test_area_northpole*)

Test image with area exactly on northpole.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_rgba_merc**(*test_area_merc*)

Get a small test image in mode RGBA and mercator.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_small_arctic_P**(*test_area_tiny_stereographic_wgs84*)

Get a small-ish test image in mode P, over Arctic.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_small_mid_atlantic_K_L**(*test_area_tiny_eqc_sphere*)

Get a small test image in units K, mode L, over Atlantic.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_small_mid_atlantic_L**(*test_area_tiny_eqc_sphere*)

Get a small test image in mode L, over Atlantic.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_small_mid_atlantic_L_no_quantity**(*test_area_tiny_eqc_s*

Get a small test image, mode L, over Atlantic, with non-quantitywvalues.

This could be the case, for example, for vis_with_night_ir.

satpy.tests.writer_tests.test_ninjogeotiff.**test_image_weird**(*test_area_weird*)

Get a small image with some weird properties to test error handling.

satpy.tests.writer_tests.test_ninjogeotiff.**test_str_ids**(*test_image_small_arctic_P*)

Test that channel and satellit IDs can be str.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_file**(*test_image_small_mid_atlantic_L*, *tmp_path*)

Test that it writes a GeoTIFF with the appropriate NinJo-tags.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_file_LA**(*test_image_latlon*, *tmp_path*)

Test writing and reading LA image.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_file_P**(*test_image_small_arctic_P*, *tmp_path*)

Test writing and reading P image.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_file_RGB**(*test_image_large_asia_RGB*, *tmp_path*)

Test writing and reading RGB.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_file_units**(*test_image_small_mid_atlantic_K_L*, *tmp_path*, *caplog*)

Test that it writes a GeoTIFF with the appropriate NinJo-tags and units.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_no_quantity**(*test_image_small_mid_atlantic_L_no*, *tmp_path*, *unit*)

Test that no scale/offset written if no valid units present.

satpy.tests.writer_tests.test_ninjogeotiff.**test_write_and_read_via_scene**(*test_image_small_mid_atlantic_L*, *tmp_path*)

Test that all attributes are written also when writing from scene.

It appears that `Satpy.Scene.save_dataset()` does not pass the filename to the writer. Test that filename is still written to header when saving this way (the regular way).

satpy.tests.writer_tests.test_ninjotiff module

Tests for the NinJoTIFF writer.

class satpy.tests.writer_tests.test_ninjotiff.**FakeImage**(*data*, *mode*)

Bases: object

Fake image.

Init fake image.

get_scaling_from_history()

Return dummy scale and offset.

class satpy.tests.writer_tests.test_ninjotiff.**TestNinjoTIFFWriter**(*methodName='runTest'*)

Bases: TestCase

The ninjo tiff writer tests.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_P_image_is_uint8(*iwsi*, *save_dataset*)

Test that a P-mode image is converted to uint8s.

test_convert_units_other()

Test that other unit conversions are not implemented.

test_convert_units_self()

Test that unit conversion to themselves do nothing.

test_convert_units_temp()

Test that temperature unit conversions works as expected.

test_dataset(*iwsd*)

Test saving a dataset.

test_dataset_skip_unit_conversion(*iwsd*)

Test saving a dataset without unit conversion.

test_image(*iwsi*, *save_dataset*)

Test saving an image.

test_init()

Test the init.

satpy.tests.writer_tests.test_simple_image module

Tests for the simple image writer.

class satpy.tests.writer_tests.test_simple_image.**TestPillowWriter**(*methodName='runTest'*)

Bases: TestCase

Test Pillow/PIL writer.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

static _get_test_datasets()

Create DataArray for testing.

setUp()

Create temporary directory to save files to.

tearDown()

Remove the temporary directory created for a test.

test_init()

Test creating the default writer.

test_simple_delayed_write()

Test writing datasets with delayed computation.

test_simple_write()

Test writing datasets with default behavior.

satpy.tests.writer_tests.test_utils module

Tests for writer utilities.

class satpy.tests.writer_tests.test_utils.**WriterUtilsTest**(*methodName='runTest'*)

Bases: TestCase

Test various writer utilities.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_flatten_dict()

Test dictionary flattening.

Module contents

The writer tests package.

Submodules

satpy.tests.conftest module

Shared preparation and utilities for testing.

This module is executed automatically by pytest.

`satpy.tests.conftest.`**`_clear_function_caches`**`()`

Clear out global function-level caches that may cause conflicts between tests.

`satpy.tests.conftest.`**`_reset_satpy_config`**`(`*tmpdir*`)`

Set satpy config to logical defaults for tests.

`satpy.tests.conftest.`**`include_test_etc`**`()`

Tell Satpy to use the config 'etc' directory from the tests directory.

satpy.tests.test_cf_roundtrip module

Test roundripping the cf writer and reader.

`satpy.tests.test_cf_roundtrip.`**`test_cf_roundtrip`**`(`*fake_dnb_file*, *tmp_path*`)`

Test the cf writing reading cycle.

satpy.tests.test_composites module

Tests for compositors in composites/__init__.py.

class `satpy.tests.test_composites.`**`TestAddBands`**`(`*methodName='runTest'*`)`

Bases: `TestCase`

Test case for the *add_bands* function.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

`_classSetupFailed = False`

`_class_cleanups = []`

`test_add_bands_l_rgb``()`

Test adding bands.

`test_add_bands_l_rgba``()`

Test adding bands.

`test_add_bands_la_rgb``()`

Test adding bands.

test_add_bands_p_l()

Test adding bands.

test_add_bands_rgb_rbga()

Test adding bands.

class satpy.tests.test_composites.**TestBackgroundCompositor**

Bases: object

Test case for the background compositor.

classmethod setup_class()

Create shared input data arrays.

test_call(*foreground_bands*, *background_bands*, *exp_bands*, *exp_result*)

Test the background compositing.

test_multiple_sensors()

Test the background compositing from multiple sensor data.

class satpy.tests.test_composites.**TestCategoricalDataCompositor**(*methodName='runTest'*)

Bases: TestCase

Test composiotor for recategorization of categorical data.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data.

test_basic_recategorization()

Test general functionality of compositor incl. attributes.

test_too_many_datasets()

Test that ValueError is raised if more than one dataset is provided.

class satpy.tests.test_composites.**TestCloudCompositorCommonMask**

Bases: object

Test the CloudCompositorCommonMask.

setup_method()

Set up the test case.

test_bad_call()

Test the CloudCompositorCommonMask without mask.

test_call_dask()

Test the CloudCompositorCommonMask with dask.

test_call_numpy()

Test the CloudCompositorCommonMask with numpy.

class satpy.tests.test_composites.**TestCloudCompositorWithoutCloudfree**

Bases: object

Test the CloudCompositorWithoutCloudfree.

setup_method()

Set up the test case.

test_bad_indata()

Test the CloudCompositorWithoutCloudfree composite generation without status.

test_call_bad_optical_conditions()

Test the CloudCompositorWithoutCloudfree composite generation.

test_call_dask_with_invalid_value_in_status()

Test the CloudCompositorWithoutCloudfree composite generation.

test_call_numpy_with_invalid_value_in_status()

Test the CloudCompositorWithoutCloudfree composite generation.

class satpy.tests.test_composites.**TestColorizeCompositor**(*methodName='runTest'*)

Bases: TestCase

Test the ColorizeCompositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_colorize_no_fill()

Test colorizing.

test_colorize_with_interpolation()

Test colorizing with interpolation.

class satpy.tests.test_composites.**TestColormapCompositor**(*methodName='runTest'*)

Bases: TestCase

Test the ColormapCompositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_build_colormap_with_int_data_and_with_meanings()

Test colormap building.

test_build_colormap_with_int_data_and_without_meanings()

Test colormap building.

class satpy.tests.test_composites.**TestDayNightCompositor**(*methodName='runTest'*)

Bases: TestCase

Test DayNightCompositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data.

test_day_only_area_with_alpha()

Test compositor with day portion with alpha_band when SZA data is not provided.

test_day_only_area_with_alpha_and_missing_data()

Test compositor with day portion with alpha_band when SZA data is not provided and there is missing data.

test_day_only_area_without_alpha()

Test compositor with day portion without alpha_band when SZA data is not provided.

test_day_only_sza_with_alpha()

Test compositor with day portion with alpha band when SZA data is included.

test_day_only_sza_without_alpha()

Test compositor with day portion without alpha band when SZA data is included.

test_daynight_area()

Test compositor both day and night portions when SZA data is not provided.

test_daynight_sza()

Test compositor with both day and night portions when SZA data is included.

test_night_only_area_with_alpha()

Test compositor with night portion with alpha band when SZA data is not provided.

test_night_only_area_without_alpha()

Test compositor with night portion without alpha band when SZA data is not provided.

test_night_only_sza_with_alpha()

Test compositor with night portion with alpha band when SZA data is included.

test_night_only_sza_without_alpha()

Test compositor with night portion without alpha band when SZA data is included.

class satpy.tests.test_composites.**TestDifferenceCompositor**(*methodName='runTest'*)

Bases: TestCase

Test case for the difference compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data.

test_bad_areas_diff()

Test that a difference where resolutions are different fails.

test_basic_diff()

Test that a basic difference composite works.

class satpy.tests.test_composites.**TestEnhance2Dataset**(*methodName='runTest'*)

Bases: TestCase

Test the enhance2dataset utility.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_enhance_l(*get_enhanced_image*)

Test enhancing a paletted dataset in P mode.

test_enhance_p(*get_enhanced_image*)

Test enhancing a paletted dataset in P mode.

test_enhance_p_to_rgb(*get_enhanced_image*)

Test enhancing a paletted dataset in RGB mode.

test_enhance_p_to_rgba(*get_enhanced_image*)

Test enhancing a paletted dataset in RGBA mode.

class satpy.tests.test_composites.**TestFillingCompositor**(*methodName='runTest'*)

Bases: TestCase

Test case for the filling compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fill()

Test filling.

class satpy.tests.test_composites.**TestGenericCompositor**(*methodName='runTest'*)

Bases: TestCase

Test generic compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data.

test_call()

Test calling generic compositor.

test_call_with_mock(*match_data_arrays*, *check_times*, *combine_metadata*, *get_sensors*)

Test calling generic compositor.

test_concat_datasets()

Test concatenation of datasets.

test_deprecation_warning()

Test deprecation warning for dcprecated composite recipes.

test_get_sensors()

Test getting sensors from the dataset attributes.

test_masking()

Test masking in generic compositor.

class satpy.tests.test_composites.**TestInferMode**(*methodName='runTest'*)

Bases: TestCase

Test the infer_mode utility.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_band_size_is_used()

Test that the band size is used.

test_bands_coords_is_used()

Test that the *bands* coord is used.

test_mode_is_used()

Test that the *mode* attribute is used.

test_no_bands_is_l()

Test that default (no band) is L.

class satpy.tests.test_composites.**TestInlineComposites**(*methodName='runTest'*)

Bases: TestCase

Test inline composites.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_inline_composites()

Test that inline composites are working.

class satpy.tests.test_composites.**TestLongitudeMaskingCompositor**(*methodName='runTest'*)

Bases: TestCase

Test case for the LongitudeMaskingCompositor compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_masking()

Test longitude masking.

class satpy.tests.test_composites.**TestLuminanceSharpeningCompositor**(*methodName='runTest'*)

Bases: TestCase

Test luminance sharpening compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_compositor()

Test luminance sharpening compositor.

class satpy.tests.test_composites.**TestMaskingCompositor**

Bases: object

Test case for the simple masking compositor.

conditions_v1()

Masking conditions with string values.

conditions_v2()

Masking conditions with numerical values.

reference_alpha()

Get reference alpha to use in masking compositor tests.

reference_data(*test_data*, *test_ct_data*)

Get reference data to use in masking compositor tests.

test_call_named_fields(*conditions_v2*, *test_data*, *test_ct_data*, *reference_data*, *reference_alpha*)

Test with named fields.

test_call_named_fields_string(*conditions_v2*, *test_data*, *test_ct_data*, *reference_data*, *reference_alpha*)

Test with named fields which are as a string in the mask attributes.

test_call_numerical_transparency_data(*conditions_v1*, *test_data*, *test_ct_data*, *reference_data*, *reference_alpha*, *mode*)

Test call the compositor with numerical transparency data.

Use parameterisation to test different image modes.

test_ct_data()

Test 2D CT data array.

test_ct_data_v3(*test_ct_data*)

Set ct data to NaN where it originally is 1.

test_data()

Test data to use with masking compositors.

test_get_flag_value()

Test reading flag value from attributes based on a name.

test_incorrect_method(*test_data*, *test_ct_data*)

Test incorrect method.

test_incorrect_mode(*conditions_v1*)

Test initiating with unsupported mode.

test_init()

Test the initializiation of compositor.

test_method_absolute_import(*test_data*, *test_ct_data_v3*)

Test "absolute_import" as method.

test_method_isnan(*test_data*, *test_ct_data*, *test_ct_data_v3*)

Test "isnan" as method.

test_rgb_dataset(*conditions_v1*, *test_ct_data*, *reference_alpha*)

Test RGB dataset.

test_rgba_dataset(*conditions_v2*, *test_ct_data*, *reference_alpha*)

Test RGBA dataset.

class satpy.tests.test_composites.**TestMatchDataArrays**(*methodName='runTest'*)

Bases: TestCase

Test the utility method 'match_data_arrays'.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_get_test_ds(*shape=(50, 100)*, *dims=('y', 'x')*)

Get a fake DataArray.

test_mult_ds_area()

Test multiple datasets successfully pass.

test_mult_ds_diff_area()

Test that datasets with different areas fail.

test_mult_ds_diff_dims()

Test that datasets with different dimensions still pass.

test_mult_ds_diff_size()

Test that datasets with different sizes fail.

test_mult_ds_no_area()

Test that all datasets must have an area attribute.

test_nondimensional_coords()

Test the removal of non-dimensional coordinates when compositing.

test_single_ds()

Test a single dataset is returned unharmed.

class satpy.tests.test_composites.**TestMultiFiller**(*methodName='runTest'*)

Bases: TestCase

Test case for the MultiFiller compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_fill()

Test filling.

class satpy.tests.test_composites.**TestNaturalEnhCompositor**(*methodName='runTest'*)

Bases: TestCase

Test NaturalEnh compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create channel data and set channel weights.

test_natural_enh(*match_data_arrays*, *repr_*)

Test NaturalEnh compositor.

class satpy.tests.test_composites.**TestPaletteCompositor**(*methodName='runTest'*)

Bases: TestCase

Test the PaletteCompositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_call()

Test palette compositing.

class satpy.tests.test_composites.**TestPrecipCloudsCompositor**(*methodName='runTest'*)

Bases: TestCase

Test the PrecipClouds compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_call()

Test the precip composite generation.

class satpy.tests.test_composites.**TestRatioSharpenedCompositors**

Bases: object

Test RatioSharpenedRGB and SelfSharpendRGB compositors.

setup_method()

Create test data.

test_bad_colors(*init_kwargs*)

Test that only valid band colors can be provided.

test_basic_no_high_res()

Test that three datasets can be passed without optional high res.

test_basic_no_sharpen()

Test that color None does no sharpening.

test_match_data_arrays()

Test that all areas have to be the same resolution.

test_more_than_three_datasets()

Test that only 3 datasets can be passed.

test_ratio_sharpening(*high_resolution_band*, *neutral_resolution_band*, *exp_r*, *exp_g*, *exp_b*)

Test RatioSharpenedRGB by different groups of high_resolution_band and neutral_resolution_band.

test_self_sharpened_basic(*exp_shape*, *exp_r*, *exp_g*, *exp_b*)

Test that three datasets can be passed without optional high res.

test_self_sharpened_no_high_res()

Test for exception when no high_res band is specified.

class satpy.tests.test_composites.**TestSandwichCompositor**

Bases: object

Test sandwich compositor.

test_compositor(*e2d*, *input_shape*, *bands*)

Test luminance sharpening compositor.

class satpy.tests.test_composites.**TestSingleBandCompositor**(*methodName='runTest'*)

Bases: TestCase

Test the single-band compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data.

test_call()

Test calling the compositor.

class satpy.tests.test_composites.**TestStaticImageCompositor**(*methodName='runTest'*)

Bases: TestCase

Test case for the static compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_call(*Scene*, *register*, *retrieve*)

Test the static compositing.

test_init(*get_area_def*)

Test the initializiation of static compositor.

satpy.tests.test_composites.**_create_fake_composite_config**(*yaml_filename: str*)

satpy.tests.test_composites.**_enhance2dataset**(*dataset*, *convert_p=False*)

Mock the enhance2dataset to return the original data.

satpy.tests.test_composites.**fake_area**()

Return a fake 2×2 area.

satpy.tests.test_composites.**fake_dataset_pair**(*fake_area*)

Return a fake pair of 2×2 datasets.

satpy.tests.test_composites.**test_bad_sensor_yaml_configs**(*tmp_path*)

Test composite YAML file with no sensor isn't loaded.

But the bad YAML also shouldn't crash composite configuration loading.

satpy.tests.test_composites.**test_ratio_compositor**(*fake_dataset_pair*)

Test the ratio compositor.

satpy.tests.test_composites.**test_sum_compositor**(*fake_dataset_pair*)

Test the sum compositor.

satpy.tests.test_config module

Test objects and functions in the satpy.config module.

class satpy.tests.test_config.**TestBuiltinAreas**(*methodName='runTest'*)

Bases: TestCase

Test that the builtin areas are all valid.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_areas_pyproj()

Test all areas have valid projections with pyproj.

test_areas_rasterio()

Test all areas have valid projections with rasterio.

class satpy.tests.test_config.**TestConfigObject**

Bases: object

Test basic functionality of the central config object.

test_bad_str_config_path()

Test that a str config path isn't allowed.

test_config_path_multiple()

Test that multiple config paths are accepted.

test_config_path_multiple_load()

Test that config paths from subprocesses load properly.

Satpy modifies the config path environment variable when it is imported. If Satpy is imported again from a subprocess then it should be able to parse this modified variable.

test_custom_config_file()

Test adding a custom configuration file using SATPY_CONFIG.

test_deprecated_env_vars()

Test that deprecated variables are mapped to new config.

test_tmp_dir_is_writable()

Check that the default temporary directory is writable.

class satpy.tests.test_config.**TestPluginsConfigs**

Bases: object

Test that plugins are working.

static _check_available_component(*available_func*, *exp_component*)

static _get_and_check_reader_writer_configs(*specified_component*, *configs_func*, *exp_yaml*)

setup_method()

Set up the test.

test_get_plugin_configs(*fake_composite_plugin_etc_path*)

Check that the plugin configs are looked for.

test_load_entry_point_composite(*fake_composite_plugin_etc_path*)

Test that composites can be loaded from plugin entry points.

test_plugin_enhancements_generic_sensor(*fake_enh_plugin_etc_path*, *sensor_name*, *exp_result*)

Test that enhancements from a plugin are available.

test_plugin_reader_available_readers(*fake_reader_plugin_etc_path*)

Test that readers can be loaded from plugin entry points.

test_plugin_reader_configs(*fake_reader_plugin_etc_path*, *specified_reader*)

Test that readers can be loaded from plugin entry points.

test_plugin_writer_available_writers(*fake_writer_plugin_etc_path*)

Test that readers can be loaded from plugin entry points.

test_plugin_writer_configs(*fake_writer_plugin_etc_path*, *specified_writer*)

Test that writers can be loaded from plugin entry points.

satpy.tests.test_config.**_create_fake_importlib_files**(*module_paths: dict[str, Path]*) → Callable[[str], Path]

satpy.tests.test_config.**_create_fake_iter_entry_points**(*entry_points: dict[str, list[EntryPoint]]*) → Callable[[], dict[str, EntryPoint]]

satpy.tests.test_config.**_create_yamlbased_plugin**(*tmp_path: Path*, *component_type: str*, *yaml_name: str*, *yaml_func: Callable[[str], None]*) → Iterator[Path]

satpy.tests.test_config.**_get_entry_points_and_etc_paths**(*tmp_path: Path*, *entry_point_names: dict[str, list[str]]*) → tuple[Path, dict[str, list[EntryPoint]], dict[str, Path]]

satpy.tests.test_config.**_is_writable**(*directory*)

satpy.tests.test_config.**_os_specific_multipaths**()

satpy.tests.test_config.**_write_fake_composite_yaml**(*yaml_filename: str*) → None

satpy.tests.test_config.**_write_fake_enh_yamls**(*yaml_filename: str*) → None

satpy.tests.test_config.**_write_fake_reader_yaml**(*yaml_filename: str*) → None

satpy.tests.test_config.**_write_fake_writer_yaml**(*yaml_filename: str*) → None

satpy.tests.test_config.**fake_composite_plugin_etc_path**(*tmp_path: Path*) → Iterator[Path]

Create a fake plugin entry point with a fake compositor YAML configuration file.

satpy.tests.test_config.**fake_enh_plugin_etc_path**(*tmp_path: Path*) → Iterator[Path]

Create a fake plugin entry point with a fake enhancement YAML configure files.

This creates a `fake_sensor.yaml` and `generic.yaml` enhancement configuration.

satpy.tests.test_config.**fake_plugin_etc_path**(*tmp_path: Path*, *entry_point_names: dict[str, list[str]]*) → Iterator[Path]

Create a fake satpy plugin entry point.

This mocks the necessary methods to trick Satpy into thinking a plugin package is installed and has made a satpy plugin available.

satpy.tests.test_config.**fake_reader_plugin_etc_path**(*tmp_path: Path*) → Iterator[Path]

Create a fake plugin entry point with a fake reader YAML configuration file.

satpy.tests.test_config.**fake_writer_plugin_etc_path**(*tmp_path: Path*) → Iterator[Path]

Create a fake plugin entry point with a fake writer YAML configuration file.

satpy.tests.test_config.**test_is_writable**()

Test writable directory check.

satpy.tests.test_crefl_utils module

Test CREFL rayleigh correction functions.

class satpy.tests.test_crefl_utils.**TestCreflUtils**(*methodName='runTest'*)

Bases: TestCase

Test crefl_utils.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_get_atm_variables_abi()

Test getting atmospheric variables for ABI.

satpy.tests.test_data_download module

Test for ancillary data downloading.

class satpy.tests.test_data_download.**TestDataDownload**

Bases: object

Test basic data downloading functionality.

_setup_custom_configs(*tmpdir*)

test_download_script()

Test basic functionality of the download script.

test_find_registerable(*readers*, *writers*, *comp_sensors*)

Test that find_registerable finds some things.

test_limited_find_registerable()

Test that find_registerable doesn't find anything when limited.

test_no_downloads_in_tests()

Test that tests aren't allowed to download stuff.

test_offline_retrieve()

Test retrieving a single file when offline.

test_offline_retrieve_all()

Test registering and retrieving all files fails when offline.

test_retrieve()

Test retrieving a single file.

test_retrieve_all()

Test registering and retrieving all files.

class satpy.tests.test_data_download.**UnfriendlyModifier**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *ModifierBase*, *DataDownloadMixin*

Fake modifier that raises an exception in __init__.

Raise an exception if we weren't provided any prerequisites.

satpy.tests.test_data_download.**_assert_comp_files_downloaded**(*comp_sensors*, *found_files*)

satpy.tests.test_data_download.**_assert_mod_files_downloaded**(*comp_sensors*, *found_files*)

satpy.tests.test_data_download.**_assert_reader_files_downloaded**(*readers*, *found_files*)

satpy.tests.test_data_download.**_assert_writer_files_downloaded**(*writers*, *found_files*)

satpy.tests.test_data_download.**_setup_custom_composite_config**(*base_dir*)

satpy.tests.test_data_download.**_setup_custom_reader_config**(*base_dir*)

satpy.tests.test_data_download.**_setup_custom_writer_config**(*base_dir*)

satpy.tests.test_dataset module

Test objects and functions in the dataset module.

class satpy.tests.test_dataset.**TestCombineMetadata**(*methodName='runTest'*)

Bases: TestCase

Test how metadata is combined.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_average_datetimes()

Test the average_datetimes helper function.

test_combine_arrays()

Test the combine_metadata with arrays.

test_combine_dask_arrays()

Test combining values that are dask arrays.

test_combine_empty_metadata()

Test combining empty metadata.

test_combine_identical_numpy_scalars()

Test combining identical fill values.

test_combine_lists_different_size()

Test combine metadata with different size lists.

test_combine_lists_identical()

Test combine metadata with identical lists.

test_combine_lists_same_size_diff_values()

Test combine metadata with lists with different values.

test_combine_nans()

Test combining nan fill values.

test_combine_numpy_arrays()

Test combining values that are numpy arrays.

test_combine_one_metadata_object()

Test combining one metadata object.

test_combine_real_world_mda()

Test with real data.

test_combine_times_with_averaging()

Test the combine_metadata with times with averaging.

test_combine_times_without_averaging()

Test the combine_metadata with times without averaging.

class satpy.tests.test_dataset.**TestDataID**(*methodName='runTest'*)

Bases: TestCase

Test DataID object creation and other methods.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_bad_calibration()

Test that asking for a bad calibration fails.

test_basic_init()

Test basic ways of creating a DataID.

test_compare_no_wl()

Compare fully qualified wavelength ID to no wavelength ID.

test_create_less_modified_query()

Test that modifications are popped correctly.

test_init_bad_modifiers()

Test that modifiers are a tuple.

test_is_modified()

Test that modifications are detected properly.

class satpy.tests.test_dataset.**TestDataQuery**

Bases: object

Test case for data queries.

test_create_less_modified_query()

Test that modifications are popped correctly.

test_dataquery()

Test DataQuery objects.

test_is_modified()

Test that modifications are detected properly.

class satpy.tests.test_dataset.**TestIDQueryInteractions**(*methodName='runTest'*)

Bases: TestCase

Test the interactions between DataIDs and DataQuerys.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp() → None

Set up the test case.

test_hash_equality()

Test hash equality.

test_id_filtering()

Check did filtering.

test_inequality()

Check (in)equality.

test_seviri_hrv_has_priority_over_vis008()

Check that the HRV channel has priority over VIS008 when querying 0.8µm.

test_sort_dataids()

Check dataid sorting.

test_sort_dataids_with_different_set_of_keys()

Check sorting data ids when the query has a different set of keys.

satpy.tests.test_dataset.**test_combine_dicts_close()**

Test combination of dictionaries whose values are close.

satpy.tests.test_dataset.**test_combine_dicts_different**(*test_mda*)

Test combination of dictionaries differing in various ways.

satpy.tests.test_dataset.**test_dataid**()

Test the DataID object.

satpy.tests.test_dataset.**test_dataid_copy**()

Test copying a DataID.

satpy.tests.test_dataset.**test_dataid_elements_picklable**()

Test individual elements of DataID can be pickled.

In some cases, like in the base reader classes, the elements of a DataID are extracted and stored in a separate dictionary. This means that the internal/fancy pickle handling of DataID does not play a part.

satpy.tests.test_dataset.**test_dataid_equal_if_enums_different**()

Check that dataids with different enums but same items are equal.

satpy.tests.test_dataset.**test_dataid_pickle**()

Test dataid pickling roundtrip.

satpy.tests.test_dataset.**test_frequency_double_side_band_channel_containment**()

Test the frequency double side band object: check if one band contains another.

satpy.tests.test_dataset.**test_frequency_double_side_band_channel_distances**()

Test the frequency double side band object: get the distance between two bands.

satpy.tests.test_dataset.**test_frequency_double_side_band_channel_equality**()

Test the frequency double side band object: check if two bands are 'equal'.

satpy.tests.test_dataset.**test_frequency_double_side_band_channel_str**()

Test the frequency double side band object: test the band description.

satpy.tests.test_dataset.**test_frequency_double_side_band_class_method_convert**()

Test the frequency double side band object: test the class method convert.

satpy.tests.test_dataset.**test_frequency_quadruple_side_band_channel_containment**()

Test the frequency quadruple side band object: check if one band contains another.

satpy.tests.test_dataset.**test_frequency_quadruple_side_band_channel_distances**()

Test the frequency quadruple side band object: get the distance between two bands.

satpy.tests.test_dataset.**test_frequency_quadruple_side_band_channel_equality**()

Test the frequency quadruple side band object: check if two bands are 'equal'.

satpy.tests.test_dataset.**test_frequency_quadruple_side_band_channel_str**()

Test the frequency quadruple side band object: test the band description.

satpy.tests.test_dataset.**test_frequency_quadruple_side_band_class_method_convert**()

Test the frequency double side band object: test the class method convert.

satpy.tests.test_dataset.**test_frequency_range_channel_containment**()

Test the frequency range object: channel containment.

satpy.tests.test_dataset.**test_frequency_range_channel_distances**()

Test the frequency range object: derive distances between bands.

satpy.tests.test_dataset.**test_frequency_range_channel_equality**()

Test the frequency range object: check if two bands are 'equal'.

satpy.tests.test_dataset.**test_frequency_range_class_method_convert**()

Test the frequency range object: test the class method convert.

satpy.tests.test_dataset.**test_frequency_range_class_method_str**()

Test the frequency range object: test the band description.

satpy.tests.test_dataset.**test_wavelength_range**()

Test the wavelength range object.

satpy.tests.test_dataset.**test_wavelength_range_cf_roundtrip**()

Test the wavelength range object roundtrip to cf.

satpy.tests.test_demo module

Tests for the satpy.demo module.

class satpy.tests.test_demo.**TestAHIDemoDownload**

Bases: object

Test the AHI demo data download.

test_ahi_full_download()

Test that the himawari download works as expected.

test_ahi_partial_download()

Test that the himawari download works as expected.

class satpy.tests.test_demo.**TestDemo**(*methodName='runTest'*)

Bases: TestCase

Test demo data download functions.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create temporary directory to save files to.

tearDown()

Remove the temporary directory created for a test.

test_get_hurricane_florence_abi(*gcsfs_mod*)

Test data download function.

test_get_us_midlatitude_cyclone_abi(*gcsfs_mod*)

Test data download function.

class satpy.tests.test_demo.**TestGCPUtils**(*methodName='runTest'*)

Bases: TestCase

Test Google Cloud Platform utilities.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_get_bucket_files(*gcsfs_mod*)

Test get_bucket_files basic cases.

test_is_gcp_instance(*uo*)

Test is_google_cloud_instance.

test_no_gcsfs()

Test that 'gcsfs' is required.

class satpy.tests.test_demo.**TestSEVIRIHRITDemoDownload**(*methodName='runTest'*)

Bases: TestCase

Test case for downloading an hrit tarball.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

tearDown()

Tear down the test case.

test_do_not_download_same_file_twice()

Test that files are not downloaded twice.

test_download_a_subset_of_files()

Test downloading a subset of files.

test_download_from_zenodo()

Test downloading SEVIRI HRIT data from zenodo.

test_download_gets_files_with_contents()

Test downloading SEVIRI HRIT data with content.

test_download_to_output_directory()

Test downloading to an output directory.

class satpy.tests.test_demo.**TestVIIRSSDRDemoDownload**

Bases: object

Test VIIRS SDR downloading.

ALL_BAND_PREFIXES = ('SVI01', 'SVI02', 'SVI03', 'SVI04', 'SVI05', 'SVM01', 'SVM02', 'SVM03', 'SVM04', 'SVM05', 'SVM06', 'SVM07', 'SVM08', 'SVM09', 'SVM10', 'SVM11', 'SVM12', 'SVM13', 'SVM14', 'SVM15', 'SVM16', 'SVDNB')

ALL_GEO_PREFIXES = ('GITCO', 'GMTCO', 'GDNBO')

static _assert_bands_in_filenames(*band_prefixes*, *filenames*, *num_files_per_band*)

_assert_bands_in_filenames_and_contents(*band_prefixes*, *filenames*, *num_files_per_band*)

static _assert_file_contents(*filenames*)

test_do_not_download_the_files_twice(*requests*, *tmpdir*)

Test re-downloading VIIRS SDR data.

test_download(*requests*, *tmpdir*)

Test downloading VIIRS SDR data.

test_download_channels_num_granules_dnb(*requests*, *tmpdir*)

Test downloading and re-downloading VIIRS SDR DNB data with select granules.

test_download_channels_num_granules_im(*requests*, *tmpdir*)

Test downloading VIIRS SDR I/M data with select granules.

test_download_channels_num_granules_im_twice(*requests*, *tmpdir*)

Test re-downloading VIIRS SDR I/M data with select granules.

class satpy.tests.test_demo.**_FakeRequest**(*url*, *stream=None*, *timeout=None*)

Bases: object

Fake object to act like a requests return value when downloading a file.

_get_fake_bytesio()

iter_content(*chunk_size*)

Return generator of 'chunk_size' at a time.

raise_for_status()

requests_log: list[str] = []

class satpy.tests.test_demo.**_GlobHelper**(*num_results*)

Bases: object

Create side effect function for mocking gcsfs glob method.

Initialize side_effect function for mocking gcsfs glob method.

Parameters

num_results (*int or list*) – Number of results for each glob call to return. If a list then number of results per call. The last number is used for any additional calls.

satpy.tests.test_demo.**_create_and_populate_dummy_tarfile**(*fn*)

Populate a dummy tarfile with dummy files.

satpy.tests.test_demo.**mock_filesystem**()

Create a mock filesystem, patching *open* and *os.path.isfile*.

satpy.tests.test_demo.**test_fci_download**(*tmp_path*, *monkeypatch*)

Test download of FCI test data.

satpy.tests.test_demo.**test_fs**()

Test the mock filesystem.

satpy.tests.test_dependency_tree module

Unit tests for the dependency tree class and dependencies.

class satpy.tests.test_dependency_tree.**TestDependencyTree**(*methodName='runTest'*)

Bases: TestCase

Test the dependency tree.

This is what we are working with:

```
None (No Data)
 +DataID(name='comp19')
 + +DataID(name='ds5', resolution=250, modifiers=('res_change',))
 + + +DataID(name='ds5', resolution=250, modifiers=())
 + + +__EMPTY_LEAF_SENTINEL__ (No Data)
 + +DataID(name='comp13')
 + + +DataID(name='ds5', resolution=250, modifiers=('res_change',))
 + + + +DataID(name='ds5', resolution=250, modifiers=())
 + + + +__EMPTY_LEAF_SENTINEL__ (No Data)
 + +DataID(name='ds2', resolution=250, calibration=<calibration.reflectance>,␣
↪modifiers=())
```

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

static _nodes_equal(*node_list1*, *node_list2*)

setUp()

Set up the test tree.

test_copy_preserves_all_nodes()

Test that dependency tree copy preserves all nodes.

test_copy_preserves_unique_empty_node()

Test that dependency tree copy preserves the uniqueness of the empty node.

test_new_dependency_tree_preserves_unique_empty_node()

Test that dependency tree instantiation preserves the uniqueness of the empty node.

class satpy.tests.test_dependency_tree.**TestMissingDependencies**(*methodName='runTest'*)

Bases: TestCase

Test the MissingDependencies exception.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_new_missing_dependencies()

Test new MissingDependencies.

test_new_missing_dependencies_with_message()

Test new MissingDependencies with a message.

class satpy.tests.test_dependency_tree.**TestMultipleResolutionSameChannelDependency**(*methodName='runTest'*)

Bases: TestCase

Test that MODIS situations where the same channel is available at multiple resolution works.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_modis_overview_1000m()

Test a modis overview dependency calculation with resolution fixed to 1000m.

class satpy.tests.test_dependency_tree.**TestMultipleSensors**(*methodName='runTest'*)

Bases: TestCase

Test cases where multiple sensors are available.

This is what we are working with:

```
None (No Data)
 +DataID(name='comp19')
 + +DataID(name='ds5', resolution=250, modifiers=('res_change',))
 + + +DataID(name='ds5', resolution=250, modifiers=())
 + + +__EMPTY_LEAF_SENTINEL__ (No Data)
 + +DataID(name='comp13')
 + + +DataID(name='ds5', resolution=250, modifiers=('res_change',))
 + + + +DataID(name='ds5', resolution=250, modifiers=())
 + + + +__EMPTY_LEAF_SENTINEL__ (No Data)
 + +DataID(name='ds2', resolution=250, calibration=<calibration.reflectance>,␣
↪modifiers=())
```

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test tree.

test_compositor_loaded_sensor_order()

Test that a compositor is loaded from the first alphabetical sensor.

test_modifier_loaded_sensor_order()

Test that a modifier is loaded from the first alphabetical sensor.

satpy.tests.test_file_handlers module

test file handler baseclass.

class satpy.tests.test_file_handlers.**TestBaseFileHandler**(*methodName='runTest'*)

Bases: TestCase

Test the BaseFileHandler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test.

test_combine_area(*sdef*)

Combine area.

test_combine_orbital_parameters()

Combine orbital parameters.

test_combine_orbits()

Combine orbits.

test_combine_time_parameters()

Combine times in 'time_parameters.

test_combine_times()

Combine times.

test_file_is_kept_intact()

Test that the file object passed (string, path, or other) is kept intact.

satpy.tests.test_file_handlers.**test_file_type_match**(*file_type*, *ds_file_type*, *exp_result*)

Test that file type matching uses exactly equality.

satpy.tests.test_file_handlers.**test_open_dataset**()

Test xr.open_dataset wrapper.

satpy.tests.test_modifiers module

Tests for modifiers in modifiers/__init__.py.

class satpy.tests.test_modifiers.**TestNIREmissivePartFromReflectance**(*methodName='runTest'*)

Bases: TestCase

Test the NIR Emissive part from reflectance compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_compositor(*calculator*, *apply_modifier_info*, *sza*)

Test the NIR emissive part from reflectance compositor.

class satpy.tests.test_modifiers.**TestNIRReflectance**(*methodName='runTest'*)

Bases: TestCase

Test NIR reflectance compositor.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

fake_refl_from_tbs(*sun_zenith*, *da_nir*, *da_tb11*, *tb_ir_co2=None*)

Fake refl_from_tbs.

setUp()

Set up the test case for the NIRReflectance compositor.

test_masking_limit_default_value_is_not_none(*calculator*, *apply_modifier_info*, *sza*)

Check that sun_zenith_threshold is not None.

test_no_sunz_no_co2(*calculator*, *apply_modifier_info*, *sza*)

Test NIR reflectance compositor with minimal parameters.

test_no_sunz_with_co2(*calculator*, *apply_modifier_info*, *sza*)

Test NIR reflectance compositor provided extra co2 info.

test_provide_masking_limit(*calculator*, *apply_modifier_info*, *sza*)

Test NIR reflectance compositor provided sunz and a sunz threshold.

test_provide_sunz_and_threshold(*calculator*, *apply_modifier_info*, *sza*)

Test NIR reflectance compositor provided sunz and a sunz threshold.

test_provide_sunz_no_co2(*calculator*, *apply_modifier_info*, *sza*)

Test NIR reflectance compositor provided only sunz.

test_sunz_threshold_default_value_is_not_none(*calculator*, *apply_modifier_info*, *sza*)

Check that sun_zenith_threshold is not None.

class satpy.tests.test_modifiers.**TestPSPAtmosphericalCorrection**(*methodName='runTest'*)

Bases: TestCase

Test the pyspectral-based atmospheric correction modifier.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_call()

Test atmospherical correction.

class satpy.tests.test_modifiers.**TestPSPRayleighReflectance**

Bases: object

Test the pyspectral-based Rayleigh correction modifier.

_create_test_data(*name*, *wavelength*, *resolution*)

_get_angles_prereqs_and_opts(*as_optionals*)

_make_data_area()

Create test area definition and data.

test_rayleigh_corrector(*name*, *wavelength*, *resolution*, *aerosol_type*, *reduce_lim_low*, *reduce_lim_high*, *reduce_strength*, *exp_mean*, *exp_unique*)

Test PSPRayleighReflectance with fake data.

test_rayleigh_with_angles(*as_optionals*)

Test PSPRayleighReflectance with angles provided.

class satpy.tests.test_modifiers.**TestSunZenithCorrector**

Bases: object

Test case for the zenith corrector.

test_basic_default_not_provided(*sunz_ds1*, *as_32bit*)

Test default limits when SZA isn't provided.

test_basic_default_provided(*data_arr*, *sunz_sza*)

Test default limits when SZA is provided.

test_basic_lims_not_provided(*sunz_ds1*)

Test custom limits when SZA isn't provided.

test_basic_lims_provided(*data_arr*, *sunz_sza*)

Test custom limits when SZA is provided.

test_imcompatible_areas(*sunz_ds2*, *sunz_sza*)

Test sunz correction on incompatible areas.

class satpy.tests.test_modifiers.**TestSunZenithReducer**

Bases: object

Test case for the sun zenith reducer.

classmethod setup_class()

Initialze SunZenithReducer classes that shall be tested.

test_custom_settings(*sunz_ds1*, *sunz_sza*)

Test custom settings with sza data available.

test_default_settings(*sunz_ds1*, *sunz_sza*)

Test default settings with sza data available.

test_invalid_max_sza(*sunz_ds1*, *sunz_sza*)

Test invalid max_sza with sza data available.

satpy.tests.test_modifiers.**_get_ds1**(*attrs*)

satpy.tests.test_modifiers.**_shared_sunz_attrs**(*area_def*)

satpy.tests.test_modifiers.**_sunz_area_def**()

Get fake area for testing sunz generation.

satpy.tests.test_modifiers.**_sunz_bigger_area_def**()

Get area that is twice the size of 'sunz_area_def'.

satpy.tests.test_modifiers.**_sunz_stacked_area_def**()

Get fake stacked area for testing sunz generation.

satpy.tests.test_modifiers.**sunz_ds1**()

Generate fake dataset for sunz tests.

satpy.tests.test_modifiers.**sunz_ds1_stacked**()

Generate fake dataset for sunz tests.

satpy.tests.test_modifiers.**sunz_ds2**()

Generate larger fake dataset for sunz tests.

satpy.tests.test_modifiers.**sunz_sza**()

Generate fake solar zenith angle data array for testing.

satpy.tests.test_node module

Unit tests for the dependency tree class and dependencies.

class satpy.tests.test_node.**FakeCompositor**(*id*)

Bases: object

A fake compositor.

Set up the fake compositor.

class satpy.tests.test_node.**TestCompositorNode**(*methodName='runTest'*)

Bases: TestCase

Test case for the compositor node object.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_add_optional_nodes()

Test adding optional nodes.

test_add_optional_nodes_twice()

Test adding optional nodes twice.

test_add_required_nodes()

Test adding required nodes.

test_add_required_nodes_twice()

Test adding required nodes twice.

test_compositor_node_init()

Test compositor node initialization.

class satpy.tests.test_node.**TestCompositorNodeCopy**(*methodName='runTest'*)

Bases: TestCase

Test case for copying a node.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_node_data_is_copied()

Test that the data of the node is copied.

test_node_data_optional_nodes_are_copies()

Test that the optional nodes of the node data are copied.

test_node_data_required_nodes_are_copies()

Test that the required nodes of the node data are copied.

satpy.tests.test_readers module

Test classes and functions in the readers/__init__.py module.

class satpy.tests.test_readers.**TestDatasetDict**(*methodName='runTest'*)

Bases: TestCase

Test DatasetDict and its methods.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create a test DatasetDict.

test_contains()

Test DatasetDict contains method.

test_get_key()

Test 'get_key' special functions.

test_getitem()

Test DatasetDict getitem with different arguments.

test_init_dict()

Test DatasetDict init with a regular dict argument.

test_init_noargs()

Test DatasetDict init with no arguments.

test_keys()

Test keys method of DatasetDict.

test_setitem()

Test setitem method of DatasetDict.

class satpy.tests.test_readers.**TestFSFile**(*methodName='runTest'*)

Bases: TestCase

Test the FSFile class.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the instance.

tearDown()

Destroy the instance.

test_equality()

Test that FSFile compares equal when it should.

test_fsfile_with_fs_open_file_abides_pathlike()

Test that FSFile abides PathLike for fsspec OpenFile instances.

test_fsfile_with_pathlike()

Test FSFile with path-like object.

test_fsfile_with_regular_filename_abides_pathlike()

Test that FSFile abides PathLike for regular filenames.

test_fsfile_with_regular_filename_and_fs_spec_abides_pathlike()

Test that FSFile abides PathLike for filename+fs instances.

test_hash()

Test that FSFile hashing behaves sanely.

test_open_local_fs_file()

Test opening a localfs file.

test_open_regular_file()

Test opening a regular file.

test_open_zip_fs_openfile()

Test opening a zipfs openfile.

test_open_zip_fs_regular_filename()

Test opening a zipfs with a regular filename provided.

test_regular_filename_is_returned_with_str()

Test that str give the filename.

test_repr_includes_filename()

Test that repr includes the filename.

test_sorting_fsfiles()

Test sorting FSFiles.

class satpy.tests.test_readers.**TestFindFilesAndReaders**

Bases: object

Test the find_files_and_readers utility function.

setup_method()

Wrap HDF5 file handler with our own fake handler.

teardown_method()

Stop wrapping the HDF5 file handler.

test_bad_sensor()

Test bad sensor doesn't find any files.

test_no_parameters(*viirs_file*)

Test with no limiting parameters.

test_no_parameters_both_atms_and_viirs(*viirs_file*, *atms_file*)

Test with no limiting parameters when there area both atms and viirs files in the same directory.

test_old_reader_name_mapping()

Test that requesting old reader names raises a warning.

test_pending_old_reader_name_mapping()

Test that requesting pending old reader names raises a warning.

test_reader_load_failed()

Test that an exception is raised when a reader can't be loaded.

test_reader_name(*viirs_file*)

Test with default base_dir and reader specified.

test_reader_name_matched_end_time(*viirs_file*)

Test with end matching the filename.

End time in the middle of the file time should still match the file.

test_reader_name_matched_start_end_time(*viirs_file*)

Test with start and end time matching the filename.

test_reader_name_matched_start_time(*viirs_file*)

Test with start matching the filename.

Start time in the middle of the file time should still match the file.

test_reader_name_unmatched_start_end_time(*viirs_file*)

Test with start and end time matching the filename.

test_reader_other_name(*monkeypatch*, *tmp_path*)

Test with default base_dir and reader specified.

test_sensor(*viirs_file*)

Test that readers for the current sensor are loaded.

test_sensor_no_files()

Test that readers for the current sensor are loaded.

class satpy.tests.test_readers.**TestGroupFiles**(*methodName='runTest'*)

Bases: TestCase

Test the 'group_files' utility function.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_filenames_abi_glm =
['OR_ABI-L1b-RadF-M6C14_G16_s19000010000000_e19000010005000_c20403662359590.nc',
'OR_ABI-L1b-RadF-M6C14_G16_s19000010010000_e19000010015000_c20403662359590.nc',
'OR_ABI-L1b-RadF-M6C14_G16_s19000010020000_e19000010025000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010000000_e19000010001000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010001000_e19000010002000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010002000_e19000010003000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010003000_e19000010004000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010004000_e19000010005000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010005000_e19000010006000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010006000_e19000010007000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010007000_e19000010008000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010008000_e19000010009000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010009000_e19000010010000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010010000_e19000010011000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010011000_e19000010012000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010012000_e19000010013000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010013000_e19000010014000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010014000_e19000010015000_c20403662359590.nc',
'OR_GLM-L2-GLMF-M3_G16_s19000010015000_e19000010016000_c20403662359590.nc']

setUp()

Set up test filenames to use.

test_bad_reader()

Test that reader not existing causes an error.

test_default_behavior()

Test the default behavior with the 'abi_l1b' reader.

test_default_behavior_set()

Test the default behavior with the 'abi_l1b' reader.

test_large_time_threshold()

Test what happens when the time threshold holds multiple files.

test_multi_readers()

Test passing multiple readers.

test_multi_readers_empty_groups_missing_skip()

Verify empty groups are skipped.

Verify that all groups lacking ABI are skipped, resulting in only three groups that are all non-empty for both instruments.

test_multi_readers_empty_groups_passed()

Verify that all groups are there, resulting in some that are empty.

test_multi_readers_empty_groups_raises_filenotfounderror()

Test behaviour on empty groups passing multiple readers.

Make sure it raises an exception, for there will be groups containing GLM but not ABI.

test_multi_readers_invalid_parameter()

Verify that invalid missing parameter raises ValueError.

test_no_reader()

Test that reader does not need to be provided.

test_non_datetime_group_key()

Test what happens when the start_time isn't used for grouping.

test_two_instruments_files()

Test the behavior when two instruments files are provided.

This is undesired from a user point of view since we don't want G16 and G17 files in the same Scene. Readers (like abi_l1b) are or can be configured to have specific group keys for handling these situations. Due to that this test forces the fallback group keys of ('start_time',).

test_two_instruments_files_split()

Test the default behavior when two instruments files are provided and split.

Tell the sorting to include the platform identifier as another field to use for grouping.

test_unknown_files()

Test that error is raised on unknown files.

test_viirs_orbits()

Test a reader that doesn't use 'start_time' for default grouping.

test_viirs_override_keys()

Test overriding a group keys to add 'start_time'.

class satpy.tests.test_readers.**TestReaderLoader**(*methodName='runTest'*)

Bases: TestCase

Test the *load_readers* function.

Assumes that the VIIRS SDR reader exists and works.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Wrap HDF5 file handler with our own fake handler.

tearDown()

Stop wrapping the HDF5 file handler.

test_all_filtered()

Test behaviour if no file matches the filter parameters.

test_all_filtered_multiple()

Test behaviour if no file matches the filter parameters.

test_almost_all_filtered()

Test behaviour if only one reader has datasets.

test_bad_reader_name_with_filenames()

Test bad reader name with filenames provided.

test_empty_filenames_as_dict()

Test passing filenames as a dictionary with an empty list of filenames.

test_filenames_and_reader()

Test with filenames and reader specified.

test_filenames_as_dict()

Test loading readers where filenames are organized by reader.

test_filenames_as_dict_bad_reader()

Test loading with filenames dict but one of the readers is bad.

test_filenames_as_dict_with_reader()

Test loading from a filenames dict with a single reader specified.

This can happen in the deprecated Scene behavior of passing a reader and a base_dir.

test_filenames_as_path()

Test with filenames specified as pathlib.Path.

test_filenames_only()

Test with filenames specified.

test_missing_requirements(**mocks*)

Test warnings and exceptions in case of missing requirements.

test_no_args()

Test no args provided.

This should check the local directory which should have no files.

class satpy.tests.test_readers.**TestYAMLFiles**

Bases: object

Test and analyze the reader configuration files.

test_available_readers()

Test the 'available_readers' function.

test_available_readers_base_loader(*monkeypatch*)

Test the 'available_readers' function for yaml loader type BaseLoader.

test_filename_matches_reader_name()

Test that every reader filename matches the name in the YAML.

satpy.tests.test_readers.**_assert_is_open_file_and_close**(*opened*)

satpy.tests.test_readers.**_generate_random_string()**

satpy.tests.test_readers.**_posixify_path**(*filename*)

satpy.tests.test_readers.**atms_file**(*tmp_path*, *monkeypatch*)

Create a dummy atms file.

satpy.tests.test_readers.**make_dataid**(***items*)

Make a data id.

satpy.tests.test_readers.**viirs_file**(*tmp_path*, *monkeypatch*)

Create a dummy viirs file.

satpy.tests.test_regressions module

Test fixed bugs.

satpy.tests.test_regressions.**generate_fake_abi_xr_dataset**(*filename*, *chunks=None*, ***kwargs*)

Create a fake xarray dataset for abi data.

This is an incomplete copy of existing file structures.

satpy.tests.test_regressions.**test_1088**(*fake_open_dataset*)

Check that copied arrays gets resampled.

satpy.tests.test_regressions.**test_1258**(*fake_open_dataset*)

Save true_color from abi with radiance doesn't need two resamplings.

satpy.tests.test_regressions.**test_no_enums**(*fake_open_dataset*)

Check that no enums are inserted in the resulting attrs.

satpy.tests.test_resample module

Unittests for resamplers.

class satpy.tests.test_resample.**TestBilinearResampler**(*methodName='runTest'*)

Bases: TestCase

Test the bilinear resampler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_bil_resampling(*xr_resampler*, *create_filename*, *move_existing_caches*)

Test the bilinear resampler.

test_move_existing_caches()

Test that existing caches are moved to a subdirectory.

class satpy.tests.test_resample.**TestBucketAvg**(*methodName='runTest'*)

Bases: TestCase

Test the bucket resampler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_compute_mocked_bucket_avg(*data*, *return_data=None*, ***kwargs*)

Compute the mocked bucket average.

setUp()

Create fake area definitions and resampler to be tested.

test_compute()

Test bucket resampler computation.

test_compute_and_not_use_skipna_handling()

Test bucket resampler computation and not use skipna handling.

test_compute_and_use_skipna_handling()

Test bucket resampler computation and use skipna handling.

test_init()

Test bucket resampler initialization.

test_precompute(*bucket*)

Test bucket resampler precomputation.

test_resample(*pyresample_bucket*)

Test bucket resamplers resample method.

class satpy.tests.test_resample.**TestBucketCount**(*methodName='runTest'*)

Bases: TestCase

Test the count bucket resampler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_compute_mocked_bucket_count(*data*, *return_data=None*, ***kwargs*)

Compute the mocked bucket count.

setUp()

Create fake area definitions and resampler to be tested.

test_compute()

Test count bucket resampler computation.

class satpy.tests.test_resample.**TestBucketFraction**(*methodName='runTest'*)

Bases: TestCase

Test the fraction bucket resampler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create fake area definitions and resampler to be tested.

test_compute()

Test fraction bucket resampler computation.

test_resample(*pyresample_bucket*)

Test fraction bucket resamplers resample method.

class satpy.tests.test_resample.**TestBucketSum**(*methodName='runTest'*)

Bases: TestCase

Test the sum bucket resampler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

_compute_mocked_bucket_sum(*data*, *return_data=None*, ***kwargs*)

Compute the mocked bucket sum.

setUp()

Create fake area definitions and resampler to be tested.

test_compute()

Test sum bucket resampler computation.

test_compute_and_not_use_skipna_handling()

Test bucket resampler computation and not use skipna handling.

test_compute_and_use_skipna_handling()

Test bucket resampler computation and use skipna handling.

class satpy.tests.test_resample.**TestCoordinateHelpers**(*methodName='runTest'*)

Bases: TestCase

Test various utility functions for working with coordinates.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_area_def_coordinates()

Test coordinates being added with an AreaDefinition.

test_swath_def_coordinates()

Test coordinates being added with an SwathDefinition.

class satpy.tests.test_resample.**TestHLResample**(*methodName='runTest'*)

Bases: TestCase

Test the higher level resampling functions.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_type_preserve()

Check that the type of resampled datasets is preserved.

class satpy.tests.test_resample.**TestKDTreeResampler**(*methodName='runTest'*)

Bases: TestCase

Test the kd-tree resampler.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_kd_resampling(*xr_resampler*, *create_filename*, *zarr_open*, *xr_dset*)

Test the kd resampler.

class satpy.tests.test_resample.**TestNativeResampler**

Bases: object

Tests for the 'native' resampling method.

setup_method()

Create test data used by multiple tests.

test_expand_dims()

Test expanding native resampling with 2D data.

test_expand_dims_3d()

Test expanding native resampling with 3D data.

test_expand_reduce_agg_rechunk()

Test that an incompatible factor for the chunk size is rechunked.

This can happen when a user chunks their data that makes sense for the overall shape of the array and for their local machine's performance, but the resulting resampling factor does not divide evenly into that chunk size.

test_expand_reduce_aggregate()

Test classmethod 'expand_reduce' to aggregate by half.

test_expand_reduce_aggregate_identity()

Test classmethod 'expand_reduce' returns the original dask array when factor is 1.

test_expand_reduce_aggregate_invalid(*dim0_factor*)

Test classmethod 'expand_reduce' fails when factor does not divide evenly.

test_expand_reduce_numpy()

Test classmethod 'expand_reduce' converts numpy arrays to dask arrays.

test_expand_reduce_replicate()

Test classmethod 'expand_reduce' to replicate by 2.

test_expand_without_dims()

Test expanding native resampling with no dimensions specified.

test_expand_without_dims_4D()

Test expanding native resampling with 4D data with no dimensions specified.

satpy.tests.test_resample.**get_test_data**(*input_shape=(100, 50)*, *output_shape=(200, 100)*, *output_proj=None*, *input_dims=('y', 'x')*)

Get common data objects used in testing.

Returns

- input_data_on_area: DataArray with dimensions as if it is a gridded dataset.
- input_area_def: AreaDefinition of the above DataArray
- input_data_on_swath: DataArray with dimensions as if it is a swath.
- input_swath: SwathDefinition of the above DataArray
- target_area_def: AreaDefinition to be used as a target for resampling

Return type

tuple

satpy.tests.test_utils module

Testing of utils.

class satpy.tests.test_utils.**TestCheckSatpy**(*methodName='runTest'*)

Bases: TestCase

Test the 'check_satpy' function.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_basic_check_satpy()

Test 'check_satpy' basic functionality.

test_specific_check_satpy()

Test 'check_satpy' with specific features provided.

class satpy.tests.test_utils.**TestGeoUtils**

Bases: object

Testing geo-related utility functions.

test_angle2xyz(*azizen*, *xyz*)

Test the angle2xyz function.

test_lonlat2xyz(*lonlat*, *xyz*)

Test the lonlat2xyz function.

test_proj_units_to_meters(*prj*, *exp_prj*)

Test proj units to meters conversion.

test_xyz2angle(*xyz*, *acos*, *azizen*)

Test xyz2angle.

test_xyz2lonlat(*xyz*, *asin*, *lonlat*)

Test xyz2lonlat.

class satpy.tests.test_utils.**TestGetSatPos**

Bases: object

Tests for 'get_satpos'.

test_get_satpos(*included_prefixes*, *preference*, *expected_result*)

Test getting the satellite position.

test_get_satpos_fails_with_informative_error(*attrs*)

Test that get_satpos raises an informative error message.

test_get_satpos_from_satname(*caplog*)

Test getting satellite position from satellite name only.

satpy.tests.test_utils.**_data_arrays_from_params**(*shapes: list[tuple[int, ...]]*, *chunks: list[tuple[int, ...]]*, *dims: list[tuple[int, ...]]*) → Generator[DataArray, None, None]

satpy.tests.test_utils.**_verify_unchanged_chunks**(*data_arrays: list[DataArray]*, *orig_arrays: list[DataArray]*) → None

satpy.tests.test_utils.**_verify_unified**(*data_arrays: list[DataArray]*) → None

satpy.tests.test_utils.**test_chunk_size_limit**()

Check the chunk size limit computations.

satpy.tests.test_utils.**test_chunk_size_limit_from_dask_config**()

Check the chunk size limit computations.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_filename_dict**()

Test convertion of remote files to fsspec objects.

Case where filenames is a dictionary mapping readers and filenames.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_fsfile**()

Test convertion of remote files to fsspec objects.

Case where the some of the files are already FSFile objects.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_local_files**()

Test convertion of remote files to fsspec objects.

Case without scheme/protocol, which should default to plain filenames.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_local_pathlib_files**()

Test convertion of remote files to fsspec objects.

Case using pathlib objects as filenames.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_mixed_sources**()

Test convertion of remote files to fsspec objects.

Case with mixed local and remote files.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_storage_options**(*open_files*)

Test convertion of remote files to fsspec objects.

Case with storage options given.

satpy.tests.test_utils.**test_convert_remote_files_to_fsspec_windows_paths**()

Test convertion of remote files to fsspec objects.

Case where windows paths are used.

satpy.tests.test_utils.**test_debug_on**(*caplog*)

Test that debug_on is working as expected.

satpy.tests.test_utils.**test_find_in_ancillary**()

Test finding a dataset in ancillary variables.

satpy.tests.test_utils.**test_get_legacy_chunk_size**()

Test getting the legacy chunk size.

satpy.tests.test_utils.**test_import_error_helper**()

Test the import error helper.

satpy.tests.test_utils.**test_logging_on_and_off**(*caplog*)

Test that switching logging on and off works.

satpy.tests.test_utils.**test_make_fake_scene**()

Test the make_fake_scene utility.

Although the make_fake_scene utility is for internal testing purposes, it has grown sufficiently complex that it needs its own testing.

satpy.tests.test_utils.**test_resolution_chunking**(*chunks*, *shape*, *previous_chunks*, *lr_mult*, *chunk_dtype*, *exp_result*)

Test normalize_low_res_chunks helper function.

satpy.tests.test_utils.**test_unify_chunks**(*shapes*, *chunks*, *dims*, *exp_unified*)

Test unify_chunks utility function.

satpy.tests.test_writers module

Test generic writer functions.

class satpy.tests.test_writers.**TestBaseWriter**

Bases: object

Test the base writer class.

setup_method()

Set up tests.

teardown_method()

Remove the temporary directory created for a test.

test_save_dataset_dynamic_filename(*fmt_fn*, *exp_fns*)

Test saving a dataset with a format filename specified.

test_save_dataset_dynamic_filename_with_dir()

Test saving a dataset with a format filename that includes a directory.

test_save_dataset_static_filename()

Test saving a dataset with a static filename specified.

class satpy.tests.test_writers.**TestComplexSensorEnhancerConfigs**

Bases: *_BaseCustomEnhancementConfigTests*

Test enhancement configs that use or expect multiple sensors.

ENH_FN = 'test_sensor1.yaml'

ENH_FN2 = 'test_sensor2.yaml'

TEST_CONFIGS: dict[str, str] = {'test_sensor1.yaml': '\nenhancements:\n test1_sensor1_specific:\n name: test1\n sensor: test_sensor1\n operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch: 0, max_stretch: 200}\n\n ', 'test_sensor2.yaml': '\nenhancements:\n default:\n operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch: 0, max_stretch: 100}\n test1_sensor2_specific:\n name: test1\n sensor: test_sensor2\n operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch: 0, max_stretch: 50}\n exact_multisensor_comp:\n name: my_comp\n sensor: [test_sensor1, test_sensor2]\n operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch: 0, max_stretch: 20}\n '}

test_enhance_bad_query_value()

Test Enhancer doesn't fail when query includes bad values.

test_multisensor_choice()

Test that a DataArray with two sensors works.

test_multisensor_exact()

Test that a DataArray with two sensors can match exactly.

class satpy.tests.test_writers.**TestComputeWriterResults**(*methodName='runTest'*)

Bases: TestCase

Test compute_writer_results().

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create temporary directory to save files to and a mock scene.

tearDown()

Remove the temporary directory created for a test.

test_empty()

Test empty result list.

test_geotiff()

Test writing to mitiff file.

test_mixed()

Test writing to multiple mixed-type files.

test_multiple_geotiff()

Test writing to mitiff file.

test_multiple_simple()

Test writing to geotiff files.

test_simple_image()

Test writing to PNG file.

class satpy.tests.test_writers.**TestEnhancer**(*methodName='runTest'*)

Bases: TestCase

Test basic *Enhancer* functionality with builtin configs.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_basic_init_no_args()

Test Enhancer init with no arguments passed.

test_basic_init_no_enh()

Test Enhancer init requesting no enhancements.

test_basic_init_provided_enh()

Test Enhancer init with string enhancement configs.

test_init_nonexistent_enh_file()

Test Enhancer init with a nonexistent enhancement configuration file.

class satpy.tests.test_writers.**TestEnhancerUserConfigs**

Bases: *_BaseCustomEnhancementConfigTests*

Test *Enhancer* functionality when user's custom configurations are present.

ENH_ENH_FN = 'enhancements/test_sensor.yaml'

ENH_ENH_FN2 = 'enhancements/test_sensor2.yaml'

ENH_FN = 'test_sensor.yaml'

ENH_FN2 = 'test_sensor2.yaml'

ENH_FN3 = 'test_empty.yaml'

TEST_CONFIGS: dict[str, str] = {'enhancements/test_sensor.yaml': '\nenhancements:\n test1_kelvin:\n name: test1\n units: kelvin\n operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch: 0, max_stretch: 20}\n\n ', 'enhancements/test_sensor2.yaml': '\n\n ', 'test_empty.yaml': '', 'test_sensor.yaml': '\nenhancements:\n test1_default:\n name: test1\n operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch: linear, cutoffs: [0., 0.]}\n\n ', 'test_sensor2.yaml': '\n\n\n '}

test_enhance_empty_config()

Test Enhancer doesn't fail with empty enhancement file.

test_enhance_with_sensor_entry()

Test enhancing an image with a configuration section.

test_enhance_with_sensor_entry2()

Test enhancing an image with a more detailed configuration section.

test_enhance_with_sensor_no_entry()

Test enhancing an image that has no configuration sections.

test_no_enhance()

Test turning off enhancements.

test_writer_custom_enhance()

Test using custom enhancements with writer.

test_writer_no_enhance()

Test turning off enhancements with writer.

class satpy.tests.test_writers.**TestOverlays**(*methodName='runTest'*)

Bases: TestCase

Tests for add_overlay and add_decorate functions.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Create test data and mock pycoast/pydecorate.

tearDown()

Turn off pycoast/pydecorate mocking.

test_add_decorate_basic_l()

Test basic add_decorate usage with L data.

test_add_decorate_basic_rgb()

Test basic add_decorate usage with RGB data.

test_add_overlay_basic_l()

Test basic add_overlay usage with L data.

test_add_overlay_basic_rgb()

Test basic add_overlay usage with RGB data.

class satpy.tests.test_writers.**TestReaderEnhancerConfigs**

Bases: *_BaseCustomEnhancementConfigTests*

Test enhancement configs that use reader name.

ENH_FN = 'test_sensor1.yaml'

```
TEST_CONFIGS: dict[str, str] = {'test_sensor1.yaml': '\nenhancements:\n
default_reader2:\n reader: reader2\n operations:\n - name: stretch\n method:
!!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch:
0, max_stretch: 75}\n default:\n operations:\n - name: stretch\n method:
!!python/name:satpy.enhancements.stretch\n kwargs: {stretch: crude, min_stretch:
0, max_stretch: 100}\n test1_reader2_specific:\n name: test1\n reader: reader2\n
operations:\n - name: stretch\n method: !!python/name:satpy.enhancements.stretch\n
kwargs: {stretch: crude, min_stretch: 0, max_stretch: 50}\n
test1_reader1_specific:\n name: test1\n reader: reader1\n operations:\n - name:
stretch\n method: !!python/name:satpy.enhancements.stretch\n kwargs: {stretch:
crude, min_stretch: 0, max_stretch: 200}\n '}
```

_get_enhanced_image(*data_arr*)

_get_test_data_array()

test_no_matching_reader()

Test that a DataArray with no matching 'reader' works.

test_no_reader()

Test that a DataArray with no 'reader' metadata works.

test_only_reader_matches()

Test that a DataArray with only a matching 'reader' works.

test_reader_and_name_match()

Test that a DataArray with a matching 'reader' and 'name' works.

class satpy.tests.test_writers.**TestWritersModule**(*methodName='runTest'*)

Bases: TestCase

Test the writers module.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_show(*mock_get_image*)

Check showing.

test_to_image_1d()

Conversion to image.

test_to_image_2d(*mock_geoimage*)

Conversion to image.

test_to_image_3d(*mock_geoimage*)

Conversion to image.

class satpy.tests.test_writers.**TestYAMLFiles**(*methodName='runTest'*)

Bases: TestCase

Test and analyze the writer configuration files.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_available_writers()

Test the 'available_writers' function.

test_filename_matches_writer_name()

Test that every writer filename matches the name in the YAML.

class satpy.tests.test_writers.**_BaseCustomEnhancementConfigTests**

Bases: object

TEST_CONFIGS: dict[str, str] = {}

classmethod setup_class()

Create fake user configurations.

classmethod teardown_class()

Remove fake user configurations.

satpy.tests.test_writers.**test_group_results_by_output_file**(*tmp_path*)

Test grouping results by output file.

Add a test for grouping the results from save_datasets(..., compute=False) by output file. This is useful if for some reason we want to treat each output file as a seperate computation (that can still be computed together later).

satpy.tests.test_yaml_reader module

Testing the yaml_reader module.

class satpy.tests.test_yaml_reader.**DummyReader**(*filename*, *filename_info*, *filetype_info*)

Bases: *BaseFileHandler*

Dummy reader instance.

Initialize the dummy reader.

property end_time

Return end time.

property start_time

Return start time.

class satpy.tests.test_yaml_reader.**FakeFH**(*start_time*, *end_time*)

Bases: *BaseFileHandler*

Fake file handler class.

Initialize fake file handler.

property end_time

Return end time.

property start_time

Return start time.

satpy.tests.test_yaml_reader.**GVSYReader**()

Get a fixture of the GEOVariableSegmentYAMLReader.

class satpy.tests.test_yaml_reader.**TestFileFileYAMLReader**(*methodName='runTest'*)

Bases: TestCase

Test units from FileYAMLReader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Prepare a reader instance with a fake config.

test_all_data_ids()

Check that all datasets ids are returned.

test_all_dataset_names()

Get all dataset names.

test_available_dataset_ids()

Get ids of the available datasets.

test_available_dataset_names()

Get ids of the available datasets.

test_deprecated_passing_config_files()

Test that we get an exception when config files are passed to inti.

test_file_covers_area(*bnd*, *adb*, *gad*)

Test that area coverage is checked properly.

test_filter_fh_by_time()

Check filtering filehandlers by time.

test_get_coordinates_for_dataset_key()

Test getting coordinates for a key.

test_get_coordinates_for_dataset_key_without()

Test getting coordinates for a key without coordinates.

test_get_coordinates_for_dataset_keys()

Test getting coordinates for keys.

test_get_file_handlers()

Test getting filehandler to load a dataset.

test_load_area_def(*sad*)

Test loading the area def for the reader.

test_load_entire_dataset(*xarray*)

Check loading an entire dataset.

test_preferred_filetype()

Test finding the preferred filetype.

test_select_from_directory()

Check select_files_from_directory.

test_select_from_pathnames()

Check select_files_from_pathnames.

test_start_end_time()

Check start and end time behaviours.

test_supports_sensor()

Check supports_sensor.

class satpy.tests.test_yaml_reader.**TestFileFileYAMLReaderMultipleFileTypes**(*methodName='runTest'*)

Bases: TestCase

Test units from FileYAMLReader with multiple file types.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Prepare a reader instance with a fake config.

test_update_ds_ids_from_file_handlers()

Test updating existing dataset IDs with information from the file.

class satpy.tests.test_yaml_reader.**TestFileFileYAMLReaderMultiplePatterns**(*methodName='runTest'*)

Bases: TestCase

Test units from FileYAMLReader with multiple readers.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Prepare a reader instance with a fake config.

test_create_filehandlers()

Check create_filehandlers.

test_fn_items_for_ft()

Check filename_items_for_filetype.

test_select_from_pathnames()

Check select_files_from_pathnames.

test_serializable()

Check that a reader is serializable by dask.

This ensures users are able to serialize a Scene object that contains readers.

class satpy.tests.test_yaml_reader.**TestFileYAMLReaderLoading**(*methodName='runTest'*)

Bases: TestCase

Tests for FileYAMLReader.load.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_check_area_for_ch01()

_classSetupFailed = False

_class_cleanups = []

setUp()

Prepare a reader instance with a fake config.

test_load_dataset_with_builtin_coords()

Test loading a dataset with builtin coordinates.

test_load_dataset_with_builtin_coords_in_wrong_order()

Test loading a dataset with builtin coordinates in the wrong order.

class satpy.tests.test_yaml_reader.**TestFileYAMLReaderWithCustomIDKey**(*methodName='runTest'*)

Bases: TestCase

Test units from FileYAMLReader with custom id_keys.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

setUp()

Set up the test case.

test_custom_type_with_dict_contents_gets_parsed_correctly()

Test custom type with dictionary contents gets parsed correctly.

class satpy.tests.test_yaml_reader.**TestGEOFlippableFileYAMLReader**(*methodName='runTest'*)

Bases: TestCase

Test GEOFlippableFileYAMLReader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_load_dataset_with_area_for_data_without_area(*ldwa*)

Test _load_dataset_with_area() for data wihtout area information.

test_load_dataset_with_area_for_single_areas(*ldwa*)

Test _load_dataset_with_area() for single area definitions.

test_load_dataset_with_area_for_stacked_areas(*ldwa*)

Test _load_dataset_with_area() for stacked area definitions.

test_load_dataset_with_area_for_swath_def_data(*ldwa*)

Test _load_dataset_with_area() for swath definition data.

class satpy.tests.test_yaml_reader.**TestGEOSegmentYAMLReader**(*methodName='runTest'*)

Bases: TestCase

Test GEOSegmentYAMLReader.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_find_missing_segments()

Test _find_missing_segments().

test_get_expected_segments(*cfh*)

Test that expected segments can come from the filename.

test_load_area_def(*pesa*, *plsa*, *sad*, *parent_load_area_def*)

Test _load_area_def().

test_load_dataset(*mss*, *xr*, *parent_load_dataset*)

Test _load_dataset().

test_pad_earlier_segments_area(*AreaDefinition*)

Test _pad_earlier_segments_area().

test_pad_later_segments_area(*AreaDefinition*)

Test _pad_later_segments_area().

class satpy.tests.test_yaml_reader.**TestGEOVariableSegmentYAMLReader**

Bases: object

Test GEOVariableSegmentYAMLReader.

test_get_empty_segment(*GVSYReader*, *fake_mss*, *fake_xr*, *fake_geswh*)

Test execution of (overridden) get_empty_segment inside _load_dataset.

test_get_empty_segment_with_height()

Test _get_empty_segment_with_height().

test_pad_earlier_segments_area(*GVSYReader*, *fake_adef*)

Test _pad_earlier_segments_area() for the variable segment case.

test_pad_later_segments_area(*GVSYReader*, *fake_adef*)

Test _pad_later_segments_area() in the variable padding case.

test_pad_later_segments_area_for_multiple_segments_gap(*GVSYReader*, *fake_adef*)

Test _pad_later_segments_area() in the variable padding case for multiple gaps with multiple segments.

class satpy.tests.test_yaml_reader.**TestUtils**(*methodName='runTest'*)

Bases: TestCase

Test the utility functions.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

_classSetupFailed = False

_class_cleanups = []

test_get_filebase()

Check the get_filebase function.

test_listify_string()

Check listify_string.

test_match_filenames()

Check that matching filenames works.

test_match_filenames_windows_forward_slash()

Check that matching filenames works on Windows with forward slashes.

This is common from Qt5 which internally uses forward slashes everywhere.

satpy.tests.test_yaml_reader.**_create_mocked_fh_and_areadef**(*aex*, *ashape*, *expected_segments*, *segment*, *chk_pos_info*)

satpy.tests.test_yaml_reader.**available_datasets**(*self*, *configured_datasets=None*)

Fake available_datasets for testing multiple file types.

satpy.tests.test_yaml_reader.**fake_adef**()

Get a fixture of the patched AreaDefinition.

satpy.tests.test_yaml_reader.**fake_geswh**()

Get a fixture of the patched _get_empty_segment_with_height.

satpy.tests.test_yaml_reader.**fake_mss**()

Get a fixture of the patched _find_missing_segments.

satpy.tests.test_yaml_reader.**fake_xr**()

Get a fixture of the patched xarray.

satpy.tests.test_yaml_reader.**file_type_matches**(*self*, *ds_ftype*)

Fake file_type_matches for testing multiple file types.

satpy.tests.utils module

Utilities for various satpy tests.

class satpy.tests.utils.**CustomScheduler**(*max_computes=1*)

Bases: object

Scheduler raising an exception if data are computed too many times.

Set starting and maximum compute counts.

class satpy.tests.utils.**FakeCompositor**(*name*, *common_channel_mask=True*, ***kwargs*)

Bases: *GenericCompositor*

Act as a compositor that produces fake RGB data.

Collect custom configuration values.

Parameters

common_channel_mask (*bool*) – If True, mask all the channels with a mask that combines all the invalid areas of the given data.

class satpy.tests.utils.**FakeFileHandler**(*filename*, *filename_info*, *filetype_info*, ***kwargs*)

Bases: *BaseFileHandler*

Fake file handler to be used by test readers.

Initialize file handler and accept all keyword arguments.

available_datasets(*configured_datasets=None*)

Report YAML datasets available unless 'not_available' is specified during creation.

property end_time

Get static end time datetime object.

get_dataset(*data_id:* DataID, *ds_info: dict*)

Get fake DataArray for testing.

property sensor_names

Get sensor name from filetype configuration.

property start_time

Get static start time datetime object.

class satpy.tests.utils.**FakeModifier**(*name*, *prerequisites=None*, *optional_prerequisites=None*, ***kwargs*)

Bases: *ModifierBase*

Act as a modifier that performs different modifications.

Initialise the compositor.

_handle_res_change(*datasets*, *info*)

satpy.tests.utils.**_compare_nonarray**(*val1: Any*, *val2: Any*) → None

satpy.tests.utils.**_compare_numpy_array**(*val1: ndarray*, *val2: ndarray*) → None

satpy.tests.utils.**_filter_datasets**(*all_ds*, *names_or_ids*)

Help filtering DataIDs by name or DataQuery.

satpy.tests.utils.**_get_did_for_fake_scene**(*area*, *arr*, *extra_attrs*, *daskify*)

Add instance to fake scene. Helper for make_fake_scene.

satpy.tests.utils.**_get_fake_scene_area**(*arr*, *area*)

Get area for fake scene. Helper for make_fake_scene.

satpy.tests.utils.**_swath_def_of_data_arrays**(*rows*, *cols*)

satpy.tests.utils.**assert_attrs_equal**(*attrs*, *attrs_exp*, *tolerance=0*)

Test that attributes are equal.

Walks dictionary recursively. Numerical attributes are compared with the given relative tolerance.

satpy.tests.utils.**assert_dict_array_equality**(*d1*, *d2*)

Check that dicts containing arrays are equal.

satpy.tests.utils.**assert_maximum_dask_computes**(*max_computes=1*)

Context manager to make sure dask computations are not executed more than `max_computes` times.

satpy.tests.utils.**convert_file_content_to_data_array**(*file_content*, *attrs=()*, *dims=('z', 'y', 'x')*)

Help old reader tests that still use numpy arrays.

A lot of old reader tests still use numpy arrays and depend on the "var_name/attr/attr_name" convention established before Satpy used xarray and dask. While these conventions are still used and should be supported, readers need to use xarray DataArrays instead.

If possible, new tests should be based on pure DataArray objects instead of the "var_name/attr/attr_name" style syntax provided by the utility file handlers.

Parameters

- **file_content** (*dict*) – Dictionary of string file keys to fake file data.
- **attrs** (*iterable*) – Series of attributes to copy to DataArray object from file content dictionary. Defaults to no attributes.
- **dims** (*iterable*) – Dimension names to use for resulting DataArrays. The second to last dimension is used for 1D arrays, so for dims of `('z', 'y', 'x')` this would use `'y'`. Otherwise, the dimensions are used starting with the last, so 2D arrays are `('y', 'x')` Dimensions are used in reverse order so the last dimension specified is used as the only dimension for 1D arrays and the last dimension for other arrays.

satpy.tests.utils.**make_cid**(***items*)

Make a DataID with a minimal set of keys to id composites.

satpy.tests.utils.**make_dataid**(***items*)

Make a DataID with default keys.

satpy.tests.utils.**make_dsq**(***items*)

Make a dataset query.

satpy.tests.utils.**make_fake_scene**(*content_dict*, *daskify=False*, *area=True*, *common_attrs=None*)

Create a fake Scene.

Create a fake Scene object from fake data. Data are provided in the `content_dict` argument. In `content_dict`, keys should be strings or DataID, and values may be either numpy.ndarray or xarray.DataArray, in either case with exactly two dimensions. The function will convert each of the numpy.ndarray objects into an xarray.DataArray and assign those as datasets to a Scene object. A fake AreaDefinition will be assigned for each array, unless disabled by passing `area=False`. When areas are automatically generated, arrays with the same shape will get the same area.

This function is exclusively intended for testing purposes.

If regular ndarrays are passed and the keyword argument daskify is True, DataArrays will be created as dask arrays. If False (default), regular DataArrays will be created. When the user passes xarray.DataArray objects then this flag has no effect.

Parameters

- **content_dict** (*Mapping*) – Mapping where keys correspond to objects accepted by `Scene.__setitem__`, i.e. strings or DataID, and values may be either `numpy.ndarray` or `xarray.DataArray`.
- **daskify** (*bool*) – optional, to use dask when converting `numpy.ndarray` to `xarray.DataArray`. No effect when the values in `content_dict` are already `xarray.DataArray`.
- **area** (*bool or BaseDefinition*) – Can be `True`, `False`, or an instance of `pyresample.geometry.BaseDefinition` such as `AreaDefinition` or `SwathDefinition`. If `True`, which is the default, automatically generate areas with the name "test-area". If `False`, values will not have assigned areas. If an instance of `pyresample.geometry.BaseDefinition`,

those instances will be used for all generated fake datasets. Warning: Passing an area as a string (`area="germ"`) is not supported.

- **common_attrs** (*Mapping*) – optional, additional attributes that will be added to every dataset in the scene.

Returns
Scene object with datasets corresponding to content_dict.

`satpy.tests.utils.spy_decorator(method_to_decorate)`

Fancy decorator to wrap an object while still calling it.

See https://stackoverflow.com/a/41599695/433202

`satpy.tests.utils.xfail_h5py_unstable_numpy2()`

Determine if h5py-based tests should be xfail in the unstable numpy 2.x environment.

`satpy.tests.utils.xfail_skyfield_unstable_numpy2()`

Determine if skyfield-based tests should be xfail in the unstable numpy 2.x environment.

Module contents

The tests package.

satpy.writers package

Submodules

satpy.writers.awips_tiled module

The AWIPS Tiled writer is used to create AWIPS-compatible tiled NetCDF4 files.

The Advanced Weather Interactive Processing System (AWIPS) is a program used by the United States National Weather Service (NWS) and others to view different forms of weather imagery. The original Sectorized Cloud and Moisture Imagery (SCMI) functionality in AWIPS was a NetCDF4 format supported by AWIPS to store one image broken up in to one or more "tiles". This format has since been expanded to support many other products and so the writer for this format in Satpy is generically called the "AWIPS Tiled" writer. You may still see SCMI referenced in this documentation or in the source code for the writer. Once AWIPS is configured for specific products this writer can be used to provide compatible products to the system.

The AWIPS Tiled writer takes 2D (y, x) geolocated data and creates one or more AWIPS-compatible NetCDF4 files. The writer and the AWIPS client may need to be configured to make things appear the way the user wants in the AWIPS client. The writer can only produce files for datasets mapped to areas with specific projections:

- lcc
- geos
- merc
- stere

This is a limitation of the AWIPS client and not of the writer. In the case where AWIPS has been updated to support additional projections, this writer may also need to be updated to support those projections.

AWIPS Configuration

Depending on how this writer is used and the data it is provided, AWIPS may need additional configuration on the server side to properly ingest the files produced. This will require administrator privileges to the ingest server(s) and is not something that can be configured on the client. Note that any changes required must be done on all servers that you wish to ingest your data files. The generic "polar" template this writer defaults to should limit the number of modifications needed for any new data fields that AWIPS previously was unaware of. Once the data is ingested, the client can be used to customize how the data looks on screen.

AWIPS requires files to follow a specific naming scheme so they can be routed to specific "decoders". For the files produced by this writer, this typically means editing the "goesr" decoder configuration in a directory like:

```
/awips2/edex/data/utility/common_static/site/<site>/distribution/goesr.xml
```

The "goesr" decoder is a subclass of the "satellite" decoder. You may see either name show up in the AWIPS ingest logs. With the correct regular expression in the above file, your files should be passed to the right decoder, opened, and parsed for data.

To tell AWIPS exactly what attributes and variables mean in your file, you'll need to create or configure an XML file in:

```
/awips2/edex/data/utility/common_static/site/<site>/satellite/goesr/descriptions/
```

See the existing files in this directory for examples. The "polar" template (see below) that this writer uses by default is already configured in the "Polar" subdirectory assuming that the TOWR-S RPM package has been installed on your AWIPS ingest server.

Templates

This writer allows for a "template" to be specified to control how the output files are structured and created. Templates can be configured in the writer YAML file (`awips_tiled.yaml`) or passed as a dictionary to the `template` keyword argument. Templates have three main sections:

1. global_attributes
2. coordinates
3. variables

Additionally, you can specify whether a template should produce files with one variable per file by specifying `single_variable: true` or multiple variables per file by specifying `single_variable: false`. You can also specify the output filename for a template using a Python format string. See `awips_tiled.yaml` for examples. Lastly, a `add_sector_id_global` boolean parameter can be specified to add the user-provided `sector_id` keyword argument as a global attribute to the file.

The `global_attributes` section takes names of global attributes and then a series of options to "render" that attribute from the metadata provided when creating files. For example:

```
product_name:
    value: "{name}"
```

For more information see the *satpy.writers.awips_tiled.NetCDFTemplate.get_attr_value()* method.

The `coordinates` and `variables` are similar to each other in that they define how a variable should be created, the attributes it should have, and the encoding to write to the file. Coordinates typically don't need to be modified as tiled files usually have only `x` and `y` dimension variables. The Variables on the other hand use a decision tree to determine what section applies for a particular DataArray being saved. The basic structure is:

```
variables:
  arbitrary_section_name:
    <decision tree matching parameters>
    var_name: "output_netcdf_variable_name"
    attributes:
      <attributes similar to global attributes>
    encoding:
      <xarray encoding parameters>
```

The "decision tree matching parameters" can be one or more of "name", "standard_name', "satellite", "sensor", "area_id', "units", or "reader". The writer will choose the best section for the DataArray being saved (the most matches). If none of these parameters are specified in a section then it will be used when no other matches are found (the "default" section).

The "encoding" parameters can be anything accepted by xarray's `to_netcdf` method. See `xarray.Dataset.to_netcdf()` for more information on the *encoding* ` keyword argument.

For more examples see the existing builtin templates defined in `awips_tiled.yaml`.

Builtin Templates

There are only a few templates provided in Sapty currently.

- **polar**: A custom format developed for the CSPP Polar2Grid project at the University of Wisconsin - Madison Space Science and Engineering Center (SSEC). This format is made available through the TOWR-S package that can be installed for GOES-R support in AWIPS. This format is meant to be very generic and should theoretically allow any variable to get ingested into AWIPS.
- **glm_l2_radc**: This format is used to produce standard files for the gridded GLM products produced by the CSPP Geo Gridded GLM package. Support for this format is also available in the TOWR-S package on an AWIPS ingest server. This format is specific to gridded GLM on the CONUS sector and is not meant to work for other data.
- **glm_l2_radf**: This format is used to produce standard files for the gridded GLM productes produced by the CSPP Geo Gridded GLM package. Support for this format is also available in the TOWR-S package on an AWIPS ingest server. This format is specific to gridded GLM on the Full Disk sector and is not meant to work for other data.

Numbered versus Lettered Grids

By default this writer will save tiles by number starting with '1' representing the upper-left image tile. Tile numbers then increase along the column and then on to the next row.

By specifying *lettered_grid* as *True* tiles can be designated with a letter. Lettered grids or sectors are preconfigured in the *awips_tiled.yaml* configuration file. The lettered tile locations are static and will not change with the data being written to them. Each lettered tile is split into a certain number of subtiles (*num_subtiles*), default 2 rows by 2 columns. Lettered tiles are meant to make it easier for receiving AWIPS clients/stations to filter what tiles they receive; saving time, bandwidth, and space.

Any tiles (numbered or lettered) not containing any valid data are not created.

Updating tiles

There are some input data cases where we want to put new data in a tile file written by a previous execution. An example is a pre-tiled input dataset that is processed one tile at a time. One input tile may map to one or more output AWIPS tiles, but may not perfectly aligned, leaving empty/unused space in the output tile. The next input tile may be able to fill in that empty space and should be allowed to write the "new" data to the file. This is the default behavior of the AWIPS tiled writer. In cases where data overlaps the existing data in the tile, the newer data has priority.

Shifting Lettered Grids

Due to the static nature of the lettered grids, there is sometimes a need to shift the locations of where these tiles are by up to 0.5 pixels in each dimension to align with the data being processed. This means that the tiles for a 1000m resolution grid may be shifted up to 500m in each direction from the original definition of the lettered "sector". This can cause differences in the location of the tiles between executions depending on the locations of the input data. In the worst case tile A01 from one execution could be shifted up to 1 grid cell from tile A01 in another execution (one is shifted 0.5 pixels to the left, the other is shifted 0.5 to the right).

This shifting makes the calculations for generating tiles easier and more accurate. By default, the lettered tile locations are changed to match the location of the data. This works well when output tiles will not be updated (see above) in future processing. In cases where output tiles will be filled in or updated with more data the `use_sector_reference` keyword argument can be set to `True` to tell the writer to shift the data's geolocation by up to 0.5 pixels in each dimension instead of shifting the lettered tile locations.

class `satpy.writers.awips_tiled.`**`AWIPSNetCDFTemplate`**(*template_dict*, *swap_end_time=False*)

Bases: *`NetCDFTemplate`*

NetCDF template renderer specifically for tiled AWIPS files.

Handle AWIPS special cases and initialize template helpers.

`_add_sector_id_global`(*new_ds*, *sector_id*)

`_data_units`(*input_metadata*)

static `_fill_units_and_standard_name`(*attrs*, *units*, *standard_name*)

Fill in units and standard_name if not set in *attrs*.

`_get_projection_attrs`(*area_def*)

Assign projection attributes per CF standard.

static `_get_vmin_vmax`(*var_config*, *input_data_arr*)

`_global_awips_id`(*input_metadata*)

`_global_physical_element`(*input_metadata*)

`_global_production_location`(*input_metadata*)

Get default global production_location attribute.

`_global_production_site`(*input_metadata*)

Get default global production_location attribute.

`_global_start_date_time`(*input_metadata*)

`_render_variable_attributes`(*var_config*, *input_metadata*)

`_render_variable_encoding`(*var_config*, *input_data_arr*)

_set_xy_coords_attrs(*new_ds*, *crs*)

_swap_attributes_end_time(*template_dict*)

Swap every use of 'start_time' to use 'end_time' instead.

apply_area_def(*new_ds*, *area_def*)

Apply information we can gather from the AreaDefinition.

apply_misc_metadata(*new_ds*, *sector_id=None*, *creator=None*, *creation_time=None*)

Add attributes that don't fit into any other category.

apply_tile_coord_encoding(*new_ds*, *xy_factors*)

Add encoding information specific to the coordinate variables.

apply_tile_info(*new_ds*, *tile_info*)

Apply attributes associated with the current tile.

render(*dataset_or_data_arrays*, *area_def*, *tile_info*, *sector_id*, *creator=None*, *creation_time=None*, *shared_attrs=None*, *extra_global_attrs=None*)

Create a `xarray.Dataset` from template using information provided.

class satpy.writers.awips_tiled.**AWIPSTiledVariableDecisionTree**(*decision_dicts*, ***kwargs*)

Bases: *DecisionTree*

Load AWIPS-specific metadata from YAML configuration.

Initialize decision tree with specific keys to look for.

class satpy.writers.awips_tiled.**AWIPSTiledWriter**(*compress=False*, *fix_awips=False*, ***kwargs*)

Bases: *Writer*

Writer for AWIPS NetCDF4 Tile files.

See *satpy.writers.awips_tiled* documentation for more information on templates and produced file format.

Initialize writer and decision trees.

_adjust_metadata_times(*ds_info*)

_delay_netcdf_creation(*delayed_gen*, *precompute=True*, *use_distributed=False*)

Workaround random dask and xarray hanging executions.

In previous implementations this writer called 'to_dataset' directly in a delayed function. This seems to cause random deadlocks where execution would hang indefinitely.

_enhance_and_split_rgbs(*datasets*)

Handle multi-band images by splitting in to separate products.

_fill_sector_info()

Convert sector extents if needed.

static _get_delayed_iter(*use_distributed=False*)

_get_lettered_sector_info(*sector_id*)

Get metadata for the current sector if configured.

This is not necessary for numbered grids. If found, the sector info will provide the overall tile layout for this grid/sector. This allows for consistent tile numbering/naming regardless of where the data being converted actually is.

_get_tile_data_info(*data_arrs*, *creation_time*, *source_name*)

_get_tile_generator(*area_def*, *lettered_grid*, *sector_id*, *num_subtiles*, *tile_size*, *tile_count*, *use_sector_reference=False*)

Get the appropriate tile generator class for lettered or numbered tiles.

_group_by_area(*datasets*)

Group datasets by their area.

_iter_area_tile_info_and_datasets(*area_datasets*, *template*, *lettered_grid*, *sector_id*, *num_subtiles*, *tile_size*, *tile_count*, *use_sector_reference*)

_iter_tile_info_and_datasets(*tile_gen*, *data_arrays*, *single_variable=True*)

_save_nonempty_mfdatasets(*datasets_to_save*, *output_filenames*, ***kwargs*)

_slice_and_update_coords(*tile_info*, *data_arrays*)

_split_rgbs(*ds*)

Split a single RGB dataset in to multiple.

_tile_filler(*tile_info*, *data_arr*)

check_tile_exists(*output_filename*)

Check if tile exists and report error accordingly.

property enhancer

Get lazy loaded enhancer object only if needed.

get_filename(*template*, *area_def*, *tile_info*, *sector_id*, ***kwargs*)

Generate output NetCDF file from metadata.

save_dataset(*dataset*, ***kwargs*)

Save a single DataArray to one or more NetCDF4 Tile files.

save_datasets(*datasets*, *sector_id=None*, *source_name=None*, *tile_count=(1, 1)*, *tile_size=None*, *lettered_grid=False*, *num_subtiles=None*, *use_end_time=False*, *use_sector_reference=False*, *template='polar'*, *check_categories=True*, *extra_global_attrs=None*, *environment_prefix='DR'*, *compute=True*, ***kwargs*)

Write a series of DataArray objects to multiple NetCDF4 Tile files.

Parameters

- **datasets** (*iterable*) – Series of gridded `DataArray` objects with the necessary metadata to be converted to a valid tile product file.
- **sector_id** (*str*) – Name of the region or sector that the provided data is on. This name will be written to the NetCDF file and will be used as the sector in the AWIPS client for the 'polar' template. For lettered grids this name should match the name configured in the writer YAML. This is required for some templates (ex. default 'polar' template) but is defined as a keyword argument for better error handling in Satpy.
- **source_name** (*str*) – Name of producer of these files (ex. "SSEC"). This name is used to create the output filename for some templates.
- **environment_prefix** (*str*) – Prefix of filenames for some templates. For operational real-time data this is usually "OR", "OT" for test data, "IR" for test system real-time data, and "IT" for test system test data. This defaults to "DR" for "Developer Real-time" to avoid anyone accidentally producing files that could be mistaken for the operational system.

- **tile_count** (*tuple*) – For numbered tiles only, how many tile rows and tile columns to produce. Default to `(1, 1)`, a single giant tile. Either `tile_count`, `tile_size`, or `lettered_grid` should be specified.
- **tile_size** (*tuple*) – For numbered tiles only, how many pixels each tile should be. This takes precedence over `tile_count` if specified. Either `tile_count`, `tile_size`, or `lettered_grid` should be specified.
- **lettered_grid** (*bool*) – Whether to use a preconfigured grid and label tiles with letters and numbers instead of only numbers. For example, tiles will be named "A01", "A02", "B01", and so on in the first row of data and continue on to "A03", "A04", and "B03" in the default case where `num_subtiles` is (2, 2). Letters start in the upper-left corner and will go from A up to Z, if necessary.
- **num_subtiles** (*tuple*) – For lettered tiles only, how many rows and columns to split each lettered tile in to. By default 2 rows and 2 columns will be created. For example, the tile for letter "A" will have "A01" and "A02" in the top row and "A03" and "A04" in the second row.
- **use_end_time** (*bool*) – Instead of using the `start_time` for the product filename and time written to the file, use the `end_time`. This is useful for multi-day composites where the `end_time` is a better representation of what data is in the file.
- **use_sector_reference** (*bool*) – For lettered tiles only, whether to shift the data locations to align with the preconfigured grid's pixels. By default this is False meaning that the grid's tiles will be shifted to align with the data locations. If True, the data is shifted. At most the data will be shifted by 0.5 pixels. See *satpy.writers.awips_tiled* for more information.
- **template** (*str or dict*) – Name of the template configured in the writer YAML file. This can also be a dictionary with a full template configuration. See the *satpy.writers.awips_tiled* documentation for more information on templates. Defaults to the 'polar' builtin template.
- **check_categories** (*bool*) – Whether category and flag products should be included in the checks for empty or not empty tiles. In some cases (ex. data quality flags) category products may look like all valid data (a non-empty tile) but shouldn't be used to determine the emptiness of the overall tile (good quality versus non-existent). Default is True. Set to False to ignore category (integer dtype or "flag_meanings" defined) when checking for valid data.
- **extra_global_attrs** (*dict*) – Additional global attributes to be added to every produced file. These attributes are applied at the end of template rendering and will therefore overwrite template generated values with the same global attribute name.
- **compute** (*bool*) – Compute and write the output immediately using dask. Default to `False`.

classmethod separate_init_kwargs(*kwargs*)

Separate keyword arguments by initialization and saving keyword arguments.

class satpy.writers.awips_tiled.**LetteredTileGenerator**(*area_definition*, *extents*, *sector_crs*, *cell_size=(2000000, 2000000)*, *num_subtiles=None*, *use_sector_reference=False*)

Bases: *NumberedTileGenerator*

Helper class to generate per-tile metadata for lettered tiles.

Initialize tile information for later generation.

Parameters

- **area_definition** (*AreaDefinition*) – Area of the data being saved.
- **extents** (*tuple*) – Four element tuple of the configured lettered area.
- **sector_crs** (*pyproj.CRS*) – CRS of the configured lettered sector area.
- **cell_size** (*tuple*) – Two element tuple of resolution of each tile in sector projection units (y, x).

_generate_tile_info()

Create generator of individual tile metadata.

_get_tile_properties(*tile_shape*, *tile_count*)

Calculate tile information for this particular sector/grid.

_get_xy_scaling_parameters()

Get the X/Y coordinate limits for the full resulting image.

_tile_identifier(*ty*, *tx*)

Get tile identifier (name) for a particular tile row/column.

class satpy.writers.awips_tiled.**NetCDFTemplate**(*template_dict*)

Bases: object

Helper class to convert a dictionary-based NetCDF template to an xarray.Dataset.

Parse template dictionary and prepare for rendering.

_get_matchable_coordinate_metadata(*coord_name*, *coord_attrs*)

_render_attrs(*attr_configs*, *input_metadata*, *prefix='_'*)

_render_coordinate_attributes(*coord_config*, *input_metadata*)

_render_coordinates(*ds*)

_render_global_attributes(*input_metadata*)

_render_variable(*data_arr*)

_render_variable_attributes(*var_config*, *input_metadata*)

_render_variable_encoding(*var_config*, *input_data_arr*)

get_attr_value(*attr_name*, *input_metadata*, *value=None*, *raw_key=None*, *raw_value=None*, *prefix='_'*)

Determine attribute value using the provided configuration information.

If *value* and *raw_key* are not provided, this method will search for a method named <prefix><attr_name>, which will be called with one argument (*input_metadata*) to get the value to return. See the documentation for the *prefix* keyword argument below for more information.

Parameters

- **attr_name** (*str*) – Name of the attribute whose value we are generating.
- **input_metadata** (*dict*) – Dictionary of metadata from the input DataArray and other context information. Used to provide information to *value* or access data from using *raw_key* if provided.

- **value** (*Any*) – Value to assign to this attribute. If a string, it may be a python format string which will be provided the data from *input_metadata*. For example, `{name}` will be filled with the value for the `"name"` in *input_metadata*. It can also include environment variables (ex. `"${MY_ENV_VAR}"`) which will be expanded. String formatting is accomplished by the special `trollsift.parser.StringFormatter` which allows for special common conversions.
- **raw_key** (*str*) – Key to access value from *input_metadata*, but without any string formatting applied to it. This allows for metadata of non-string types to be requested.
- **raw_value** (*Any*) – Static hardcoded value to set this attribute to. Overrides all other options.
- **prefix** (*str*) – Prefix to use when *value* and *raw_key* are both `None`. Default is `"_"`. This will be used to find custom attribute handlers in subclasses. For example, if *value* and *raw_key* are both `None` and *attr_name* is `"my_attr"`, then the method `self._my_attr` will be called as `return self._my_attr(input_metadata)`. See `NetCDFTemplate.render_global_attributes()` for additional information (prefix is `"_global_"`).

get_filename(*base_dir=''*, ***kwargs*)

Generate output NetCDF file from metadata.

render(*dataset_or_data_arrays*, *shared_attrs=None*)

Create `xarray.Dataset` from provided data.

class satpy.writers.awips_tiled.**NumberedTileGenerator**(*area_definition*, *tile_shape=None*, *tile_count=None*)

Bases: `object`

Helper class to generate per-tile metadata for numbered tiles.

Initialize and generate tile information for this sector/grid for later use.

_generate_tile_info()

Get numbered tile metadata.

_get_tile_properties(*tile_shape*, *tile_count*)

Generate tile information for numbered tiles.

_get_xy_arrays()

Get the overall X/Y coordinate variable arrays.

_get_xy_scaling_parameters()

Get the X/Y coordinate limits for the full resulting image.

_tile_identifier(*ty*, *tx*)

Get tile identifier for numbered tiles.

_tile_number(*ty*, *tx*)

Get tile number from tile row/column.

class satpy.writers.awips_tiled.**TileInfo**(*tile_count*, *image_shape*, *tile_shape*, *tile_row_offset*, *tile_column_offset*, *tile_id*, *tile_number*, *x*, *y*, *xy_factors*, *tile_slices*, *data_slices*)

Bases: `tuple`

Create new instance of TileInfo(tile_count, image_shape, tile_shape, tile_row_offset, tile_column_offset, tile_id, tile_number, x, y, xy_factors, tile_slices, data_slices)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('tile_count', 'image_shape', 'tile_shape', 'tile_row_offset', 'tile_column_offset', 'tile_id', 'tile_number', 'x', 'y', 'xy_factors', 'tile_slices', 'data_slices')

classmethod _make(*iterable*)

Make a new TileInfo object from a sequence or iterable

_replace(***kwds*)

Return a new TileInfo object replacing specified fields with new values

data_slices

Alias for field number 11

image_shape

Alias for field number 1

tile_column_offset

Alias for field number 4

tile_count

Alias for field number 0

tile_id

Alias for field number 5

tile_number

Alias for field number 6

tile_row_offset

Alias for field number 3

tile_shape

Alias for field number 2

tile_slices

Alias for field number 10

x

Alias for field number 7

xy_factors

Alias for field number 9

y

Alias for field number 8

class satpy.writers.awips_tiled.**XYFactors**(*mx*, *bx*, *my*, *by*)

Bases: tuple

Create new instance of XYFactors(mx, bx, my, by)

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('mx', 'bx', 'my', 'by')

classmethod _make(*iterable*)

Make a new XYFactors object from a sequence or iterable

_replace(***kwds*)

Return a new XYFactors object replacing specified fields with new values

bx

Alias for field number 1

by

Alias for field number 3

mx

Alias for field number 0

my

Alias for field number 2

satpy.writers.awips_tiled.**_add_valid_ranges**(*data_arrs*)

Add 'valid_range' metadata if not present.

If valid_range or valid_min/valid_max are not present in a DataArrays metadata (`.attrs`), then lazily compute it with dask so it can be computed later when we write tiles out.

AWIPS requires that scale_factor/add_offset/_FillValue be the **same** for all tiles. We must do this calculation before splitting the data into tiles otherwise the values will be different.

satpy.writers.awips_tiled.**_any_notnull**(*data_arr*, *check_categories*)

satpy.writers.awips_tiled.**_copy_to_existing**(*dataset_to_save*, *output_filename*)

satpy.writers.awips_tiled.**_create_debug_array**(*sector_info*, *num_subtiles*, *font_path='Verdana.ttf'*)

satpy.writers.awips_tiled.**_extract_factors**(*dataset_to_save*)

satpy.writers.awips_tiled.**_get_data_vmin_vmax**(*input_data_arr*)

satpy.writers.awips_tiled.**_get_factor_offset_fill**(*input_data_arr*, *vmin*, *vmax*, *encoding*)

satpy.writers.awips_tiled.**_is_empty_tile**(*dataset_to_save*, *check_categories*)

satpy.writers.awips_tiled.**_notnull**(*data_arr*, *check_categories=True*)

satpy.writers.awips_tiled.**_reapply_factors**(*dataset_to_save*, *factors*)

satpy.writers.awips_tiled.**create_debug_lettered_tiles**(***writer_kwargs*)

Create tile files with tile identifiers "burned" in to the image data for debugging.

satpy.writers.awips_tiled.**draw_rectangle**(*draw*, *coordinates*, *outline=None*, *fill=None*, *width=1*)

Draw simple rectangle in to a numpy array image.

satpy.writers.awips_tiled.**fix_awips_file**(*fn*)

Hack the NetCDF4 files to workaround NetCDF-Java bugs used by AWIPS.

This should not be needed for new versions of AWIPS.

satpy.writers.awips_tiled.**main**()

Command line interface mimicing CSPP Polar2Grid.

satpy.writers.awips_tiled.**tile_filler**(*data_arr_data*, *tile_shape*, *tile_slices*, *fill_value*)

Create an empty tile array and fill the proper locations with data.

satpy.writers.awips_tiled.**to_nonempty_netcdf**(*dataset_to_save: Dataset*, *factors: dict*, *output_filename: str*, *update_existing: bool = True*, *check_categories: bool = True*)

Save `xarray.Dataset` to a NetCDF file if not all fills.

In addition to checking certain Dataset variables for fill values, this function can also "update" an existing NetCDF file with the new valid data provided.

satpy.writers.cf_writer module

Writer for netCDF4/CF.

Example usage

The CF writer saves datasets in a Scene as CF-compliant netCDF file. Here is an example with MSG SEVIRI data in HRIT format:

```
>>> from satpy import Scene
>>> import glob
>>> filenames = glob.glob('data/H*201903011200*')
>>> scn = Scene(filenames=filenames, reader='seviri_l1b_hrit')
>>> scn.load(['VIS006', 'IR_108'])
>>> scn.save_datasets(writer='cf', datasets=['VIS006', 'IR_108'], filename='seviri_test.
↪nc',
                      exclude_attrs=['raw_metadata'])
```

- You can select the netCDF backend using the `engine` keyword argument. If *None* if follows `to_netcdf()` engine choices with a preference for 'netcdf4'.
- For datasets with area definition you can exclude lat/lon coordinates by setting `include_lonlats=False`. If the area has a projected CRS, units are assumed to be in metre. If the area has a geographic CRS, units are assumed to be in degrees. The writer does not verify that the CRS is supported by the CF conventions. One commonly used projected CRS not supported by the CF conventions is the equirectangular projection, such as EPSG 4087.
- By default non-dimensional coordinates (such as scanline timestamps) are prefixed with the corresponding dataset name. This is because they are likely to be different for each dataset. If a non-dimensional coordinate is identical for all datasets, the prefix can be removed by setting `pretty=True`.
- Some dataset names start with a digit, like AVHRR channels 1, 2, 3a, 3b, 4 and 5. This doesn't comply with CF https://cfconventions.org/Data/cf-conventions/cf-conventions-1.7/build/ch02s03.html. These channels are prefixed with `"CHANNEL_"` by default. This can be controlled with the variable *numeric_name_prefix* to *save_datasets*. Setting it to *None* or '' will skip the prefixing.

Grouping

All datasets to be saved must have the same projection coordinates `x` and `y`. If a scene holds datasets with different grids, the CF compliant workaround is to save the datasets to separate files. Alternatively, you can save datasets with common grids in separate netCDF groups as follows:

```
>>> scn.load(['VIS006', 'IR_108', 'HRV'])
>>> scn.save_datasets(writer='cf', datasets=['VIS006', 'IR_108', 'HRV'],
                      filename='seviri_test.nc', exclude_attrs=['raw_metadata'],
                      groups={'visir': ['VIS006', 'IR_108'], 'hrv': ['HRV']})
```

Note that the resulting file will not be fully CF compliant.

Dataset Encoding

Dataset encoding can be specified in two ways:

1) Via the `encoding` keyword argument of `save_datasets`:

```
>>> my_encoding = {
...     'my_dataset_1': {
...         'compression': 'zlib',
...         'complevel': 9,
...         'scale_factor': 0.01,
...         'add_offset': 100,
...         'dtype': np.int16
...     },
...     'my_dataset_2': {
...         'compression': None,
...         'dtype': np.float64
...     }
... }
>>> scn.save_datasets(writer='cf', filename='encoding_test.nc', encoding=my_
↪encoding)
```

2) Via the `encoding` attribute of the datasets in a scene. For example

```
>>> scn['my_dataset'].encoding = {'compression': 'zlib'}
>>> scn.save_datasets(writer='cf', filename='encoding_test.nc')
```

See the xarray encoding documentation for all encoding options.

Note: Chunk-based compression can be specified with the `compression` keyword since

```
netCDF4-1.6.0
libnetcdf-4.9.0
xarray-2022.12.0
```

The `zlib` keyword is deprecated. Make sure that the versions of these modules are all above or all below that reference. Otherwise, compression might fail or be ignored silently.

Attribute Encoding

In the above examples, raw metadata from the HRIT files have been excluded. If you want all attributes to be included, just remove the `exclude_attrs` keyword argument. By default, dict-type dataset attributes, such as the raw metadata, are encoded as a string using json. Thus, you can use json to decode them afterwards:

```
>>> import xarray as xr
>>> import json
>>> # Save scene to nc-file
>>> scn.save_datasets(writer='cf', datasets=['VIS006', 'IR_108'], filename='seviri_test.
↪nc')
>>> # Now read data from the nc-file
>>> ds = xr.open_dataset('seviri_test.nc')
>>> raw_mda = json.loads(ds['IR_108'].attrs['raw_metadata'])
>>> print(raw_mda['RadiometricProcessing']['Level15ImageCalibration']['CalSlope'])
[0.020865   0.0278287  0.0232411  0.00365867 0.00831811 0.03862197
 0.12674432 0.10396091 0.20503568 0.22231115 0.1576069  0.0352385]
```

Alternatively it is possible to flatten dict-type attributes by setting `flatten_attrs=True`. This is more human readable as it will create a separate nc-attribute for each item in every dictionary. Keys are concatenated with underscore separators. The *CalSlope* attribute can then be accessed as follows:

```
>>> scn.save_datasets(writer='cf', datasets=['VIS006', 'IR_108'], filename='seviri_test.
↪nc',
                      flatten_attrs=True)
>>> ds = xr.open_dataset('seviri_test.nc')
>>> print(ds['IR_108'].attrs['raw_metadata_RadiometricProcessing_Level15ImageCalibration_
↪CalSlope'])
[0.020865   0.0278287  0.0232411  0.00365867 0.00831811 0.03862197
 0.12674432 0.10396091 0.20503568 0.22231115 0.1576069  0.0352385]
```

This is what the corresponding `ncdump` output would look like in this case:

```
$ ncdump -h test_seviri.nc
...
IR_108:raw_metadata_RadiometricProcessing_Level15ImageCalibration_CalOffset = -1.064, ...
↪;
IR_108:raw_metadata_RadiometricProcessing_Level15ImageCalibration_CalSlope = 0.021, ...;
IR_108:raw_metadata_RadiometricProcessing_MPEFCalFeedback_AbsCalCoeff = 0.021, ...;
...
```

class satpy.writers.cf_writer.**CFWriter**(*name=None*, *filename=None*, *base_dir=None*, ***kwargs*)

Bases: *Writer*

Writer producing NetCDF/CF compatible datasets.

Initialize the writer object.

Parameters

- **name** (*str*) – A name for this writer for log and error messages. If this writer is configured in a YAML file its name should match the name of the YAML file. Writer names may also appear in output file attributes.
- **filename** (*str*) – Filename to save data to. This filename can and should specify certain python string formatting fields to differentiate between data written to the files. Any attributes

provided by the `.attrs` of a DataArray object may be included. Format and conversion specifiers provided by the `trollsift` package may also be used. Any directories in the provided pattern will be created if they do not exist. Example:

```
{platform_name}_{sensor}_{name}_{start_time:%Y%m%d_%H%M%S}.tif
```

- **base_dir** (*str*) – Base destination directories for all created files.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the *Plugin* class.

static da2cf(*dataarray*, *epoch=None*, *flatten_attrs=False*, *exclude_attrs=None*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*)

Convert the dataarray to something cf-compatible.

Parameters

- **dataarray** (*xr.DataArray*) – The data array to be converted.
- **epoch** (*str*) – Reference time for encoding of time coordinates. If None, the default reference time is defined using *from satpy.cf.coords import EPOCH*
- **flatten_attrs** (*bool*) – If True, flatten dict-type attributes.
- **exclude_attrs** (*list*) – List of dataset attributes to be excluded.
- **include_orig_name** (*bool*) – Include the original dataset name in the netcdf variable attributes.
- **numeric_name_prefix** (*str*) – Prepend dataset name with this if starting with a digit.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, ***kwargs*)

Save the *dataset* to a given *filename*.

save_datasets(*datasets*, *filename=None*, *groups=None*, *header_attrs=None*, *engine=None*, *epoch=None*, *flatten_attrs=False*, *exclude_attrs=None*, *include_lonlats=True*, *pretty=False*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*, ***to_netcdf_kwargs*)

Save the given datasets in one netCDF file.

Note that all datasets (if grouping: in one group) must have the same projection coordinates.

Parameters

- **datasets** (*list*) – List of xr.DataArray to be saved.
- **filename** (*str*) – Output file.
- **groups** (*dict*) – Group datasets according to the given assignment: *{'group_name': ['dataset1', 'dataset2', ...]}*. The group name *None* corresponds to the root of the file, i.e., no group will be created. Warning: The results will not be fully CF compliant!
- **header_attrs** – Global attributes to be included.
- **engine** (*str, optional*) – Module to be used for writing netCDF files. Follows xarray's `to_netcdf()` engine choices with a preference for 'netcdf4'.
- **epoch** (*str, optional*) – Reference time for encoding of time coordinates. If None, the default reference time is defined using *from satpy.cf.coords import EPOCH*.
- **flatten_attrs** (*bool, optional*) – If True, flatten dict-type attributes.
- **exclude_attrs** (*list, optional*) – List of dataset attributes to be excluded.
- **include_lonlats** (*bool, optional*) – Always include latitude and longitude coordinates, even for datasets with area definition.

- **pretty** (*bool, optional*) – Don't modify coordinate names, if possible. Makes the file prettier, but possibly less consistent.
- **include_orig_name** (*bool, optional*) – Include the original dataset name as a variable attribute in the final netCDF.
- **numeric_name_prefix** (*str, optional*) – Prefix to add to each variable with a name starting with a digit. Use '' or None to leave this out.

static update_encoding(*dataset*, *to_netcdf_kwargs*)

Update encoding info (deprecated).

satpy.writers.cf_writer.**_backend_versions_match**()

satpy.writers.cf_writer.**_check_backend_versions**()

Issue warning if backend versions do not match.

satpy.writers.cf_writer.**_get_backend_versions**()

satpy.writers.cf_writer.**_initialize_root_netcdf**(*filename*, *engine*, *header_attrs*, *to_netcdf_kwargs*)

Initialize root empty netCDF.

satpy.writers.cf_writer.**_sanitize_writer_kwargs**(*writer_kwargs*)

Remove satpy-specific kwargs.

satpy.writers.geotiff module

GeoTIFF writer objects for creating GeoTIFF files from *DataArray* objects.

class satpy.writers.geotiff.**GeoTIFFWriter**(*dtype=None*, *tags=None*, ***kwargs*)

Bases: *ImageWriter*

Writer to save GeoTIFF images.

Basic example from Scene:

```
>>> scn.save_datasets(writer='geotiff')
```

By default the writer will use the *Enhancer* class to linear stretch the data (see *Enhancements*). To get Unenhanced images `enhance=False` can be specified which will write a geotiff with the data type of the dataset. The fill value defaults to the the datasets `"_FillValue"` attribute if not `None` and no value is passed to `fill_value` for integer data. In case of float data if `fill_value` is not passed NaN will be used. If a geotiff with a certain datatype is desired for example 32 bit floating point geotiffs:

```
>>> scn.save_datasets(writer='geotiff', dtype=np.float32, enhance=False)
```

To add custom metadata use *tags*:

```
>>> scn.save_dataset(dataset_name, writer='geotiff',
...                  tags={'offset': 291.8, 'scale': -0.35})
```

Images are tiled by default. To create striped TIFF files `tiled=False` can be specified:

```
>>> scn.save_datasets(writer='geotiff', tiled=False)
```

For performance tips on creating geotiffs quickly and making them smaller see the Frequently Asked Questions.

Init the writer.

GDAL_OPTIONS = ('tfw', 'rpb', 'rpctxt', 'interleave', 'tiled', 'blockxsize', 'blockysize', 'nbits', 'compress', 'num_threads', 'predictor', 'discard_lsb', 'sparse_ok', 'jpeg_quality', 'jpegtablesmode', 'zlevel', 'photometric', 'alpha', 'profile', 'bigtiff', 'pixeltype', 'copy_src_overviews', 'blocksize', 'resampling', 'quality', 'level', 'overview_resampling', 'warp_resampling', 'overview_compress', 'overview_quality', 'overview_predictor', 'tiling_scheme', 'zoom_level_strategy', 'target_srs', 'res', 'extent', 'aligned_levels', 'add_alpha')

_get_gdal_options(*kwargs*)

save_image(*img: XRImage, filename: str | None = None, compute: bool = True, dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any] = None, fill_value: int | float | None = None, keep_palette: bool = False, cmap: Colormap | None = None, tags: dict[str, Any] | None = None, overviews: list[int] | None = None, overviews_minsize: int = 256, overviews_resampling: str | None = None, include_scale_offset: bool = False, scale_offset_tags: tuple[str, str] | None = None, colormap_tag: str | None = None, driver: str | None = None, tiled: bool = True, **kwargs*)

Save the image to the given `filename` in geotiff format.

Note this writer requires the `rasterio` library to be installed.

Parameters

- **img** (*xarray.DataArray*) – Data to save to geotiff.
- **filename** (*str*) – Filename to save the image to. Defaults to `filename` passed during writer creation. Unlike the creation `filename` keyword argument, this filename does not get formatted with data attributes.
- **compute** (*bool*) – Compute dask arrays and save the image immediately. If `False` then the return value can be passed to *compute_writer_results()* to do the computation. This is useful when multiple images may share input calculations where dask can benefit from not repeating them multiple times. Defaults to `True` in the writer by itself, but is typically passed as `False` by callers where calculations can be combined.
- **dtype** (*DTypeLike*) – Numpy data type to save the image as. Defaults to 8-bit unsigned integer (`np.uint8`) or the data type of the data to be saved if `enhance=False`. If the `dtype` argument is provided during writer creation then that will be used as the default.
- **fill_value** (*float or int*) – Value to use where data values are NaN/null. If this is specified in the writer configuration file that value will be used as the default.
- **keep_palette** (*bool*) – Save palette/color table to geotiff. To be used with images that were palettized with the "palettize" enhancement. Setting this to `True` will cause the colormap of the image to be written as a "color table" in the output geotiff and the image data values will represent the index values in to that color table. By default, this will use the colormap used in the "palettize" operation. See the `cmap` option for other options. This option defaults to `False` and palettized images will be converted to RGB/A.
- **cmap** (*trollimage.colormap.Colormap or None*) – Colormap to save as a color table in the output geotiff. See `keep_palette` for more information. Defaults to the palette of the provided `img` object. The colormap's range should be set to match the index range of the palette (ex. *cmap.set_range(0, len(colors))*).
- **tags** (*dict*) – Extra metadata to store in geotiff.
- **overviews** (*list*) – The reduction factors of the overviews to include in the image, eg:

```
scn.save_datasets(overviews=[2, 4, 8, 16])
```

If provided as an empty list, then levels will be computed as powers of two until the last level has less pixels than *overviews_minsize*. Default is to not add overviews.

- **overviews_minsize** (*int*) – Minimum number of pixels for the smallest overview size generated when *overviews* is auto-generated. Defaults to 256.
- **overviews_resampling** (*str*) – Resampling method to use when generating overviews. This must be the name of an enum value from `rasterio.enums.Resampling` and only takes effect if the *overviews* keyword argument is provided. Common values include *nearest* (default), *bilinear*, *average*, and many others. See the rasterio documentation for more information.
- **scale_offset_tags** (*Tuple[str, str]*) – If set, include inclusion of scale and offset in the GeoTIFF headers in the GDALMetaData tag. The value of this argument should be a keyword argument `(scale_label, offset_label)`, for example, `("scale", "offset")`, indicating the labels to be used.
- **colormap_tag** (*Optional[str]*) – If set and the image being saved was colorized or palettized then a comma-separated version of the colormap is saved to a custom geotiff tag with the provided name. See `trollimage.colormap.Colormap.to_csv()` for more information.
- **driver** (*Optional[str]*) – Name of GDAL driver to use to save the geotiff. If not specified or None (default) the "GTiff" driver is used. Another common option is "COG" for Cloud Optimized GeoTIFF. See GDAL documentation for more information.
- **tiled** (*bool*) – For performance this defaults to `True`. Pass `False` to created striped TIFF files.
- **include_scale_offset** (*deprecated, bool*) – Deprecated. Use `scale_offset_tags=("scale", "offset")` to include scale and offset tags.

classmethod separate_init_kwargs(*kwargs*)

Separate the init keyword args.

satpy.writers.mitiff module

MITIFF writer objects for creating MITIFF files from *Dataset* objects.

class satpy.writers.mitiff.**MITIFFWriter**(*name=None*, *tags=None*, ***kwargs*)

Bases: *ImageWriter*

Writer to produce MITIFF image files.

Initialize reader with tag and other configuration information.

_add_calibration(*channels*, *cns*, *datasets*, ***kwargs*)

_add_calibration_datasets(*ch*, *datasets*, *reverse_offset*, *reverse_scale*, *decimals*)

_add_corners(*datasets*, *first_dataset*)

_add_palette_info(*datasets*, *palette_unit*, *palette_description*, ***kwargs*)

_add_pixel_sizes(*datasets*, *first_dataset*)

_add_proj4_string(*datasets*, *first_dataset*)

_add_sizes(*datasets*, *first_dataset*)

_append_projection_center(*proj4_string*, *datasets*, *first_dataset*, *x_0*, *y_0*)

_calibrate_data(*dataset*, *calibration*, *min_val*, *max_val*)

_channel_names(*channels*, *cns*, ***kwargs*)

_convert_epsg_to_proj(*proj4_string*, *x_0*)

_generate_intermediate_filename(*gen_filename*)

Replace mitiff ext because pillow doesn't recognise the file type.

_get_dataset_len(*datasets*)

_get_platform_name(*first_dataset*, *translate_platform_name*, *kwargs*)

_make_channel_list(*datasets*, ***kwargs*)

_make_image_description(*datasets*, ***kwargs*)

Generate image description for mitiff.

Satellite: NOAA 18 Date and Time: 06:58 31/05-2016 SatDir: 0 Channels: 6 In this file: 1-VIS0.63 2-VIS0.86 3(3B)-IR3.7 4-IR10.8 5-IR11.5 6(3A)-VIS1.6 Xsize: 4720 Ysize: 5544 Map projection: Stereographic Proj string: +proj=stere +lon_0=0 +lat_0=90 +lat_ts=60 +ellps=WGS84 +towgs84=0,0,0 +units=km +x_0=2526000.000000 +y_0=5806000.000000 TrueLat: 60 N GridRot: 0 Xunit:1000 m Yunit: 1000 m NPX: 0.000000 NPY: 0.000000 Ax: 1.000000 Ay: 1.000000 Bx: -2526.000000 By: -262.000000

Satellite: <satellite name> Date and Time: <HH:MM dd/mm-yyyy> SatDir: 0 Channels: <number of chanels> In this file: <channels names in order> Xsize: <number of pixels x> Ysize: <number of pixels y> Map projection: Stereographic Proj string: <proj4 string with +x_0 and +y_0 which is the positive distance from proj origo to the lower left corner of the image data> TrueLat: 60 N GridRot: 0 Xunit:1000 m Yunit: 1000 m NPX: 0.000000 NPY: 0.000000 Ax: <pixels size x in km> Ay: <pixel size y in km> Bx: <left corner of upper right pixel in km> By: <upper corner of upper right pixel in km>

if palette image write special palette if normal channel write table calibration: Table_calibration: <channel name>, <calibration type>, [<unit>], <no of bits of data>, [<calibration values space separated>]nn

_reorder_channels(*datasets*, ***kwargs*)

_save_as_enhanced(*datasets*, *tmp_gen_filename*, ***kwargs*)

Save datasets as an enhanced RGB image.

_save_as_palette(*datasets*, *tmp_gen_filename*, *tiffinfo*, ***kwargs*)

_save_datasets_as_mitiff(*datasets*, *image_description*, *gen_filename*, ***kwargs*)

Put all together and save as a tiff file.

Include the special tags making it a mitiff file.

_save_single_dataset(*datasets*, *cns*, *tmp_gen_filename*, *tiffinfo*, *kwargs*)

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *compute=True*, ***kwargs*)

Save single dataset as mitiff file.

save_datasets(*datasets*, *filename=None*, *fill_value=None*, *compute=True*, ***kwargs*)

Save all datasets to one or more files.

save_image()

Save dataset as an image array.

satpy.writers.mitiff.**_adjust_kwargs**(*dataset*, *kwargs*)

satpy.writers.ninjogeotiff module

Writer for GeoTIFF images with tags for the NinJo visualization tool.

Starting with NinJo 7, NinJo is able to read standard GeoTIFF images, with required metadata encoded as a set of XML tags in the GDALMetadata TIFF tag. Each of the XML tags must be prepended with `'NINJO_'`. For NinJo delivery, these GeoTIFF files supersede the old NinJoTIFF format. The *NinJoGeoTIFFWriter* therefore supersedes the old Satpy NinJoTIFF writer and the pyninjotiff package.

The reference documentation for valid NinJo tags and their meaning is contained in NinJoPedia. Since this page is not in the public web, there is a (possibly outdated) mirror.

There are some user-facing differences between the old NinJoTIFF writer and the new NinJoGeoTIFF writer. Most notably, keyword arguments that correspond to tags directly passed by the user are now identical, including case, to how they will be written to the GDALMetaData and interpreted by NinJo. That means some keyword arguments have changed, such as summarised in this table:

Table 5: Migrating to NinJoGeoTIFF, keyword arguments for the writer

ninjotiff (old)	ninjogeotiff (new)	Notes
`chan_id`	`ChannelID`	mandatory
`data_cat`	`DataType`	mandatory
`physic_unit`	`PhysicUnit`	mandatory
`physic_val`	`PhysicValue`	mandatory
`sat_id`	`SatelliteNameID`	mandatory
`data_source`	`DataSource`	optional

Moreover, two keyword arguments are no longer supported because their functionality has become redundant. This applies to `ch_min_measurement_unit` and `ch_max_measurement_unit`. Instead, pass those values in source units to the *stretch()* enhancement with the `min_stretch` and `max_stretch` arguments.

For images where the pixel value corresponds directly to a physical value, NinJo has a functionality to read the corresponding quantity (example: brightness temperature or reflectance). To make this possible, the writer adds the tags `Gradient` and `AxisIntercept`. Those tags are added if and only if the image has mode `L` or `LA` and `PhysicUnit` is not set to `"N/A"`. In other words, to suppress those tags for images with mode `L` or `LA` (for example, for the composite `vis_with_ir`, where the physical interpretation of individual pixels is lost), one should set `PhysicUnit` to `"N/A"`, `"n/a"`, `"1"`, or `""` (empty string).

class satpy.writers.ninjogeotiff.**NinJoGeoTIFFWriter**(*dtype=None*, *tags=None*, ***kwargs*)

Bases: *GeoTIFFWriter*

Writer for GeoTIFFs with NinJo tags.

This writer is experimental. API may be subject to change.

For information, see module docstring and documentation for *save_image()*.

Init the writer.

_check_include_scale_offset(*image*, *unit*)

Check if scale-offset tags should be included.

_fix_units(*image*, *quantity*, *unit*)

Adapt units between °C and K.

This will return a new XRImage, to make sure the old data and enhancement history aren't touched.

save_image(*image*, *filename=None*, *fill_value=None*, *compute=True*, *keep_palette=False*, *cmap=None*, *overviews=None*, *overviews_minsize=256*, *overviews_resampling=None*, *tags=None*, *config_files=None*, ***, *ChannelID*, *DataType*, *PhysicUnit*, *PhysicValue*, *SatelliteNameID*, ***kwargs*)

Save image along with NinJo tags.

Save image along with NinJo tags. Interface as for GeoTIFF, except NinJo expects some additional tags. Those tags will be prepended with `ninjo_` and added as GDALMetaData.

Writing such images requires trollimage 1.16 or newer.

Importing such images with NinJo requires NinJo 7 or newer.

Parameters

- **image** (*XRImage*) – Image to save.
- **filename** (*str*) – Where to save the file.
- **fill_value** (*int*) – Which pixel value is fill value?
- **compute** (*bool*) – To compute or not to compute, that is the question.
- **keep_palette** (*bool*) – As for parent GeoTIFF *save_image()*.
- **cmap** (*trollimage.colormap.Colormap*) – As for parent *save_image()*.
- **overviews** (*list*) – As for *save_image()*.
- **overviews_minsize** (*int*) – As for *save_image()*.
- **overviews_resampling** (*str*) – As for *save_image()*.
- **tags** (*dict*) – Extra (not NinJo) tags to add to GDAL MetaData
- **config_files** (*Any*) – Not directly used by this writer, supported for compatibility with other writers.

Remaining keyword arguments are either passed as GDAL options, if contained in `self.GDAL_OPTIONS`, or they are passed to *NinJoTagGenerator*, which will include them as NinJo tags in GDALMetadata. Supported tags are defined in `NinJoTagGenerator.optional_tags`. The meaning of those (and other) tags are defined in the NinJo documentation (see module documentation for a link to NinJoPedia). The following tags are mandatory and must be provided as keyword arguments:

ChannelID (int)
NinJo Channel ID

DataType (int)
NinJo Data Type

SatelliteNameID (int)
NinJo Satellite ID

PhysicUnit (str)
NinJo label for unit (example: "C"). If PhysicValue is set to "Temperature", PhysicUnit is set to "C", but data attributes incidate the data have unit "K", then the writer will adapt the header `ninjo_AxisIntercept` such that data are interpreted in units of "C". If PhysicUnit is set to "N/A", no AxisIntercept and Gradient tags will be written.

PhysicValue (str)
NinJo label for quantity (example: "temperature")

`scale_offset_tag_names = ('ninjo_Gradient', 'ninjo_AxisIntercept')`

class `satpy.writers.ninjogeotiff.NinJoTagGenerator`(*image*, *fill_value*, *filename*, ***kwargs*)

Bases: `object`

Class to collect NinJo tags.

This class is used by `NinJoGeoTIFFWriter` to collect NinJo tags. Most end-users will not need to create instances of this class directly.

Tags are gathered from three sources:

- Fixed tags, contained in the attribute `fixed_tags`. The value of those tags is hardcoded and never changes.
- Tags passed by the user, contained in the attribute `passed_tags`. Those tags must be passed by the user as arguments to the writer, which will pass them on when instantiating this class.
- Tags calculated from data and metadata. Those tags are defined in the attribute `dynamic_tags`. They are either calculated from image data, from image metadata, or from arguments passed by the user to the writer.

Some tags are mandatory (defined in `mandatory_tags`). All tags that are not mandatory are optional. By default, optional tags are generated if and only if the required information is available.

Initialise tag generator.

Parameters

- **image** (`trollimage.xrimage.XRImage`) – XRImage for which NinJo tags should be calculated.
- **fill_value** (*int*) – Fill value corresponding to image.
- **filename** (*str*) – Filename to be written.
- ****kwargs** – Any additional tags to be included as-is.

`_epoch = datetime.datetime(1970, 1, 1, 0, 0, tzinfo=datetime.timezone.utc)`

```
dynamic_tags = {'CentralMeridian': 'central_meridian', 'ColorDepth':
'color_depth', 'CreationDateID': 'creation_date_id', 'DateID': 'date_id',
'EarthRadiusLarge': 'earth_radius_large', 'EarthRadiusSmall':
'earth_radius_small', 'FileName': 'filename', 'MaxGrayValue': 'max_gray_value',
'MinGrayValue': 'min_gray_value', 'Projection': 'projection',
'ReferenceLatitude1': 'ref_lat_1', 'TransparentPixel': 'transparent_pixel',
'XMaximum': 'xmaximum', 'YMaximum': 'ymaximum'}
```

```
fixed_tags = {'HeaderVersion': 2, 'Magic': 'NINJO', 'XMinimum': 1, 'YMinimum':
1}
```

get_all_tags()

Get a dictionary with all tags for NinJo.

get_central_meridian()

Calculate central meridian.

get_color_depth()

Return the color depth.

get_creation_date_id()

Calculate the creation date ID.

That's seconds since UNIX Epoch for the time the image is created.

get_date_id()

Calculate the date ID.

That's seconds since UNIX Epoch for the time corresponding to the satellite image start of measurement time.

get_earth_radius_large()

Return the Earth semi-major axis in metre.

get_earth_radius_small()

Return the Earth semi-minor axis in metre.

get_filename()

Return the filename.

get_max_gray_value()

Calculate maximum gray value.

get_meridian_east()

Get the easternmost longitude of the area.

Currently not implemented. In pyninjotiff it was implemented but the answer was incorrect.

get_meridian_west()

Get the westernmost longitude of the area.

Currently not implemented. In pyninjotiff it was implemented but the answer was incorrect.

get_min_gray_value()

Calculate minimum gray value.

get_projection()

Get NinJo projection string.

From the documentation, valid values are:

- NPOL/SPOL: polar-sterographic North/South
- PLAT: „Plate Carrée", equirectangular projection
- MERC: Mercator projection

Derived from AreaDefinition.

get_ref_lat_1()

Get reference latitude one.

Derived from area definition.

get_ref_lat_2()

Get reference latitude two.

This is not implemented and never was correctly implemented in pyninjotiff either. It doesn't appear to be used by NinJo.

get_tag(*tag*)

Get value for NinJo tag.

get_transparent_pixel()

Get the transparent pixel value, also known as the fill value.

When the no fill value is defined (value *None*), such as for RGBA or LA images, returns -1, in accordance with the file format specification.

get_xmaximum()

Get the maximum value of x, i.e. the meridional extent of the image in pixels.

get_ymaximum()

Get the maximum value of y, i.e. the zonal extent of the image in pixels.

mandatory_tags = {'AxisIntercept', 'ChannelID', 'ColorDepth', 'CreationDateID', 'DataType', 'DateID', 'Gradient', 'HeaderVersion', 'MaxGrayValue', 'MinGrayValue', 'PhysicUnit', 'PhysicValue', 'Projection', 'SatelliteNameID', 'SatelliteNumber', 'TransparentPixel', 'XMaximum', 'XMinimum', 'YMaximum', 'YMinimum'}

optional_tags = {'AOSAzimuth', 'Altitude', 'CentralMeridian', 'ColorTable', 'DataSource', 'Description', 'EarthRadiusLarge', 'EarthRadiusSmall', 'GeoLatitude', 'GeoLongitude', 'GeodeticDate', 'IsAtmosphereCorrected', 'IsBlackLinesCorrection', 'IsCalibrated', 'IsNormalized', 'IsValueTableAvailable', 'LOSAzimuth', 'MaxElevation', 'MeridianEast', 'MeridianWest', 'OriginalHeader', 'OverFlightTime', 'OverflightDirection', 'ReferenceLatitude1', 'ReferenceLatitude2', 'ValueTableFloatField'}

passed_tags = {'ChannelID', 'DataType', 'PhysicUnit', 'PhysicValue', 'SatelliteNameID'}

postponed_tags = {'AxisIntercept', 'Gradient'}

satpy.writers.ninjotiff module

Writer for TIFF images compatible with the NinJo visualization tool (NinjoTIFFs).

NinjoTIFFs can be color images or monochromatic. For monochromatic images, the physical units and scale and offsets to retrieve the physical values are provided. Metadata is also recorded in the file.

In order to write ninjotiff files, some metadata needs to be provided to the writer. Here is an example on how to write a color image:

```
chn = "airmass"
ninjoRegion = load_area("areas.def", "nrEURO3km")

filenames = glob("data/*__")
global_scene = Scene(reader="hrit_msg", filenames=filenames)
global_scene.load([chn])
local_scene = global_scene.resample(ninjoRegion)
local_scene.save_dataset(chn, filename="airmass.tif", writer='ninjotiff',
                         sat_id=6300014,
                         chan_id=6500015,
                         data_cat='GPRN',
                         data_source='EUMCAST',
                         nbits=8)
```

Here is an example on how to write a color image:

```
chn = "IR_108"
ninjoRegion = load_area("areas.def", "nrEURO3km")

filenames = glob("data/*__")
global_scene = Scene(reader="hrit_msg", filenames=filenames)
global_scene.load([chn])
local_scene = global_scene.resample(ninjoRegion)
local_scene.save_dataset(chn, filename="msg.tif", writer='ninjotiff',
                         sat_id=6300014,
                         chan_id=900015,
                         data_cat='GORN',
                         data_source='EUMCAST',
                         physic_unit='K',
                         nbits=8)
```

The metadata to provide to the writer can also be stored in a configuration file (see pyninjotiff), so that the previous example can be rewritten as:

```
chn = "IR_108"
ninjoRegion = load_area("areas.def", "nrEURO3km")

filenames = glob("data/*__")
global_scene = Scene(reader="hrit_msg", filenames=filenames)
global_scene.load([chn])
local_scene = global_scene.resample(ninjoRegion)
local_scene.save_dataset(chn, filename="msg.tif", writer='ninjotiff',
                         # ninjo product name to look for in .cfg file
                         ninjo_product_name="IR_108",
                         # custom configuration file for ninjo tiff products
                         # if not specified PPP_CONFIG_DIR is used as config file directory
                         ninjo_product_file="/config_dir/ninjotiff_products.cfg")
```

class satpy.writers.ninjotiff.**NinjoTIFFWriter**(*tags=None*, ***kwargs*)

Bases: *ImageWriter*

Writer for NinjoTiff files.

Inititalize the writer.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *compute=True*, *convert_temperature_units=True*, ***kwargs*)

Save a dataset to ninjotiff format.

This calls *save_image* in turn, but first preforms some unit conversion if necessary and desired. Unit conversion can be suppressed by passing `convert_temperature_units=False`.

save_image(*img*, *filename=None*, *compute=True*, ***kwargs*)

Save the image to the given *filename* in ninjotiff format.

satpy.writers.ninjotiff.**convert_units**(*dataset*, *in_unit*, *out_unit*)

Convert units of *dataset*.

Convert dataset units for the benefit of writing NinJoTIFF. The main background here is that NinJoTIFF would like brightness temperatures in °C, but satellite data files are in K. For simplicity of implementation, this function can only convert from K to °C.

This function will convert input data from K to °C and write the new unit in the `"units"` attribute. When output and input units are equal, it returns the input dataset.

Parameters

- **dataset** (*xarray DataArray*) – Dataarray for which to convert the units.
- **in_unit** (*str*) – Unit for input data.
- **out_unit** (*str*) – Unit for output data.

Returns
dataset, possibly with new units.

satpy.writers.simple_image module

Generic PIL/Pillow image format writer.

class satpy.writers.simple_image.**PillowWriter**(***kwargs*)

Bases: *ImageWriter*

Generic PIL image format writer.

Initialize image writer plugin.

save_image(*img*, *filename=None*, *compute=True*, ***kwargs*)

Save Image object to a given `filename`.

Parameters

- **img** (*trollimage.xrimage.XRImage*) – Image object to save to disk.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.
- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a *dask.delayed.Delayed* object or tuple of (source, target). See the return values below for more information.
- ****kwargs** – Keyword arguments to pass to the images *save* method.

Returns
Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a *dask.delayed.Delayed* object or running *dask.array.store*. If *compute* is *False* then the returned value is either a *dask.delayed.Delayed* object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to *dask.array.store*. If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

satpy.writers.utils module

Writer utilities.

satpy.writers.utils.**flatten_dict**(*d*, *parent_key=''*, *sep='_'*)

Flatten a nested dictionary.

Based on https://stackoverflow.com/a/6027615/5703449

Module contents

Shared objects of the various writer classes.

For now, this includes enhancement configuration utilities.

class satpy.writers.**DecisionTree**(*decision_dicts*, *match_keys*, *multival_keys=None*)

Bases: object

Structure to search for nearest match from a set of parameters.

This class is used to find the best configuration section by matching a set of attributes. The provided dictionary contains a mapping of "section name" to "decision" dictionaries. Each decision dictionary contains the attributes that will be used for matching plus any additional keys that could be useful when matched. This class will search these decisions and return the one with the most matching parameters to the attributes passed to the *find_match()* method.

Note that decision sections are provided as a dict instead of a list so that they can be overwritten or updated by doing the equivalent of a `current_dicts.update(new_dicts)`.

Examples

Decision sections are provided as a dictionary of dictionaries. The returned match will be the first result found by searching provided *match_keys* in order.

```
decisions = {
    'first_section': {
        'a': 1,
        'b': 2,
        'useful_key': 'useful_value',
    },
    'second_section': {
        'a': 5,
        'useful_key': 'other_useful_value1',
    },
    'third_section': {
        'b': 4,
        'useful_key': 'other_useful_value2',
    },
}
tree = DecisionTree(decisions, ('a', 'b'))
tree.find_match(a=5, b=2)  # second_section dict
tree.find_match(a=1, b=2)  # first_section dict
tree.find_match(a=5, b=4)  # second_section dict
tree.find_match(a=3, b=2)  # no match
```

Init the decision tree.

Parameters

- **decision_dicts** (*dict*) – Dictionary of dictionaries. Each sub-dictionary contains key/value pairs that can be matched from the *find_match* method. Sub-dictionaries can include additional keys outside the `match_keys` provided to act as the "result" of a query. The keys of the root dict are arbitrary.
- **match_keys** (*list*) – Keys of the provided dictionary to use for matching.

- **multival_keys** (*list*) – Keys of *match_keys* that can be provided as multiple values. A multi-value key can be specified as a single value (typically a string) or a set. If a set, it will be sorted and converted to a tuple and then used for matching. When querying the tree, these keys will be searched for exact multi-value results (the sorted tuple) and if not found then each of the values will be searched individually in alphabetical order.

_build_tree(*conf*)

Build the tree.

Create a tree structure of dicts where each level represents the possible matches for a specific `match_key`. When finding matches we will iterate through the tree matching each key that we know about. The last dict in the "tree" will contain the configure section whose match values led down that path in the tree.

See *DecisionTree.find_match()* for more information.

static _convert_query_val_to_hashable(*query_val*)

_find_match(*curr_level*, *remaining_match_keys*, *query_dict*)

Find a match.

_find_match_if_known(*curr_level*, *remaining_match_keys*, *query_dict*)

_get_query_values(*query_dict*, *curr_match_key*)

add_config_to_tree(**decision_dicts*)

Add a configuration to the tree.

any_key = None

find_match(***query_dict*)

Find a match.

Recursively search through the tree structure for a path that matches the provided match parameters.

class satpy.writers.**EnhancementDecisionTree**(**decision_dicts*, ***kwargs*)

Bases: *DecisionTree*

The enhancement decision tree.

Init the decision tree.

add_config_to_tree(**decision_dict*)

Add configuration to tree.

find_match(***query_dict*)

Find a match.

class satpy.writers.**Enhancer**(*enhancement_config_file=None*)

Bases: `object`

Helper class to get enhancement information for images.

Initialize an Enhancer instance.

Parameters

enhancement_config_file – The enhancement configuration to apply, False to leave as is.

add_sensor_enhancements(*sensor*)

Add sensor-specific enhancements.

apply(*img*, ***info*)

Apply the enhancements.

get_sensor_enhancement_config(*sensor*)

Get the sensor-specific config.

class satpy.writers.**ImageWriter**(*name=None*, *filename=None*, *base_dir=None*, *enhance=None*, ***kwargs*)

Bases: *Writer*

Base writer for image file formats.

Initialize image writer object.

Parameters

- **name** (*str*) – A name for this writer for log and error messages. If this writer is configured in a YAML file its name should match the name of the YAML file. Writer names may also appear in output file attributes.
- **filename** (*str*) – Filename to save data to. This filename can and should specify certain python string formatting fields to differentiate between data written to the files. Any attributes provided by the `.attrs` of a DataArray object may be included. Format and conversion specifiers provided by the `trollsift` package may also be used. Any directories in the provided pattern will be created if they do not exist. Example:

```
{platform_name}_{sensor}_{name}_{start_time:%Y%m%d_%H%M%S}.tif
```

- **base_dir** (*str*) – Base destination directories for all created files.
- **enhance** (*bool or Enhancer*) – Whether to automatically enhance data to be more visually useful and to fit inside the file format being saved to. By default, this will default to using the enhancement configuration files found using the default *Enhancer* class. This can be set to *False* so that no enhancments are performed. This can also be an instance of the *Enhancer* class if further custom enhancement is needed.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the `Writer` base class.

Changed in version 0.10: Deprecated *enhancement_config_file* and 'enhancer' in favor of *enhance*. Pass an instance of the *Enhancer* class to *enhance* instead.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *overlay=None*, *decorate=None*, *compute=True*, *units=None*, ***kwargs*)

Save the `dataset` to a given `filename`.

This method creates an enhanced image using *get_enhanced_image()*. The image is then passed to *save_image()*. See both of these functions for more details on the arguments passed to this method.

save_image(*img: XRImage*, *filename: str | None = None*, *compute: bool = True*, ***kwargs*)

Save Image object to a given `filename`.

Parameters

- **img** (*trollimage.xrimage.XRImage*) – Image object to save to disk.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.
- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a Dask Delayed object or tuple of (source, target). See the return values below for more information.
- ****kwargs** – Other keyword arguments to pass to this writer.

Returns

Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a Dask Delayed object or running `dask.array.store()`. If *compute* is *False* then the returned value is either a Dask Delayed object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to `dask.array.store()`. If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

classmethod separate_init_kwargs(*kwargs*)

Separate the init kwargs.

class satpy.writers.**Writer**(*name=None*, *filename=None*, *base_dir=None*, ***kwargs*)

Bases: *Plugin*, *DataDownloadMixin*

Base Writer class for all other writers.

A minimal writer subclass should implement the *save_dataset* method.

Initialize the writer object.

Parameters

- **name** (*str*) – A name for this writer for log and error messages. If this writer is configured in a YAML file its name should match the name of the YAML file. Writer names may also appear in output file attributes.
- **filename** (*str*) – Filename to save data to. This filename can and should specify certain python string formatting fields to differentiate between data written to the files. Any attributes provided by the `.attrs` of a DataArray object may be included. Format and conversion specifiers provided by the `trollsift` package may also be used. Any directories in the provided pattern will be created if they do not exist. Example:

```
{platform_name}_{sensor}_{name}_{start_time:%Y%m%d_%H%M%S}.tif
```

- **base_dir** (*str*) – Base destination directories for all created files.
- **kwargs** (*dict*) – Additional keyword arguments to pass to the *Plugin* class.

static _prepare_metadata_for_filename_formatting(*attrs*)

create_filename_parser(*base_dir*)

Create a `trollsift.parser.Parser` object for later use.

get_filename(***kwargs*)

Create a filename where output data will be saved.

Parameters

kwargs (*dict*) – Attributes and other metadata to use for formatting the previously provided *filename*.

save_dataset(*dataset*, *filename=None*, *fill_value=None*, *compute=True*, *units=None*, ***kwargs*)

Save the `dataset` to a given `filename`.

This method must be overloaded by the subclass.

Parameters

- **dataset** (*xarray.DataArray*) – Dataset to save using this writer.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. If not provided then *filename* which can be provided to the init method will be used and formatted by dataset attributes.

- **fill_value** (*int or float*) – Replace invalid values in the dataset with this fill value if applicable to this writer.
- **compute** (*bool*) – If *True* (default), compute and save the dataset. If *False* return either a Dask Delayed object or tuple of (source, target). See the return values below for more information.
- **units** (*str or None*) – If not None, will convert the dataset to the given unit using pint-xarray before saving. Default is not to do any conversion.
- ****kwargs*** – Other keyword arguments for this particular writer.

Returns

Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a Dask Delayed object or running `dask.array.store()`. If *compute* is *False* then the returned value is either a Dask Delayed object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to `dask.array.store()`. If target is provided the caller is responsible for calling *target.close()* if the target has this method.

save_datasets(*datasets*, *compute=True*, ***kwargs*)

Save all datasets to one or more files.

Subclasses can use this method to save all datasets to one single file or optimize the writing of individual datasets. By default this simply calls *save_dataset* for each dataset provided.

Parameters

- **datasets** (*iterable*) – Iterable of *xarray.DataArray* objects to save using this writer.
- **compute** (*bool*) – If *True* (default), compute all the saves to disk. If *False* then the return value is either a Dask Delayed object or two lists to be passed to a `dask.array.store()` call. See return values below for more details.
- ****kwargs*** – Keyword arguments to pass to *save_dataset*. See that documentation for more details.

Returns

Value returned depends on *compute* keyword argument. If *compute* is *True* the value is the result of either a `dask.array.store()` operation or a Dask Delayed compute, typically this is *None*. If *compute* is *False* then the result is either a Dask Delayed object that can be computed with *delayed.compute()* or a two element tuple of sources and targets to be passed to `dask.array.store()`. If *targets* is provided then it is the caller's responsibility to close any objects that have a "close" method.

classmethod separate_init_kwargs(*kwargs*)

Help separating arguments between init and save methods.

Currently the *Scene* is passed one set of arguments to represent the Writer creation and saving steps. This is not preferred for Writer structure, but provides a simpler interface to users. This method splits the provided keyword arguments between those needed for initialization and those needed for the `save_dataset` and `save_datasets` method calls.

Writer subclasses should try to prefer keyword arguments only for the save methods only and leave the init keyword arguments to the base classes when possible.

satpy.writers.**_burn_overlay**(*img*, *image_metadata*, *area*, *cw_*, *overlays*)

Burn the overlay in the image array.

satpy.writers.**_create_overlays_dict**(*color*, *width*, *grid*, *level_coast*, *level_borders*)

Fill in the overlays dict.

satpy.writers.**_determine_mode**(*dataset*)

satpy.writers.**add_decorate**(*orig*, *fill_value=None*, ***decorate*)

Decorate an image with text and/or logos/images.

This call adds text/logos in order as given in the input to keep the alignment features available in pydecorate.

An example of the decorate config:

```
decorate = {
    'decorate': [
        {'logo': {'logo_path': <path to a logo>, 'height': 143, 'bg': 'white', 'bg_
↪opacity': 255}},
        {'text': {'txt': start_time_txt,
                  'align': {'top_bottom': 'bottom', 'left_right': 'right'},
                  'font': <path to ttf font>,
                  'font_size': 22,
                  'height': 30,
                  'bg': 'black',
                  'bg_opacity': 255,
                  'line': 'white'}}
    ]
}
```

Any numbers of text/logo in any order can be added to the decorate list, but the order of the list is kept as described above.

Note that a feature given in one element, eg. bg (which is the background color) will also apply on the next elements unless a new value is given.

align is a special keyword telling where in the image to start adding features, top_bottom is either top or bottom and left_right is either left or right.

satpy.writers.**add_logo**(*orig*, *dc*, *img*, *logo*)

Add logos or other images to an image using the pydecorate package.

All the features of pydecorate's `add_logo` are available. See documentation of Welcome to the Pydecorate documentation! for more info.

satpy.writers.**add_overlay**(*orig_img*, *area*, *coast_dir*, *color=None*, *width=None*, *resolution=None*, *level_coast=None*, *level_borders=None*, *fill_value=None*, *grid=None*, *overlays=None*)

Add coastline, political borders and grid(graticules) to image.

Uses `color` for feature colors where `color` is a 3-element tuple of integers between 0 and 255 representing (R, G, B).

> **Warning:** This function currently loses the data mask (alpha band).

`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

`grid` is a dictionary with key values as documented in detail in pycoast

eg. overlay={'grid': {'major_lonlat': (10, 10),
'write_text': False, 'outline': (224, 224, 224), 'width': 0.5}}

Here major_lonlat is plotted every 10 deg for both longitude and latitude, no labels for the grid lines are plotted, the color used for the grid lines is light gray, and the width of the gratucules is 0.5 pixels.

For grid if aggdraw is used, font option is mandatory, if not `write_text` is set to False:

```
font = aggdraw.Font('black', '/usr/share/fonts/truetype/msttcorefonts/Arial.ttf',
                    opacity=127, size=16)
```

`satpy.writers.add_scale(orig, dc, img, scale)`

Add scale to an image using the pydecorate package.

All the features of pydecorate's `add_scale` are available. See documentation of Welcome to the Pydecorate documentation! for more info.

`satpy.writers.add_text(orig, dc, img, text)`

Add text to an image using the pydecorate package.

All the features of pydecorate's `add_text` are available. See documentation of Welcome to the Pydecorate documentation! for more info.

`satpy.writers.available_writers(as_dict=False)`

Available writers based on current configuration.

Parameters
as_dict (*bool*) – Optionally return writer information as a dictionary. Default: False

Returns: List of available writer names. If *as_dict* is *True* then
a list of dictionaries including additionally writer information is returned.

`satpy.writers.compute_writer_results(results)`

Compute all the given dask graphs *results* so that the files are saved.

Parameters
results (*iterable*) – Iterable of dask graphs resulting from calls to *scn.save_datasets(..., compute=False)*

`satpy.writers.configs_for_writer(writer=None)`

Generate writer configuration files for one or more writers.

Parameters
writer (*Optional[str]*) – Yield configs only for this writer

Returns: Generator of lists of configuration files

`satpy.writers.get_enhanced_image(dataset, enhance=None, overlay=None, decorate=None, fill_value=None)`

Get an enhanced version of *dataset* as an `XRImage` instance.

Parameters

- **dataset** (*xarray.DataArray*) – Data to be enhanced and converted to an image.
- **enhance** (*bool or Enhancer*) – Whether to automatically enhance data to be more visually useful and to fit inside the file format being saved to. By default, this will default to using the enhancement configuration files found using the default *Enhancer* class. This can

be set to *False* so that no enhancments are performed. This can also be an instance of the *Enhancer* class if further custom enhancement is needed.

- **overlay** (*dict*) – Options for image overlays. See *add_overlay()* for available options.
- **decorate** (*dict*) – Options for decorating the image. See *add_decorate()* for available options.
- **fill_value** (*int or float*) – Value to use when pixels are masked or invalid. Default of *None* means to create an alpha channel. See `finalize()` for more details. Only used when adding overlays or decorations. Otherwise it is up to the caller to "finalize" the image before using it except if calling `img.show()` or providing the image to a writer as these will finalize the image.

`satpy.writers.`**`group_results_by_output_file`**(*sources*, *targets*)

Group results by output file.

For writers that return sources and targets for `compute=False`, split the results by output file.

When not only the data but also GeoTIFF tags are dask arrays, then `save_datasets(..., compute=False)`` returns a tuple of flat lists, where the second list consists of a mixture of `RIOTag` and `RIODataset` objects (from trollimage). In some cases, we may want to get a seperate delayed object for each file; for example, if we want to add a wrapper to do something with the file as soon as it's finished. This function unflattens the flat lists into a list of (src, target) tuples.

For example, to close files as soon as computation is completed:

```
>>> @dask.delayed
>>> def closer(obj, targs):
...     for targ in targs:
...         targ.close()
...     return obj
>>> (srcs, targs) = sc.save_datasets(writer="ninjogeotiff", compute=False, **ninjo_
↪tags)
>>> for (src, targ) in group_results_by_output_file(srcs, targs):
...     delayed_store = da.store(src, targ, compute=False)
...     wrapped_store = closer(delayed_store, targ)
...     wrapped.append(wrapped_store)
>>> compute_writer_results(wrapped)
```

In the wrapper you can do other useful tasks, such as writing a log message or moving files to a different directory.

> **Warning:** Adding a callback may impact runtime and RAM. The pattern or cause is unclear. Tests with FCI data show that for resampling with high RAM use (from around 15 GB), runtime increases when a callback is added. Tests with ABI or low RAM consumption rather show a decrease in runtime. More information, see these GitHub comments Users who find out more are encouraged to contact the Satpy developers with clues.

Parameters

- **sources** – List of sources (typically dask.array) as returned by `Scene.save_datasets()`.
- **targets** – List of targets (should be `RIODataset` or `RIOTag`) as returned by `Scene.save_datasets()`.

Returns

List of `Tuple(List[sources], List[targets])` with a length equal to the number of output files planned to be written by `Scene.save_datasets()`.

satpy.writers.**load_writer**(*writer*, ***writer_kwargs*)

Find and load writer *writer* in the available configuration files.

satpy.writers.**load_writer_configs**(*writer_configs*, ***writer_kwargs*)

Load the writer from the provided *writer_configs*.

satpy.writers.**read_writer_config**(*config_files*, *loader=<class 'yaml.loader.UnsafeLoader'>*)

Read the writer *config_files* and return the info extracted.

satpy.writers.**show**(*dataset*, ***kwargs*)

Display the dataset as an image.

satpy.writers.**split_results**(*results*)

Split results.

Get sources, targets and delayed objects to separate lists from a list of results collected from (multiple) writer(s).

satpy.writers.**to_image**(*dataset*)

Convert `dataset` into a `XRImage` instance.

Convert the `dataset` into an instance of the `XRImage` class. This function makes no other changes. To get an enhanced image, possibly with overlays and decoration, see *`get_enhanced_image()`*.

Parameters

dataset (*`xarray.DataArray`*) – Data to be converted to an image.

Returns

Instance of `XRImage`.

Submodules

satpy._compat module

Backports and compatibility fixes for satpy.

satpy._config module

Satpy Configuration directory and file handling.

satpy._config.**_entry_point_module**(*entry_point*)

satpy._config.**cached_entry_point**(*group_name: str*) → Iterable[EntryPoint]

Return entry_point for specified `group`.

This is a dummy proxy to allow caching and provide compatibility between versions of Python and importlib_metadata.

satpy._config.**config_search_paths**(*filename*, *search_dirs=None*, ***kwargs*)

Get series of configuration base paths where Satpy configs are located.

satpy._config.**get_config_path**(*filename*)

Get the path to the highest priority version of a config file.

satpy._config.**get_config_path_safe**()

Get 'config_path' and check for proper 'list' type.

satpy._config.**get_entry_points_config_dirs**(*group_name: str*, *include_config_path: bool = True*) → list[str]

Get the config directories for all entry points of given name.

satpy._config.**glob_config**(*pattern*, *search_dirs=None*)

Return glob results for all possible configuration locations.

Note: This method does not check the configuration "base" directory if the pattern includes a subdirectory.

This is done for performance since this is usually used to find *all* configs for a certain component.

satpy._scene_converters module

Helper functions for converting the Scene object to some other object.

satpy._scene_converters.**_get_dataarrays_from_identifiers**(*scn*, *identifiers*)

Return a list of DataArray based on a single or list of identifiers.

An identifier can be a DataID or a string with name of a valid DataID.

satpy._scene_converters.**to_xarray**(*scn*, *datasets=None*, *header_attrs=None*, *exclude_attrs=None*, *flatten_attrs=False*, *pretty=True*, *include_lonlats=True*, *epoch=None*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*)

Merge all xr.DataArray(s) of a satpy.Scene to a CF-compliant xarray object.

If all Scene DataArrays are on the same area, it returns an xr.Dataset. If Scene DataArrays are on different areas, currently it fails, although in future we might return a DataTree object, grouped by area.

Parameters

- **scn** (*satpy.Scene*) – Satpy Scene.
- **datasets** (*iterable, optional*) – List of Satpy Scene datasets to include in the output xr.Dataset. Elements can be string name, a wavelength as a number, a DataID, or DataQuery object. If None (the default), it includes all loaded Scene datasets.
- **header_attrs** – Global attributes of the output xr.Dataset.
- **epoch** (*str, optional*) – Reference time for encoding the time coordinates (if available). Format example: "seconds since 1970-01-01 00:00:00". If None, the default reference time is retrieved using "from satpy.cf_writer import EPOCH".
- **flatten_attrs** (*bool, optional*) – If True, flatten dict-type attributes.
- **exclude_attrs** (*list, optional*) – List of xr.DataArray attribute names to be excluded.
- **include_lonlats** (*bool, optional*) – If True, includes 'latitude' and 'longitude' coordinates. If the 'area' attribute is a SwathDefinition, it always includes latitude and longitude coordinates.
- **pretty** (*bool, optional*) – Don't modify coordinate names, if possible. Makes the file prettier, but possibly less consistent.
- **include_orig_name** (*bool, optional*) – Include the original dataset name as a variable attribute in the xr.Dataset.
- **numeric_name_prefix** (*str, optional*) – Prefix to add to each variable with name starting with a digit. Use '' or None to leave this out.

Returns

A CF-compliant xr.Dataset

Return type
xr.Dataset

satpy.aux_download module

Functions and utilities for downloading ancillary data.

class satpy.aux_download.**DataDownloadMixin**

Bases: object

Mixin class for Satpy components to download files.

This class simplifies the logic needed to download and cache data files needed for operations in a Satpy component (readers, writers, etc). It does this in a two step process where files that might be downloaded are "registered" and then "retrieved" when they need to be used.

To use this class include it as one of the subclasses of your Satpy component. Then in the `__init__` method, call the `register_data_files` function during initialization.

Note: This class is already included in the `FileYAMLReader` and `Writer` base classes. There is no need to define a custom class.

The below code is shown as an example:

```
from satpy.readers.yaml_reader import AbstractYAMLReader
from satpy.aux_download import DataDownloadMixin

class MyReader(AbstractYAMLReader, DataDownloadMixin):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.register_data_files()
```

This class expects data files to be configured in either a `self.info['data_files']` (standard for readers/writers) or `self.config['data_files']` list. The `data_files` item itself is a list of dictionaries. This information can also be passed directly to `register_data_files` for more complex cases. In YAML, for a reader, this might look like this:

```
reader:
    name: abi_l1b
    short_name: ABI L1b
    long_name: GOES-R ABI Level 1b
    ... other metadata ...
    data_files:
      - url: "https://example.com/my_data_file.dat"
      - url: "https://raw.githubusercontent.com/pytroll/satpy/main/README.rst"
        known_hash:
↪"sha256:5891286b63e7745de08c4b0ac204ad44cfdb9ab770309debaba90308305fa759"
      - url: "https://raw.githubusercontent.com/pytroll/satpy/main/RELEASING.md"
        filename: "satpy_releasing.md"
```

In this example we register two files that might be downloaded. If `known_hash` is not provided or None (null in YAML) then the data file will not be checked for validity when downloaded. See *register_file()* for more information. You can optionally specify `filename` to define the in-cache name when this file is downloaded. This can be useful in cases when the filename can not be easily determined from the URL.

When it comes time to needing the file, you can retrieve the local path by calling ~satpy.aux_download.retrieve(cache_key) with the "cache key" generated during registration. These keys will be in the format: <component_type>/<filename>. For a reader this would be readers/satpy_release.md.

This Mixin is not the only way to register and download files for a Satpy component, but is the most generic and flexible. Feel free to use the *register_file()* and *retrieve()* functions directly. However, *find_registerable_files()* must also be updated to support your component (if files are not register during initialization).

DATA_FILE_COMPONENTS = {'composit': 'composites', 'corr': 'modifiers', 'modifi': 'modifiers', 'reader': 'readers', 'writer': 'writers'}

property _data_file_component_type

static _register_data_file(*data_file_entry*, *comp_type*)

register_data_files(*data_files=None*)

Register a series of files that may be downloaded later.

See *DataDownloadMixin* for more information on the assumptions and structure of the data file configuration dictionary.

satpy.aux_download.**_find_registerable_files_compositors**(*sensors=None*)

Load all compositor configs so that files are registered.

Compositor objects should register files when they are initialized.

satpy.aux_download.**_find_registerable_files_readers**(*readers=None*)

Load all readers so that files are registered.

satpy.aux_download.**_find_registerable_files_writers**(*writers=None*)

Load all writers so that files are registered.

satpy.aux_download.**_generate_filename**(*filename*, *component_type*)

satpy.aux_download.**_register_modifier_files**(*modifiers*)

satpy.aux_download.**_retrieve_all_with_pooch**(*pooch_kwargs*)

satpy.aux_download.**_retrieve_offline**(*data_dir*, *cache_key*)

satpy.aux_download.**_should_download**(*cache_key*)

Check if we're running tests and can download this file.

satpy.aux_download.**find_registerable_files**(*readers=None*, *writers=None*, *composite_sensors=None*)

Load all Satpy components so they can be downloaded.

Parameters

- **readers** (*list or None*) – Limit searching to these readers. If not specified or None then all readers are searched. If an empty list then no readers are searched.
- **writers** (*list or None*) – Limit searching to these writers. If not specified or None then all writers are searched. If an empty list then no writers are searched.
- **composite_sensors** (*list or None*) – Limit searching to composite configuration files for these sensors. If None then all sensor configs will be searched. If an empty list then no composites will be searched.

satpy.aux_download.**register_file**(*url*, *filename*, *component_type=None*, *known_hash=None*)

Register file for future retrieval.

This function only prepares Satpy to be able to download and cache the provided file. It will not download the file. See *satpy.aux_download.retrieve()* for more information.

Parameters

- **url** (*str*) – URL where remote file can be downloaded.
- **filename** (*str*) – Filename used to identify and store the downloaded file as.
- **component_type** (*str or None*) – Name of the type of Satpy component that will use this file. Typically "readers", "composites", "writers", or "enhancements" for consistency. This will be prepended to the filename when storing the data in the cache.
- **known_hash** (*str*) – Hash used to verify the file is downloaded correctly. See https://www.fatiando.org/pooch/v1.3.0/beginner.html#hashes for more information. If not provided then the file is not checked.

Returns

Cache key that can be used to retrieve the file later. The cache key consists of the `component_type` and provided `filename`. This should be passed to `satpy.aux_download_retrieve()` when the file will be used.

satpy.aux_download.**retrieve**(*cache_key*, *pooch_kwargs=None*)

Download and cache the file associated with the provided `cache_key`.

Cache location is controlled by the config `data_dir` key. See *Data Directory* for more information.

Parameters

- **cache_key** (*str*) – Cache key returned by *register_file()*.
- **pooch_kwargs** (*dict or None*) – Extra keyword arguments to pass to `pooch.Pooch.fetch()`.

Returns

Local path of the cached file.

satpy.aux_download.**retrieve_all**(*readers=None*, *writers=None*, *composite_sensors=None*, *pooch_kwargs=None*)

Find cache-able data files for Satpy and download them.

The typical use case for this function is to download all ancillary files before going to an environment/system that does not have internet access.

Parameters

- **readers** (*list or None*) – Limit searching to these readers. If not specified or `None` then all readers are searched. If an empty list then no readers are searched.
- **writers** (*list or None*) – Limit searching to these writers. If not specified or `None` then all writers are searched. If an empty list then no writers are searched.
- **composite_sensors** (*list or None*) – Limit searching to composite configuration files for these sensors. If `None` then all sensor configs will be searched. If an empty list then no composites will be searched.
- **pooch_kwargs** (*dict*) – Additional keyword arguments to pass to pooch `fetch`.

satpy.aux_download.**retrieve_all_cmd**(*argv=None*)

Call 'retrieve_all' function from console script 'satpy_retrieve_all'.

satpy.conftest module

Pytest configuration and setup functions.

satpy.conftest.**pytest_configure**(*config*)

Set test configuration.

satpy.conftest.**pytest_unconfigure**(*config*)

Undo previous configurations.

satpy.dependency_tree module

Implementation of a dependency tree.

class satpy.dependency_tree.**DependencyTree**(*readers*, *compositors=None*, *modifiers=None*, *available_only=False*)

Bases: *Tree*

Structure to discover and store *Dataset* dependencies.

Used primarily by the *Scene* object to organize dependency finding. Dependencies are stored used a series of *Node* objects which this class is a subclass of.

Collect Dataset generating information.

Collect the objects that generate and have information about Datasets including objects that may depend on certain Datasets being generated. This includes readers, compositors, and modifiers.

Composites and modifiers are defined per-sensor. If multiple sensors are available, compositors and modifiers are searched for in sensor alphabetical order.

Parameters

- **readers** (*dict*) – Reader name -> Reader Object
- **compositors** (*dict*) – Sensor name -> Composite ID -> Composite Object. Empty dictionary by default.
- **modifiers** (*dict*) – Sensor name -> Modifier name -> (Modifier Class, modifier options). Empty dictionary by default.
- **available_only** (*bool*) – Whether only reader's available/loadable datasets should be used when searching for dependencies (True) or use all known/configured datasets regardless of whether the necessary files were provided to the reader (False). Note that when `False` loadable variations of a dataset will have priority over other known variations. Default is `False`.

_create_implicit_dependency_subtree(*dataset_key*, *query*)

_create_optional_subtrees(*parent*, *prereqs*, *query=None*)

Determine optional prerequisite Nodes for a composite.

Parameters

- **parent** (*Node*) – Compositor node to add these prerequisites under
- **prereqs** (*sequence*) – Strings (names), floats (wavelengths), or DataQuerys to analyze.

_create_prerequisite_subtrees(*parent*, *prereqs*, *query=None*)

Determine prerequisite Nodes for a composite.

Parameters

- **parent** (Node) – Compositor node to add these prerequisites under
- **prereqs** (*sequence*) – Strings (names), floats (wavelengths), DataQuerys or Nodes to analyze.

_create_required_subtrees(*parent*, *prereqs*, *query=None*)

Determine required prerequisite Nodes for a composite.

Parameters

- **parent** (Node) – Compositor node to add these prerequisites under
- **prereqs** (*sequence*) – Strings (names), floats (wavelengths), DataQuerys or Nodes to analyze.

_create_subtree_for_key(*dataset_key*, *query=None*)

Find the dependencies for *dataset_key*.

Parameters

- **dataset_key** (*str*, *float*, DataID, DataQuery) – Dataset identifier to locate and find any additional dependencies for.
- **query** (DataQuery) – Additional filter parameters. See *satpy.readers.get_key* for more details.

_create_subtree_from_compositors(*dataset_key*, *query*)

_create_subtree_from_reader(*dataset_key*, *query*)

_find_compositor(*dataset_key*, *query*)

Find the compositor object for the given dataset_key.

_find_matching_ids_in_readers(*dataset_key*)

_find_reader_node(*dataset_key*, *query*)

Attempt to find a *DataID* in the available readers.

Parameters

dataset_key (*str*, *float*, DataID, DataQuery) – Dataset name, wavelength, *DataID* or *DataQuery* to use in searching for the dataset from the available readers.

_get_subtree_for_existing_key(*dsq*)

_get_subtree_for_existing_name(*dsq*)

static _get_unique_id_from_sorted_ids(*sorted_ids*, *distances*)

_get_unique_matching_id(*matching_ids*, *dataset_key*, *query*)

Get unique matching id from *matching_ids*, for a given *dataset_key* and some optional *query*.

_get_unique_reader_node_from_id(*data_id*, *reader_name*)

_promote_query_to_modified_dataid(*query*, *dep_key*)

Promote a query to an id based on the dataset it will modify (dep).

Typical use case is requesting a modified dataset (query). This modified dataset most likely depends on a less-modified dataset (dep_key). The less-modified dataset must come from a reader (at least for now) or will eventually depend on a reader dataset. The original request key may be limited like (wavelength=0.67, modifiers=('a', 'b')) while the reader-based key should have all of its properties specified. This method updates the original request key so it is fully specified and should reduce the chance of Node's not being unique.

copy()

Copy this node tree.

Note all references to readers are removed. This is meant to avoid tree copies accessing readers that would return incompatible (Area) data. Theoretically it should be possible for tree copies to request compositor or modifier information as long as they don't depend on any datasets not already existing in the dependency tree.

get_compositor(*key*)

Get a compositor.

get_modifier(*comp_id*)

Get a modifer.

populate_with_keys(*dataset_keys: set*, *query=None*)

Populate the dependency tree.

Parameters

- **dataset_keys** (*set*) – Strings, DataIDs, DataQuerys to find dependencies for
- **query** (DataQuery) – Additional filter parameters. See *satpy.readers.get_key* for more details.

Returns

Root node of the dependency tree and a set of unknown datasets

Return type

(*Node*, set)

update_compositors_and_modifiers(*compositors: dict*, *modifiers: dict*) → None

Add additional compositors and modifiers to the tree.

Provided dictionaries and the first sub-level dictionaries are copied to avoid modifying the input.

Parameters

- **compositors** (*dict*) – Sensor name -> composite ID -> Composite Object
- **modifiers** (*dict*) – Sensor name -> Modifier name -> (Modifier Class, modifier options)

update_node_name(*node*, *new_name*)

Update 'name' property of a node and any related metadata.

class satpy.dependency_tree.**Tree**

Bases: object

A tree implementation.

Set up the tree.

add_child(*parent*, *child*)

Add a child to the tree.

add_leaf(*ds_id*, *parent=None*)

Add a leaf to the tree.

contains(*item*)

Check contains when we know the *exact* DataID or DataQuery.

empty_node = <Node ('__EMPTY_LEAF_SENTINEL__')>

getitem(*item*)

Get Node when we know the *exact* DataID or DataQuery.

leaves(*limit_nodes_to: Iterable[DataID] | None = None*, *unique: bool = True*) → list[*Node*]

Get the leaves of the tree starting at the root.

Parameters

- **limit_nodes_to** – Limit leaves to Nodes with the names (DataIDs) specified.
- **unique** – Only include individual leaf nodes once.

Returns

list of leaf nodes

trunk(*limit_nodes_to: Iterable[DataID] | None = None*, *unique: bool = True*, *limit_children_to: Container[DataID] | None = None*) → list[*Node*]

Get the trunk nodes of the tree starting at this root.

Parameters

- **limit_nodes_to** – Limit searching to trunk nodes with the names (DataIDs) specified and the children of these nodes.
- **unique** – Only include individual trunk nodes once
- **limit_children_to** – Limit searching to the children with the specified names. These child nodes will be included in the result, but not their children.

Returns

list of trunk nodes

class satpy.dependency_tree.**_DataIDContainer**

Bases: dict

Special dictionary object that can handle dict operations based on dataset name, wavelength, or DataID.

Note: Internal dictionary keys are *DataID* objects.

get_key(*match_key*)

Get multiple fully-specified keys that match the provided query.

Parameters

match_key (DataID) – DataID or DataQuery of query parameters to use for searching. Can also be a string representing the dataset name or a number representing the dataset wavelength.

keys()

Give currently contained keys.

satpy.node module

Nodes to build trees.

class satpy.node.**CompositorNode**(*compositor*)

Bases: *Node*

Implementation of a compositor-specific node.

Set up the node.

_copy_name_and_data(*node_cache=None*)

add_optional_nodes(*children*)

Add nodes to the optional field.

add_required_nodes(*children*)

Add nodes to the required field.

property compositor

Get the compositor.

property optional_nodes

Get the optional nodes.

property required_nodes

Get the required nodes.

exception satpy.node.**MissingDependencies**(*missing_dependencies*, **args*, ***kwargs*)

Bases: RuntimeError

Exception when dependencies are missing.

Set up the exception.

class satpy.node.**Node**(*name*, *data=None*)

Bases: object

A node object.

Init the node object.

_copy_name_and_data(*node_cache=None*)

add_child(*obj*)

Add a child to the node.

copy(*node_cache=None*)

Make a copy of the node.

display(*previous=0*, *include_data=False*)

Display the node.

flatten(*d=None*)

Flatten tree structure to a one level dictionary.

Parameters

d (*dict, optional*) – output dictionary to update

Returns

Node.name -> Node. The returned dictionary includes the current Node and all its children.

Return type
dict

property is_leaf

Check if the node is a leaf.

leaves(*unique=True*)

Get the leaves of the tree starting at this root.

trunk(*unique=True*, *limit_children_to=None*)

Get the trunk of the tree starting at this root.

update_name(*new_name*)

Update 'name' property.

class satpy.node.**ReaderNode**(*unique_id*, *reader_name*)

Bases: *Node*

Implementation of a storage-based node.

Set up the node.

_copy_name_and_data(*node_cache*)

property reader_name

Get the name of the reader.

satpy.plugin_base module

Classes and utilities for defining generic "plugin" components.

class satpy.plugin_base.**Plugin**(*default_config_filename=None*, *config_files=None*, ***kwargs*)

Bases: `object`

Base plugin class for all dynamically loaded and configured objects.

Load configuration files related to this plugin.

This initializes a *self.config* dictionary that can be used to customize the subclass.

Parameters

- **default_config_filename** (*str*) – Configuration filename to use if no other files have been specified with *config_files*.
- **config_files** (*list or str*) – Configuration files to load instead of those automatically found in *SATPY_CONFIG_PATH* and other default configuration locations.
- **kwargs** (*dict*) – Unused keyword arguments.

load_yaml_config(*conf*)

Load a YAML configuration file and recursively update the overall configuration.

satpy.resample module

Resampling in Satpy.

Satpy provides multiple resampling algorithms for resampling geolocated data to uniform projected grids. The easiest way to perform resampling in Satpy is through the *Scene* object's *resample()* method. Additional utility functions are also available to assist in resampling data. Below is more information on resampling with Satpy as well as links to the relevant API documentation for available keyword arguments.

Resampling algorithms

Table 6: Available Resampling Algorithms

Resampler	Description	Related
nearest	Nearest Neighbor	*KDTreeResampler*
ewa	Elliptical Weighted Averaging	DaskEWAResampler
ewa_legacy	Elliptical Weighted Averaging (Legacy)	LegacyDaskEWAResampler
native	Native	*NativeResampler*
bilinear	Bilinear	*BilinearResampler*
bucket_avg	Average Bucket Resampling	*BucketAvg*
bucket_sum	Sum Bucket Resampling	*BucketSum*
bucket_count	Count Bucket Resampling	*BucketCount*
bucket_fraction	Fraction Bucket Resampling	*BucketFraction*
gradient_search	Gradient Search Resampling	GradientSearchResampler

The resampling algorithm used can be specified with the `resampler` keyword argument and defaults to `nearest`:

```
>>> scn = Scene(...)
>>> euro_scn = scn.resample('euro4', resampler='nearest')
```

Warning: Some resampling algorithms expect certain forms of data. For example, the EWA resampling expects polar-orbiting swath data and prefers if the data can be broken in to "scan lines". See the API documentation for a specific algorithm for more information.

Resampling for comparison and composites

While all the resamplers can be used to put datasets of different resolutions on to a common area, the 'native' resampler is designed to match datasets to one resolution in the dataset's original projection. This is extremely useful when generating composites between bands of different resolutions.

```
>>> new_scn = scn.resample(resampler='native')
```

By default this resamples to the *highest resolution area* (smallest footprint per pixel) shared between the loaded datasets. You can easily specify the lowest resolution area:

```
>>> new_scn = scn.resample(scn.coarsest_area(), resampler='native')
```

Providing an area that is neither the minimum or maximum resolution area may work, but behavior is currently undefined.

Caching for geostationary data

Satpy will do its best to reuse calculations performed to resample datasets, but it can only do this for the current processing and will lose this information when the process/script ends. Some resampling algorithms, like `nearest` and `bilinear`, can benefit by caching intermediate data on disk in the directory specified by *cache_dir* and using it next time. This is most beneficial with geostationary satellite data where the locations of the source data and the target pixels don't change over time.

```
>>> new_scn = scn.resample('euro4', cache_dir='/path/to/cache_dir')
```

See the documentation for specific algorithms to see availability and limitations of caching for that algorithm.

Create custom area definition

See `pyresample.geometry.AreaDefinition` for information on creating areas that can be passed to the resample method:

```
>>> from pyresample.geometry import AreaDefinition
>>> my_area = AreaDefinition(...)
>>> local_scene = scn.resample(my_area)
```

Create dynamic area definition

See `pyresample.geometry.DynamicAreaDefinition` for more information.

Examples coming soon…

Store area definitions

Area definitions can be saved to a custom YAML file (see pyresample's writing to disk) and loaded using pyresample's utility methods (pyresample's loading from disk):

```
>>> from pyresample import load_area
>>> my_area = load_area('my_areas.yaml', 'my_area')
```

Or using *satpy.resample.get_area_def()*, which will search through all `areas.yaml` files in your `SATPY_CONFIG_PATH`:

```
>>> from satpy.resample import get_area_def
>>> area_eurol = get_area_def("eurol")
```

For examples of area definitions, see the file `etc/areas.yaml` that is included with Satpy and where all the area definitions shipped with Satpy are defined.

class satpy.resample.**BilinearResampler**(*source_geo_def*, *target_geo_def*)

Bases: `BaseResampler`

Resample using bilinear interpolation.

This resampler implements on-disk caching when the *cache_dir* argument is provided to the *resample* method. This should provide significant performance improvements on consecutive resampling of geostationary data.

Parameters

- **cache_dir** (*str*) – Long term storage directory for intermediate results.
- **radius_of_influence** (*float*) – Search radius cut off distance in meters
- **epsilon** (*float*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time.
- **reduce_data** (*bool*) – Reduce the input data to (roughly) match the target area.

Init BilinearResampler.

compute(*data*, *fill_value=None*, ***kwargs*)

Resample the given data using bilinear interpolation.

load_bil_info(*cache_dir*, ***kwargs*)

Load bilinear resampling info from cache directory.

precompute(*mask=None*, *radius_of_influence=50000*, *epsilon=0*, *reduce_data=True*, *cache_dir=False*, ***kwargs*)

Create bilinear coefficients and store them for later use.

save_bil_info(*cache_dir*, ***kwargs*)

Save bilinear resampling info to cache directory.

class satpy.resample.**BucketAvg**(*source_geo_def*, *target_geo_def*)

Bases: *BucketResamplerBase*

Class for averaging bucket resampling.

Bucket resampling calculates the average of all the values that are closest to each bin and inside the target area.

Parameters

- **fill_value** (*float (default: np.nan)*) – Fill value to mark missing/invalid values in the input data, as well as in the binned and averaged output data.
- **skipna** (*boolean (default: True)*) – If True, skips missing values (as marked by NaN or *fill_value*) for the average calculation (similarly to Numpy's *nanmean*). Buckets containing only missing values are set to fill_value. If False, sets the bucket to fill_value if one or more missing values are present in the bucket (similarly to Numpy's *mean*). In both cases, empty buckets are set to *fill_value*.

Initialize bucket resampler.

compute(*data*, *fill_value=nan*, *skipna=True*, ***kwargs*)

Call the resampling.

Parameters

- **data** (*numpy.Array, dask.Array*) – Data to be resampled
- **fill_value** (*numpy.nan, int*) – fill_value. Defaults to numpy.nan
- **skipna** (*boolean*) – Skip NA's. Default *True*

Returns

dask.Array

class satpy.resample.**BucketCount**(*source_geo_def*, *target_geo_def*)

Bases: *BucketResamplerBase*

Class for bucket resampling which implements hit-counting.

This resampler calculates the number of occurences of the input data closest to each bin and inside the target area.

Initialize bucket resampler.

compute(*data*, ***kwargs*)

Call the resampling.

class satpy.resample.**BucketFraction**(*source_geo_def*, *target_geo_def*)

Bases: BucketResamplerBase

Class for bucket resampling to compute category fractions.

This resampler calculates the fraction of occurences of the input data per category.

Initialize bucket resampler.

compute(*data*, *fill_value=nan*, *categories=None*, ***kwargs*)

Call the resampling.

class satpy.resample.**BucketResamplerBase**(*source_geo_def*, *target_geo_def*)

Bases: BaseResampler

Base class for bucket resampling which implements averaging.

Initialize bucket resampler.

compute(*data*, ***kwargs*)

Call the resampling.

precompute(***kwargs*)

Create X and Y indices and store them for later use.

resample(*data*, ***kwargs*)

Resample *data* by calling *precompute* and *compute* methods.

Parameters

data (*xarray.DataArray*) – Data to be resampled

Returns (xarray.DataArray): Data resampled to the target area

class satpy.resample.**BucketSum**(*source_geo_def*, *target_geo_def*)

Bases: BucketResamplerBase

Class for bucket resampling which implements accumulation (sum).

This resampler calculates the cumulative sum of all the values that are closest to each bin and inside the target area.

Parameters

- **fill_value** (*float (default: np.nan)*) – Fill value for missing data
- **skipna** (*boolean (default: True)*) – If True, skips NaN values for the sum calculation (similarly to Numpy's *nansum*). Buckets containing only NaN are set to zero. If False, sets the bucket to NaN if one or more NaN values are present in the bucket (similarly to Numpy's *sum*). In both cases, empty buckets are set to 0.

Initialize bucket resampler.

compute(*data*, *skipna=True*, ***kwargs*)

Call the resampling.

class satpy.resample.**KDTreeResampler**(*source_geo_def*, *target_geo_def*)

Bases: BaseResampler

Resample using a KDTree-based nearest neighbor algorithm.

This resampler implements on-disk caching when the *cache_dir* argument is provided to the *resample* method. This should provide significant performance improvements on consecutive resampling of geostationary data. It is not recommended to provide *cache_dir* when the *mask* keyword argument is provided to *precompute* which occurs by default for *SwathDefinition* source areas.

Parameters

- **cache_dir** (*str*) – Long term storage directory for intermediate results.
- **mask** (*bool*) – Force resampled data's invalid pixel mask to be used when searching for nearest neighbor pixels. By default this is True for SwathDefinition source areas and False for all other area definition types.
- **radius_of_influence** (*float*) – Search radius cut off distance in meters
- **epsilon** (*float*) – Allowed uncertainty in meters. Increasing uncertainty reduces execution time.

Init KDTreeResampler.

_adjust_radius_of_influence(*radius_of_influence*)

Adjust radius of influence.

_apply_cached_index(*val*, *idx_name*, *persist=False*)

Reassign resampler index attributes.

_read_resampler_attrs()

Read certain attributes from the resampler for caching.

compute(*data*, *weight_funcs=None*, *fill_value=nan*, *with_uncert=False*, ***kwargs*)

Resample data.

load_neighbour_info(*cache_dir*, *mask=None*, ***kwargs*)

Read index arrays from either the in-memory or disk cache.

precompute(*mask=None*, *radius_of_influence=None*, *epsilon=0*, *cache_dir=None*, ***kwargs*)

Create a KDTree structure and store it for later use.

Note: The *mask* keyword should be provided if geolocation may be valid where data points are invalid.

save_neighbour_info(*cache_dir*, *mask=None*, ***kwargs*)

Cache resampler's index arrays if there is a cache dir.

class satpy.resample.**NativeResampler**(*source_geo_def: SwathDefinition | AreaDefinition*, *target_geo_def: CoordinateDefinition | AreaDefinition*)

Bases: BaseResampler

Expand or reduce input datasets to be the same shape.

If data is higher resolution (more pixels) than the destination area then data is averaged to match the destination resolution.

If data is lower resolution (less pixels) than the destination area then data is repeated to match the destination resolution.

This resampler does not perform any caching or masking due to the simplicity of the operations.

Initialize resampler with geolocation information.

Parameters

- **source_geo_def** – Geolocation definition for the data to be resampled
- **target_geo_def** – Geolocation definition for the area to resample data to.

classmethod _expand_reduce(*d_arr*, *repeats*)

Expand reduce.

compute(*data*, *expand=True*, ***kwargs*)

Resample data with NativeResampler.

resample(*data*, *cache_dir=None*, *mask_area=False*, ***kwargs*)

Run NativeResampler.

satpy.resample.**_aggregate**(*d*, *y_size*, *x_size*)

Average every 4 elements (2x2) in a 2D array.

satpy.resample.**_get_arg_to_pass_for_skipna_handling**(***kwargs*)

Determine if skipna can be passed to the compute functions for the average and sum bucket resampler.

satpy.resample.**_get_replicated_chunk_sizes**(*d_arr*, *repeats*)

satpy.resample.**_mean**(*data*, *y_size*, *x_size*)

satpy.resample.**_move_existing_caches**(*cache_dir*, *filename*)

Move existing cache files out of the way.

satpy.resample.**_rechunk_if_nonfactor_chunks**(*dask_arr*, *y_size*, *x_size*)

satpy.resample.**_repeat_by_factor**(*data*, *block_info=None*)

satpy.resample.**_replicate**(*d_arr*, *repeats*)

Repeat data pixels by the per-axis factors specified.

satpy.resample.**add_crs_xy_coords**(*data_arr*, *area*)

Add `pyproj.crs.CRS` and x/y or lons/lats to coordinates.

For SwathDefinition or GridDefinition areas this will add a *crs* coordinate and coordinates for the 2D arrays of *lons* and *lats*.

For AreaDefinition areas this will add a *crs* coordinate and the 1-dimensional *x* and *y* coordinate variables.

Parameters

- **data_arr** (*xarray.DataArray*) – DataArray to add the 'crs' coordinate.
- **area** (*pyresample.geometry.AreaDefinition*) – Area to get CRS information from.

satpy.resample.**add_xy_coords**(*data_arr*, *area*, *crs=None*)

Assign x/y coordinates to DataArray from provided area.

If 'x' and 'y' coordinates already exist then they will not be added.

Parameters

- **data_arr** (*xarray.DataArray*) – data object to add x/y coordinates to
- **area** (*pyresample.geometry.AreaDefinition*) – area providing the coordinate data.
- **crs** (*pyproj.crs.CRS or None*) – CRS providing additional information about the area's coordinate reference system if available. Requires pyproj 2.0+.

Returns (xarray.DataArray): Updated DataArray object

satpy.resample.**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 $SATPY_CONFIG_PATH directory, and its name is defined in satpy's configuration file.

satpy.resample.**get_area_file**()

Find area file(s) to use.

The files are to be named *areas.yaml* or *areas.def*.

satpy.resample.**get_fill_value**(*dataset*)

Get the fill value of the *dataset*, defaulting to np.nan.

satpy.resample.**hash_dict**(*the_dict*, *the_hash=None*)

Calculate a hash for a dictionary.

satpy.resample.**prepare_resampler**(*source_area*, *destination_area*, *resampler=None*, ***resample_kwargs*)

Instantiate and return a resampler.

satpy.resample.**resample**(*source_area*, *data*, *destination_area*, *resampler=None*, ***kwargs*)

Do the resampling.

satpy.resample.**resample_dataset**(*dataset*, *destination_area*, ***kwargs*)

Resample *dataset* and return the resampled version.

Parameters

- **dataset** (*xarray.DataArray*) – Data to be resampled.
- **destination_area** – The destination onto which to project the data, either a full blown area definition or a string corresponding to the name of the area as defined in the area file.
- ****kwargs** – The extra parameters to pass to the resampler objects.

Returns

A resampled DataArray with updated `.attrs["area"]` field. The dtype of the array is preserved.

satpy.resample.**update_resampled_coords**(*old_data*, *new_data*, *new_area*)

Add coordinate information to newly resampled DataArray.

Parameters

- **old_data** (*xarray.DataArray*) – Old data before resampling.
- **new_data** (*xarray.DataArray*) – New data after resampling.
- **new_area** (*pyresample.geometry.BaseDefinition*) – Area definition for the newly resampled data.

satpy.scene module

Scene object to hold satellite data.

exception satpy.scene.**DelayedGeneration**

Bases: KeyError

Mark that a dataset can't be generated without further modification.

class satpy.scene.**Scene**(*filenames=None*, *reader=None*, *filter_parameters=None*, *reader_kwargs=None*)

Bases: object

The Almighty Scene Class.

Example usage:

```
from satpy import Scene
from glob import glob

# create readers and open files
scn = Scene(filenames=glob('/path/to/files/*'), reader='viirs_sdr')

# load datasets from input files
scn.load(['I01', 'I02'])

# resample from satellite native geolocation to builtin 'eurol' Area
new_scn = scn.resample('eurol')

# save all resampled datasets to geotiff files in the current directory
new_scn.save_datasets()
```

Initialize Scene with Reader and Compositor objects.

To load data *filenames* and preferably *reader* must be specified:

```
scn = Scene(filenames=glob('/path/to/viirs/sdr/files/*'), reader='viirs_sdr')
```

If `filenames` is provided without `reader` then the available readers will be searched for a Reader that can support the provided files. This can take a considerable amount of time so it is recommended that `reader` always be provided. Note without `filenames` the Scene is created with no Readers available requiring Datasets to be added manually:

```
scn = Scene()
scn['my_dataset'] = Dataset(my_data_array, **my_info)
```

Further, notice that it is also possible to load a combination of files or sets of files each requiring their specific reader. For that `filenames` needs to be a *dict* (see parameters list below), e.g.:

```
scn = Scene(filenames={'nwcsaf-pps_nc': glob('/path/to/nwc/saf/pps/files/*'),
                       'modis_l1b': glob('/path/to/modis/lvl1/files/*')})
```

Parameters

- **filenames** (*iterable or dict*) – A sequence of files that will be used to load data from. A `dict` object should map reader names to a list of filenames for that reader.
- **reader** (*str or list*) – The name of the reader to use for loading the data or a list of names.

- **filter_parameters** (*dict*) – Specify loaded file filtering parameters. Shortcut for *reader_kwargs['filter_parameters']*.
- **reader_kwargs** (*dict*) – Keyword arguments to pass to specific reader instances. Either a single dictionary that will be passed onto to all reader instances, or a dictionary mapping reader names to sub-dictionaries to pass different arguments to different reader instances.

  Keyword arguments for remote file access are also given in this dictionary. See documentation for usage examples.

_check_known_composites(*available_only=False*)

Create new dependency tree and check what composites we know about.

static _compare_area_defs(*compare_func: Callable*, *area_defs: list[AreaDefinition]*) → list[AreaDefinition]

_compare_areas(*datasets=None*, *compare_func=<built-in function max>*)

Compare areas for the provided datasets.

Parameters

- **datasets** (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets. This can also be a series of area objects, typically AreaDefinitions.
- **compare_func** (*callable*) – *min* or *max* or other function used to compare the dataset's areas.

static _compare_swath_defs(*compare_func: Callable*, *swath_defs: list[SwathDefinition]*) → list[SwathDefinition]

_contained_sensor_names() → set[str]

_copy_datasets_and_wishlist(*new_scn*, *datasets*)

_create_reader_instances(*filenames=None*, *reader=None*, *reader_kwargs=None*)

Find readers and return their instances.

_filter_loaded_datasets_from_trunk_nodes(*trunk_nodes*)

_gather_all_areas(*datasets*)

Gather all areas from datasets.

They have to be of the same type, and at least one dataset should have an area.

_generate_composite(*comp_node: CompositorNode*, *keepables: set*)

Collect all composite prereqs and create the specified composite.

Parameters

- **comp_node** – Composite Node to generate a Dataset for
- **keepables** – *set* to update if any datasets are needed when generation is continued later. This can happen if generation is delayed to incompatible areas which would require resampling first.

_generate_composites_from_loaded_datasets()

Compute all the composites contained in *requirements*.

_generate_composites_nodes_from_loaded_datasets(*compositor_nodes*)

Read (generate) composites.

_get_finalized_destination_area(*destination_area*, *new_scn*)

_get_prereq_datasets(*comp_id*, *prereq_nodes*, *keepables*, *skip=False*)

Get a composite's prerequisites, generating them if needed.

Parameters

- **comp_id** (DataID) – DataID for the composite whose prerequisites are being collected.
- **prereq_nodes** (*sequence of Nodes*) – Prerequisites to collect
- **keepables** (*set*) – *set* to update if any prerequisites can't be loaded at this time (see *_generate_composite*).
- **skip** (*bool*) – If True, consider prerequisites as optional and only log when they are missing. If False, prerequisites are considered required and will raise an exception and log a warning if they can't be collected. Defaults to False.

Raises

KeyError – If required (skip=False) prerequisite can't be collected.

static _get_writer_by_ext(*extension*)

Find the writer matching the `extension`.

Defaults to "simple_image".

Example Mapping:

- geotiff: .tif, .tiff
- cf: .nc
- mitiff: .mitiff
- simple_image: .png, .jpeg, .jpg, …

Parameters

extension (*str*) – Filename extension starting with "." (ex. ".png").

Returns

The name of the writer to use for this extension.

Return type

str

_ipython_key_completions_()

_load_datasets_by_readers(*reader_datasets*, ***kwargs*)

_prepare_resampler(*source_area*, *destination_area*, *resamplers*, *resample_kwargs*)

_read_dataset_nodes_from_storage(*reader_nodes*, ***kwargs*)

Read the given dataset nodes from storage.

_read_datasets_from_storage(***kwargs*)

Load datasets from the necessary reader.

Parameters

****kwargs** – Keyword arguments to pass to the reader's *load* method.

Returns

DatasetDict of loaded datasets

_reader_times(*time_prop_name*)

_reduce_data(*dataset*, *source_area*, *destination_area*, *reduce_data*, *reductions*, *resample_kwargs*)

_remove_failed_datasets(*keepables*)

Remove the datasets that we couldn't create.

_resampled_scene(*new_scn*, *destination_area*, *reduce_data=True*, ***resample_kwargs*)

Resample *datasets* to the *destination* area.

If data reduction is enabled, some local caching is perfomed in order to avoid recomputation of area intersections.

static _slice_area_from_bbox(*src_area*, *dst_area*, *ll_bbox=None*, *xy_bbox=None*)

Slice the provided area using the bounds provided.

_slice_data(*source_area*, *slices*, *dataset*)

Slice the data to reduce it.

_slice_datasets(*dataset_ids*, *slice_key*, *new_area*, *area_only=True*)

Slice scene in-place for the datasets specified.

_sort_dataset_nodes_by_reader(*reader_nodes*)

_update_dependency_tree(*needed_datasets*, *query*)

aggregate(*dataset_ids=None*, *boundary='trim'*, *side='left'*, *func='mean'*, ***dim_kwargs*)

Create an aggregated version of the Scene.

Parameters

- **dataset_ids** (*iterable*) – DataIDs to include in the returned *Scene*. Defaults to all datasets.
- **func** (*string, callable*) – Function to apply on each aggregation window. One of 'mean', 'sum', 'min', 'max', 'median', 'argmin', 'argmax', 'prod', 'std', 'var' strings or a custom function. 'mean' is the default.
- **boundary** – See `xarray.DataArray.coarsen()`, 'trim' by default.
- **side** – See `xarray.DataArray.coarsen()`, 'left' by default.
- **dim_kwargs** – the size of the windows to aggregate.

Returns

A new aggregated scene

See also:

xarray.DataArray.coarsen

Example

scn.aggregate(func='min', x=2, y=2) will apply the *min* function across a window of size 2 pixels.

all_composite_ids()

Get all IDs for configured composites.

all_composite_names()

Get all names for all configured composites.

all_dataset_ids(*reader_name=None*, *composites=False*)

Get IDs of all datasets from loaded readers or *reader_name* if specified.

Excludes composites unless `composites=True` is passed.

Parameters

- **reader_name** (*str, optional*) – Name of reader for which to return dataset IDs. If not passed, return dataset IDs for all readers.
- **composites** (*bool, optional*) – If True, return dataset IDs including composites. If False (default), return only non-composite dataset IDs.

Returns: list of all dataset IDs

all_dataset_names(*reader_name=None*, *composites=False*)

Get all known dataset names configured for the loaded readers.

Note that some readers dynamically determine what datasets are known by reading the contents of the files they are provided. This means that the list of datasets returned by this method may change depending on what files are provided even if a product/dataset is a "standard" product for a particular reader.

Excludes composites unless `composites=True` is passed.

Parameters

- **reader_name** (*str, optional*) – Name of reader for which to return dataset IDs. If not passed, return dataset names for all readers.
- **composites** (*bool, optional*) – If True, return dataset IDs including composites. If False (default), return only non-composite dataset names.

Returns: list of all dataset names

all_modifier_names()

Get names of configured modifier objects.

property all_same_area

All contained data arrays are on the same area.

property all_same_proj

All contained data array are in the same projection.

available_composite_ids()

Get IDs of composites that can be generated from the available datasets.

available_composite_names()

Names of all configured composites known to this Scene.

available_dataset_ids(*reader_name=None*, *composites=False*)

Get DataIDs of loadable datasets.

This can be for all readers loaded by this Scene or just for `reader_name` if specified.

Available dataset names are determined by what each individual reader can load. This is normally determined by what files are needed to load a dataset and what files have been provided to the scene/reader. Some readers dynamically determine what is available based on the contents of the files provided.

By default, only returns non-composite dataset IDs. To include composite dataset IDs, pass `composites=True`.

Parameters

- **reader_name** (*str, optional*) – Name of reader for which to return dataset IDs. If not passed, return dataset IDs for all readers.
- **composites** (*bool, optional*) – If True, return dataset IDs including composites. If False (default), return only non-composite dataset IDs.

Returns: list of available dataset IDs

available_dataset_names(*reader_name=None*, *composites=False*)

Get the list of the names of the available datasets.

By default, this only shows names of datasets directly defined in (one of the) readers. Names of composites are not returned unless the argument `composites=True` is passed.

Parameters

- **reader_name** (*str, optional*) – Name of reader for which to return dataset IDs. If not passed, return dataset names for all readers.
- **composites** (*bool, optional*) – If True, return dataset IDs including composites. If False (default), return only non-composite dataset names.

Returns: list of available dataset names

chunk(***kwargs*)

Call *chunk* on all Scene data arrays.

See `xarray.DataArray.chunk()` for more details.

coarsest_area(*datasets=None*)

Get lowest resolution area for the provided datasets.

Parameters

datasets (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets.

compute(***kwargs*)

Call *compute* on all Scene data arrays.

See `xarray.DataArray.compute()` for more details. Note that this will convert the contents of the DataArray to numpy arrays which may not work with all parts of Satpy which may expect dask arrays.

copy(*datasets=None*)

Create a copy of the Scene including dependency information.

Parameters

datasets (*list, tuple*) – *DataID* objects for the datasets to include in the new Scene object.

crop(*area=None*, *ll_bbox=None*, *xy_bbox=None*, *dataset_ids=None*)

Crop Scene to a specific Area boundary or bounding box.

Parameters

- **area** (*AreaDefinition*) – Area to crop the current Scene to
- **ll_bbox** (*tuple, list*) – 4-element tuple where values are in lon/lat degrees. Elements are `(xmin, ymin, xmax, ymax)` where X is longitude and Y is latitude.
- **xy_bbox** (*tuple, list*) – Same as *ll_bbox* but elements are in projection units.
- **dataset_ids** (*iterable*) – DataIDs to include in the returned *Scene*. Defaults to all datasets.

This method will attempt to intelligently slice the data to preserve relationships between datasets. For example, if we are cropping two DataArrays of 500m and 1000m pixel resolution then this method will assume that exactly 4 pixels of the 500m array cover the same geographic area as a single 1000m pixel. It handles these cases based on the shapes of the input arrays and adjusting slicing indexes accordingly. This method will have trouble handling cases where data arrays seem related but don't cover the same geographic area or if the coarsest resolution data is not related to the other arrays which are related.

It can be useful to follow cropping with a call to the native resampler to resolve all datasets to the same resolution and compute any composites that could not be generated previously:

```
>>> cropped_scn = scn.crop(ll_bbox=(-105., 40., -95., 50.))
>>> remapped_scn = cropped_scn.resample(resampler='native')
```

Note: The *resample* method automatically crops input data before resampling to save time/memory.

property end_time

Return the end time of the file.

If no data is currently contained in the Scene then loaded readers will be consulted. If no readers are loaded then the `Scene.start_time` is returned.

finest_area(*datasets=None*)

Get highest resolution area for the provided datasets.

Parameters

datasets (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets.

generate_possible_composites(*unload*)

See which composites can be generated and generate them.

Parameters

unload (*bool*) – if the dependencies of the composites should be unloaded after successful generation.

get(*key*, *default=None*)

Return value from DatasetDict with optional default.

images()

Generate images for all the datasets from the scene.

iter_by_area()

Generate datasets grouped by Area.

Returns

generator of (area_obj, list of dataset objects)

keys(***kwargs*)

Get DataID keys for the underlying data container.

load(*wishlist*, *calibration='*'*, *resolution='*'*, *polarization='*'*, *level='*'*, *modifiers='*'*, *generate=True*, *unload=True*, ***kwargs*)

Read and generate requested datasets.

When the *wishlist* contains *DataQuery* objects they can either be fully-specified *DataQuery* objects with every parameter specified or they can not provide certain parameters and the "best" parameter will be

chosen. For example, if a dataset is available in multiple resolutions and no resolution is specified in the wishlist's DataQuery then the highest (the smallest number) resolution will be chosen.

Loaded *DataArray* objects are created and stored in the Scene object.

Parameters

- **wishlist** (*iterable*) – List of names (str), wavelengths (float), DataQuery objects or DataID of the requested datasets to load. See *available_dataset_ids()* for what datasets are available.
- **calibration** (*list | str*) – Calibration levels to limit available datasets. This is a shortcut to having to list each DataQuery/DataID in *wishlist*.
- **resolution** (*list | float*) – Resolution to limit available datasets. This is a shortcut similar to calibration.
- **polarization** (*list | str*) – Polarization ('V', 'H') to limit available datasets. This is a shortcut similar to calibration.
- **modifiers** (*tuple | str*) – Modifiers that should be applied to the loaded datasets. This is a shortcut similar to calibration, but only represents a single set of modifiers as a tuple. For example, specifying `modifiers=('sunz_corrected', 'rayleigh_corrected')` will attempt to apply both of these modifiers to all loaded datasets in the specified order ('sunz_corrected' first).
- **level** (*list | str*) – Pressure level to limit available datasets. Pressure should be in hPa or mb. If an altitude is used it should be specified in inverse meters (1/m). The units of this parameter ultimately depend on the reader.
- **generate** (*bool*) – Generate composites from the loaded datasets (default: True)
- **unload** (*bool*) – Unload datasets that were required to generate the requested datasets (composite dependencies) but are no longer needed.

max_area(*datasets=None*)

Get highest resolution area for the provided datasets. Deprecated.

Deprecated. Use *finest_area()* instead.

Parameters

datasets (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets.

min_area(*datasets=None*)

Get lowest resolution area for the provided datasets. Deprecated.

Deprecated. Use *coarsest_area()* instead.

Parameters

datasets (*iterable*) – Datasets whose areas will be compared. Can be either *xarray.DataArray* objects or identifiers to get the DataArrays from the current Scene. Defaults to all datasets.

property missing_datasets

Set of DataIDs that have not been successfully loaded.

persist(***kwargs*)

Call *persist* on all Scene data arrays.

See `xarray.DataArray.persist()` for more details.

resample(*destination=None*, *datasets=None*, *generate=True*, *unload=True*, *resampler=None*, *reduce_data=True*, ***resample_kwargs*)

Resample datasets and return a new scene.

Parameters

- **destination** (*AreaDefinition, GridDefinition*) – area definition to resample to. If not specified then the area returned by *Scene.finest_area()* will be used.
- **datasets** (*list*) – Limit datasets to resample to these specified data arrays. By default all currently loaded datasets are resampled.
- **generate** (*bool*) – Generate any requested composites that could not be previously due to incompatible areas (default: True).
- **unload** (*bool*) – Remove any datasets no longer needed after requested composites have been generated (default: True).
- **resampler** (*str*) – Name of resampling method to use. By default, this is a nearest neighbor KDTree-based resampling ('nearest'). Other possible values include 'native', 'ewa', etc. See the *resample* documentation for more information.
- **reduce_data** (*bool*) – Reduce data by matching the input and output areas and slicing the data arrays (default: True)
- **resample_kwargs** – Remaining keyword arguments to pass to individual resampler classes. See the individual resampler class documentation *here* for available arguments.

save_dataset(*dataset_id*, *filename=None*, *writer=None*, *overlay=None*, *decorate=None*, *compute=True*, ***kwargs*)

Save the `dataset_id` to file using `writer`.

Parameters

- **dataset_id** (*str or Number or* DataID *or* DataQuery) – Identifier for the dataset to save to disk.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.
- **writer** (*str*) – Name of writer to use when writing data to disk. Default to `"geotiff"`. If not provided, but `filename` is provided then the filename's extension is used to determine the best writer to use.
- **overlay** (*dict*) – See *satpy.writers.add_overlay()*. Only valid for "image" writers like *geotiff* or *simple_image*.
- **decorate** (*dict*) – See *satpy.writers.add_decorate()*. Only valid for "image" writers like *geotiff* or *simple_image*.
- **compute** (*bool*) – If *True* (default), compute all of the saves to disk. If *False* then the return value is either a Dask Delayed object or two lists to be passed to a *dask.array.store* call. See return values below for more details.
- **kwargs** – Additional writer arguments. See *Writing* for more information.

Returns

Value returned depends on *compute*. If *compute* is *True* then the return value is the result of computing a Dask Delayed object or running `dask.array.store()`. If *compute* is *False* then the returned value is either a Dask Delayed object that can be computed using *delayed.compute()* or a tuple of (source, target) that should be passed to `dask.array.store()`.

If target is provided the the caller is responsible for calling *target.close()* if the target has this method.

save_datasets(*writer=None*, *filename=None*, *datasets=None*, *compute=True*, ***kwargs*)

Save requested datasets present in a scene to disk using `writer`.

Note that dependency datasets (those loaded solely to create another and not requested explicitly) that may be contained in this Scene will not be saved by default. The default datasets are those explicitly requested through `.load` and exist in the Scene currently. Specify dependency datasets using the `datasets` keyword argument.

Parameters

- **writer** (*str*) – Name of writer to use when writing data to disk. Default to `"geotiff"`. If not provided, but `filename` is provided then the filename's extension is used to determine the best writer to use.
- **filename** (*str*) – Optionally specify the filename to save this dataset to. It may include string formatting patterns that will be filled in by dataset attributes.
- **datasets** (*iterable*) – Limit written products to these datasets. Elements can be string name, a wavelength as a number, a DataID, or DataQuery object.
- **compute** (*bool*) – If *True* (default), compute all of the saves to disk. If *False* then the return value is either a Dask Delayed object or two lists to be passed to a *dask.array.store* call. See return values below for more details.
- **kwargs** – Additional writer arguments. See *Writing* for more information.

Returns

Value returned depends on *compute* keyword argument. If *compute* is *True* the value is the result of a either a *dask.array.store* operation or a Dask Delayed compute, typically this is *None*. If *compute* is *False* then the result is either a Dask Delayed object that can be computed with *delayed.compute()* or a two element tuple of sources and targets to be passed to `dask.array.store()`. If *targets* is provided then it is the caller's responsibility to close any objects that have a "close" method.

property sensor_names: set[str]

Return sensor names for the data currently contained in this Scene.

Sensor information is collected from data contained in the Scene whether loaded from a reader or generated as a composite with *load()* or added manually using `scn["name"] = data_arr`). Sensor information is also collected from any loaded readers. In some rare cases this may mean that the reader includes sensor information for data that isn't actually loaded or even available.

show(*dataset_id*, *overlay=None*)

Show the *dataset* on screen as an image.

Show dataset on screen as an image, possibly with an overlay.

Parameters

- **dataset_id** (*DataID, DataQuery or str*) – Either a DataID, a DataQuery or a string, that refers to a data array that has been previously loaded using Scene.load.
- **overlay** (*dict, optional*) – Add an overlay before showing the image. The keys/values for this dictionary are as the arguments for *add_overlay()*. The dictionary should contain at least the key `"coast_dir"`, which should refer to a top-level directory containing shapefiles. See the pycoast package documentation for coastline shapefile installation instructions.

slice(*key*)

Slice Scene by dataset index.

Note: DataArrays that do not have an `area` attribute will not be sliced.

property start_time

Return the start time of the contained data.

If no data is currently contained in the Scene then loaded readers will be consulted.

to_geoviews(*gvtype=None*, *datasets=None*, *kdims=None*, *vdims=None*, *dynamic=False*)

Convert satpy Scene to geoviews.

Parameters

- **gvtype** (*gv plot type*) – One of gv.Image, gv.LineContours, gv.FilledContours, gv.Points Default to `geoviews.Image`. See Geoviews documentation for details.
- **datasets** (*list*) – Limit included products to these datasets
- **kdims** (*list of str*) – Key dimensions. See geoviews documentation for more information.
- **vdims** (*list of str, optional*) – Value dimensions. See geoviews documentation for more information. If not given defaults to first data variable
- **dynamic** (*bool, optional*) – Load and compute data on-the-fly during visualization. Default is `False`. See https://holoviews.org/user_guide/Gridded_Datasets.html#working-with-xarray-data-types for more information. Has no effect when data to be visualized only has 2 dimensions (y/x or longitude/latitude) and doesn't require grouping via the Holoviews `groupby` function.

Returns: geoviews object

to_xarray(*datasets=None*, *header_attrs=None*, *exclude_attrs=None*, *flatten_attrs=False*, *pretty=True*, *include_lonlats=True*, *epoch=None*, *include_orig_name=True*, *numeric_name_prefix='CHANNEL_'*)

Merge all xr.DataArray(s) of a satpy.Scene to a CF-compliant xarray object.

If all Scene DataArrays are on the same area, it returns an xr.Dataset. If Scene DataArrays are on different areas, currently it fails, although in future we might return a DataTree object, grouped by area.

Parameters

- **(iterable)** (*datasets*) – List of Satpy Scene datasets to include in the output xr.Dataset. Elements can be string name, a wavelength as a number, a DataID, or DataQuery object. If None (the default), it include all loaded Scene datasets.
- **header_attrs** – Global attributes of the output xr.Dataset.
- **(str)** (*numeric_name_prefix*) – Reference time for encoding the time coordinates (if available). Example format: "seconds since 1970-01-01 00:00:00". If None, the default reference time is defined using "from satpy.cf.coords import EPOCH"
- **(bool)** (*pretty*) – If True, flatten dict-type attributes.
- **(list)** (*exclude_attrs*) – List of xr.DataArray attribute names to be excluded.
- **(bool)** – If True, it includes 'latitude' and 'longitude' coordinates. If the 'area' attribute is a SwathDefinition, it always includes latitude and longitude coordinates.

- **(bool)** – Don't modify coordinate names, if possible. Makes the file prettier, but possibly less consistent.
- **(bool).** (*include_orig_name*) – Include the original dataset name as a variable attribute in the xr.Dataset.
- **(str)** – Prefix to add the each variable with name starting with a digit. Use '' or None to leave this out.
- **Returns** –
- **-------** –
- **ds** – A CF-compliant xr.Dataset
- **xr.Dataset** – A CF-compliant xr.Dataset

to_xarray_dataset(*datasets=None*)

Merge all xr.DataArrays of a scene to a xr.DataSet.

Parameters
datasets (*list*) – List of products to include in the `xarray.Dataset`

Returns: `xarray.Dataset`

unload(*keepables=None*)

Unload all unneeded datasets.

Datasets are considered unneeded if they weren't directly requested or added to the Scene by the user or they are no longer needed to generate composites that have yet to be generated.

Parameters
keepables (*iterable*) – DataIDs to keep whether they are needed or not.

values()

Get values for the underlying data container.

property wishlist

Return a copy of the wishlist.

satpy.scene.**_aggregate_data_array**(*data_array*, *func*, ***coarsen_kwargs*)

Aggregate xr.DataArray.

satpy.scene.**_get_area_resolution**(*area*)

Attempt to retrieve resolution from AreaDefinition.

satpy.utils module

Module defining various utilities.

exception satpy.utils.**PerformanceWarning**

Bases: `Warning`

Warning raised when there is a possible performance impact.

class satpy.utils.**_WarningManager**

Bases: `object`

Class to handle switching warnings on and off.

filt = None

satpy.utils.**_all_dims_same_size**(*data_arrays: tuple[DataArray, ...]*) → bool

satpy.utils.**_check_file_protocols**(*filenames*, *storage_options*)

satpy.utils.**_check_file_protocols_for_dicts**(*filenames*, *storage_options*)

satpy.utils.**_check_import**(*module_names*)

Import the specified modules and provide status.

satpy.utils.**_check_yaml_configs**(*configs*, *key*)

Get a diagnostic for the yaml *configs*.

key is the section to look for to get a name for the config at hand.

satpy.utils.**_filenames_to_fsfile**(*filenames*, *storage_options*)

satpy.utils.**_get_chunk_pixel_size**()

Compute the maximum chunk size from PYTROLL_CHUNK_SIZE.

satpy.utils.**_get_first_available_item**(*data_dict*, *possible_keys*)

satpy.utils.**_get_prefix_order_by_preference**(*prefixes*, *preference*)

satpy.utils.**_get_pytroll_chunk_size**()

satpy.utils.**_get_sat_altitude**(*data_arr*, *key_prefixes*)

satpy.utils.**_get_sat_lonlat**(*data_arr*, *key_prefixes*)

satpy.utils.**_get_satpos_from_platform_name**(*cth_dataset*)

Get satellite position if no orbital parameters in metadata.

Some cloud top height datasets lack orbital parameter information in metadata. Here, orbital parameters are calculated based on the platform name and start time, via Two Line Element (TLE) information.

Needs pyorbital, skyfield, and astropy to be installed.

satpy.utils.**_get_storage_dictionary_options**(*reader_kwargs*)

satpy.utils.**_get_sunz_corr_li_and_shibata**(*cos_zen*)

satpy.utils.**_sort_files_to_local_remote_and_fsfiles**(*filenames*)

satpy.utils.**angle2xyz**(*azi*, *zen*)

Convert azimuth and zenith to cartesian.

satpy.utils.**atmospheric_path_length_correction**(*data*, *cos_zen*, *limit=88.0*, *max_sza=95.0*)

Perform Sun zenith angle correction.

This function uses the correction method proposed by Li and Shibata (2006): https://doi.org/10.1175/JAS3682.1

The correction is limited to `limit` degrees (default: 88.0 degrees). For larger zenith angles, the correction is the same as at the `limit` if `max_sza` is *None*. The default behavior is to gradually reduce the correction past `limit` degrees up to `max_sza` where the correction becomes 0. Both `data` and `cos_zen` should be 2D arrays of the same shape.

satpy.utils.**check_satpy**(*readers=None*, *writers=None*, *extras=None*)

Check the satpy readers and writers for correct installation.

Parameters

- **readers** (*list or None*) – Limit readers checked to those specified

- **writers** (*list or None*) – Limit writers checked to those specified
- **extras** (*list or None*) – Limit extras checked to those specified

Returns: bool
True if all specified features were successfully loaded.

satpy.utils.**convert_remote_files_to_fsspec**(*filenames*, *storage_options=None*)

Check filenames for transfer protocols, convert to FSFile objects if possible.

satpy.utils.**debug**(*deprecation_warnings=True*)

Context manager to temporarily set debugging on.

Example:

```
>>> with satpy.utils.debug():
...     code_here()
```

Parameters
deprecation_warnings (*Optional[bool]*) – Switch on deprecation warnings. Defaults to True.

satpy.utils.**debug_off**()

Turn debugging logging off.

This disables both debugging logging and the global visibility of deprecation warnings.

satpy.utils.**debug_on**(*deprecation_warnings=True*)

Turn debugging logging on.

Sets up a StreamHandler to to *sys.stderr* at debug level for all loggers, such that all debug messages (and log messages with higher severity) are logged to the standard error stream.

By default, since Satpy 0.26, this also enables the global visibility of deprecation warnings. This can be suppressed by passing a false value.

Parameters
deprecation_warnings (*Optional[bool]*) – Switch on deprecation warnings. Defaults to True.

Returns
None

satpy.utils.**deprecation_warnings_off**()

Switch off deprecation warnings.

satpy.utils.**deprecation_warnings_on**()

Switch on deprecation warnings.

satpy.utils.**find_in_ancillary**(*data*, *dataset*)

Find a dataset by name in the ancillary vars of another dataset.

Parameters

- **data** (*xarray.DataArray*) – Array for which to search the ancillary variables
- **dataset** (*str*) – Name of ancillary variable to look for.

satpy.utils.**get_chunk_size_limit**(*dtype=<class 'float'>*)

Compute the chunk size limit in bytes given *dtype* (float by default).

It is derived from PYTROLL_CHUNK_SIZE if defined (although deprecated) first, from dask config's *array.chunk-size* then. It defaults to 128MiB.

Returns
The recommended chunk size in bytes.

satpy.utils.**get_dask_chunk_size_in_bytes**()

Get the dask configured chunk size in bytes.

satpy.utils.**get_legacy_chunk_size**()

Get the legacy chunk size.

This function should only be used while waiting for code to be migrated to use satpy.utils.get_chunk_size_limit instead.

satpy.utils.**get_logger**(*name*)

Return logger with null handler added if needed.

satpy.utils.**get_satpos**(*data_arr: DataArray, preference: str | None = None, use_tle: bool = False*) → tuple[float, float, float]

Get satellite position from dataset attributes.

Parameters

- **data_arr** – DataArray object to access `.attrs` metadata from.
- **preference** – Optional preference for one of the available types of position information. If not provided or `None` then the default preference is:
  - Longitude & Latitude: nadir, actual, nominal, projection
  - Altitude: actual, nominal, projection

  The provided `preference` can be any one of these individual strings (nadir, actual, nominal, projection). If the preference is not available then the original preference list is used. A warning is issued when projection values have to be used because nothing else is available and it wasn't provided as the `preference`.
- **use_tle** – If true, try to obtain position via satellite name and TLE if it can't be determined otherwise. This requires pyorbital, skyfield, and astropy to be installed and may need network access to obtain the TLE. Note that even if `use_tle` is true, the TLE will not be used if the dataset metadata contain the satellite position directly.

Returns
Geodetic longitude, latitude, altitude [km]

satpy.utils.**get_storage_options_from_reader_kwargs**(*reader_kwargs*)

Read and clean storage options from reader_kwargs.

satpy.utils.**ignore_invalid_float_warnings**()

Ignore warnings generated for working with NaN/inf values.

Numpy and dask sometimes don't like NaN or inf values in normal function calls. This context manager hides/ignores them inside its context.

Examples

Use around numpy operations that you expect to produce warnings:

```
with ignore_invalid_float_warnings():
    np.nanmean(np.nan)
```

satpy.utils.**ignore_pyproj_proj_warnings**()

Wrap operations that we know will produce a PROJ.4 precision warning.

Only to be used internally to Pyresample when we have no other choice but to use PROJ.4 strings/dicts. For example, serialization to YAML or other human-readable formats or testing the methods that produce the PROJ.4 versions of the CRS.

satpy.utils.**import_error_helper**(*dependency_name*)

Give more info on an import error.

satpy.utils.**in_ipynb**()

Check if we are in a jupyter notebook.

satpy.utils.**logging_off**()

Turn logging off.

satpy.utils.**logging_on**(*level=30*)

Turn logging on.

satpy.utils.**lonlat2xyz**(*lon*, *lat*)

Convert lon lat to cartesian.

For a sphere with unit radius, convert the spherical coordinates longitude and latitude to cartesian coordinates.

Parameters

- **lon** (*number or array of numbers*) – Longitude in °.
- **lat** (*number or array of numbers*) – Latitude in °.

Returns

(x, y, z) Cartesian coordinates [1]

satpy.utils.**normalize_low_res_chunks**(*chunks: tuple[int | Literal['auto'], ...], input_shape: tuple[int, ...], previous_chunks: tuple[int, ...], low_res_multipliers: tuple[int, ...], input_dtype: dtype[Any] | None | type[Any] | _SupportsDType[dtype[Any]] | str | tuple[Any, int] | tuple[Any, SupportsIndex | Sequence[SupportsIndex]] | list[Any] | _DTypeDict | tuple[Any, Any]*) → tuple[int, ...]

Compute dask chunk sizes based on data resolution.

First, chunks are computed for the highest resolution version of the data. This is done by multiplying the input array shape by the `low_res_multiplier` and then using Dask's utility functions and configuration to produce a chunk size to fit into a specific number of bytes. See Chunks for more information. Next, the same multiplier is used to reduce the high resolution chunk sizes to the lower resolution of the input data. The end result of reading multiple resolutions of data is that each dask chunk covers the same geographic region. This also means replicating or aggregating one resolution and then combining arrays should not require any rechunking.

Parameters

- **chunks** – Requested chunk size for each dimension. This is passed directly to dask. Use `"auto"` for dimensions that should have chunks determined for them, `-1` for dimensions that

should be whole (not chunked), and 1 or any other positive integer for dimensions that have a known chunk size beforehand.

- **input_shape** – Shape of the array to compute dask chunk size for.
- **previous_chunks** – Any previous chunking or structure of the data. This can also be thought of as the smallest number of high (fine) resolution elements that make up a single "unit" or chunk of data. This could be a multiple or factor of the scan size for some instruments and/or could be based on the on-disk chunk size. This value ensures that chunks are aligned to the underlying data structure for best performance. On-disk chunk sizes should be multiplied by the largest low resolution multiplier if it is the same between all files (ex. 500m file has 226 chunk size, 1km file has 226 chunk size, etc).. Otherwise, the resulting low resolution chunks may not be aligned to the on-disk chunks. For example, if dask decides on a chunk size of 226 * 3 for 500m data, that becomes 226 * 3 / 2 for 1km data which is not aligned to the on-disk chunk size of 226.
- **low_res_multipliers** – Number of high (fine) resolution pixels that fit in a single low (coarse) resolution pixel.
- **input_dtype** – Dtype for the final unscaled array. This is usually 32-bit float (`np.float32`) or 64-bit float (`np.float64`) for non-category data. If this doesn't represent the final data type of the data then the final size of chunks in memory will not match the user's request via dask's `array.chunk-size` configuration. Sometimes it is useful to keep this as a single dtype for all reading functionality (ex. `np.float32`) in order to keep all read variable chunks the same size regardless of dtype.

Returns

A tuple where each element is the chunk size for that axis/dimension.

`satpy.utils.`**`proj_units_to_meters`**(*proj_str*)

Convert projection units from kilometers to meters.

`satpy.utils.`**`recursive_dict_update`**(*d*, *u*)

Recursive dictionary update.

Copied from:

http://stackoverflow.com/questions/3232943/update-value-of-a-nested-dictionary-of-varying-depth

`satpy.utils.`**`trace_on`**()

Turn trace logging on.

`satpy.utils.`**`unify_chunks`**(**data_arrays: DataArray*) → tuple[DataArray, ...]

Run `xarray.unify_chunks()` if input dimensions are all the same size.

This is mostly used in `satpy.composites.CompositeBase` to safe guard against running `dask.array.core.map_blocks()` with arrays of different chunk sizes. Doing so can cause unexpected results or errors. However, xarray's `unify_chunks` will raise an exception if dimensions of the provided DataArrays are different sizes. This is a common case for Satpy. For example, the "bands" dimension may be 1 (L), 2 (LA), 3 (RGB), or 4 (RGBA) for most compositor operations that combine other composites together.

`satpy.utils.`**`xyz2angle`**(*x*, *y*, *z*, *acos=False*)

Convert cartesian to azimuth and zenith.

`satpy.utils.`**`xyz2lonlat`**(*x*, *y*, *z*, *asin=False*)

Convert cartesian to lon lat.

For a sphere with unit radius, convert cartesian coordinates to spherical coordinates longitude and latitude.

Parameters

- **x** (*number or array of numbers*) – x-coordinate, unitless
- **y** (*number or array of numbers*) – y-coordinate, unitless
- **z** (*number or array of numbers*) – z-coordinate, unitless
- **asin** (*optional, bool*) – If true, use arcsin for calculations. If false, use arctan2 for calculations.

Returns
Longitude and latitude in °.

Return type
(lon, lat)

satpy.version module

Module contents

Satpy Package initializer.

2.16 FAQ

Below you'll find frequently asked questions, performance tips, and other topics that don't really fit in to the rest of the Satpy documentation.

If you have any other questions that aren't answered here feel free to make an issue on GitHub or talk to us on the Slack team or mailing list. See the *contributing* documentation for more information.

Topics

2.16.1 How can I speed up creation of composites that need resampling?

Satpy performs some initial image generation on the fly, but for composites that need resampling (like the `true_color` composite for GOES/ABI) the data must be resampled to a common grid before the final image can be produced, as the input channels are at differing spatial resolutions. In such cases, you may see a substantial performance improvement by passing `generate=False` when you load your composite:

```
scn = Scene(filenames=filenames, reader='abi_l1b')
scn.load(['true_color'], generate=False)
scn_res = scn.resample(...)
```

By default, `generate=True` which means that Satpy will create as many composites as it can with the available data. In some cases this could mean a lot of intermediate products (ex. rayleigh corrected data using dynamically generated angles for each band resolution) that will then need to be resampled. By setting `generate=False`, Satpy will only load the necessary dependencies from the reader, but not attempt generating any composites or applying any modifiers. In these cases this can save a lot of time and memory as only one resolution of the input data have to be processed. Note that this option has no effect when only loading data directly from readers (ex. IR/visible bands directly from the files) and where no composites or modifiers are used. Also note that in cases where most of your composite inputs are already at the same resolution and you are only generating a limited number of composites, `generate=False` may actually hurt performance.

2.16.2 Why is Satpy slow on my powerful machine?

Satpy depends heavily on the dask library for its performance. However, on some systems dask's default settings can actually hurt performance. By default dask will create a "worker" for each logical core on your system. In most systems you have twice as many logical cores (also known as threaded cores) as physical cores. Managing and communicating with all of these workers can slow down dask, especially when they aren't all being used by most Satpy calculations. One option is to limit the number of workers by doing the following at the **top** of your python code:

```
import dask
dask.config.set(num_workers=8)
# all other Satpy imports and code
```

This will limit dask to using 8 workers. Typically numbers between 4 and 8 are good starting points. Number of workers can also be set from an environment variable before running the python script, so code modification isn't necessary:

```
DASK_NUM_WORKERS=4 python myscript.py
```

Similarly, if you have many workers processing large chunks of data you may be using much more memory than you expect. If you limit the number of workers *and* the size of the data chunks being processed by each worker you can reduce the overall memory usage. Default chunk size can be configured in Satpy by using the following around your code:

```
with dask.config.set("array.chunk-size": "32MiB"):
  # your code here
```

For more information about chunk sizes in Satpy, please refer to the *Data Chunks* section in *Overview*.

Note: The PYTROLL_CHUNK_SIZE variable is pending deprecation, so the above-mentioned dask configuration parameter should be used instead.

2.16.3 Why multiple CPUs are used even with one worker?

Many of the underlying Python libraries use math libraries like BLAS and LAPACK written in C or FORTRAN, and they are often compiled to be multithreaded. If necessary, it is possible to force the number of threads they use by setting an environment variable:

```
OMP_NUM_THREADS=2 python myscript.py
```

2.16.4 What is the difference between number of workers and number of threads?

The above questions handle two different stages of parallellization: Dask workers and math library threading.

The number of Dask workers affect how many separate tasks are started, effectively telling how many chunks of the data are processed at the same time. The more workers are in use, the higher also the memory usage will be.

The number of threads determine how much parallel computations are run for the chunk handled by each worker. This has minimal effect on memory usage.

The optimal setup is often a mix of these two settings, for example

```
DASK_NUM_WORKERS=2 OMP_NUM_THREADS=4 python myscript.py
```

would create two workers, and each of them would process their chunk of data using 4 threads when calling the underlying math libraries.

2.16.5 How do I avoid memory errors?

If your environment is using many dask workers, it may be using more memory than it needs to be using. See the "Why is Satpy slow on my powerful machine?" question above for more information on changing Satpy's memory usage.

2.16.6 Reducing GDAL output size?

Sometimes GDAL-based products, like geotiffs, can be much larger than expected. This can be caused by GDAL's internal memory caching conflicting with dask's chunking of the data arrays. Modern versions of GDAL default to using 5% of available memory for holding on to data before compressing it and writing it to disk. On more powerful systems (~128GB of memory) this is usually not a problem. However, on low memory systems this may mean that GDAL is only compressing a small amount of data before writing it to disk. This results in poor compression and large overhead from the many small compressed areas. One solution is to increase the chunk size used by dask but this can result in poor performance during computation. Another solution is to increase `GDAL_CACHEMAX`, an environment variable that GDAL uses. This defaults to `"5%"`, but can be increased:

```
export GDAL_CACHEMAX="15%"
```

For more information see GDAL's documentation.

2.16.7 How do I use multi-threaded compression when writing GeoTIFFs?

The GDAL library's GeoTIFF driver has a lot of options for changing how your GeoTIFF is formatted and written. One of the most important ones when it comes to writing GeoTIFFs is using multiple threads to compress your data. By default Satpy will use DEFLATE compression which can be slower to compress than other options out there, but faster to read. GDAL gives us the option to control the number of threads used during compression by specifying the `num_threads` option. This option defaults to `1`, but it is recommended to set this to at least the same number of dask workers you use. Do this by adding `num_threads` to your *save_dataset* or *save_datasets* call:

```
scn.save_datasets(base_dir='/tmp', num_threads=8)
```

Satpy also stores our data as "tiles" instead of "stripes" which is another way to get more efficient compression of our GeoTIFF image. You can disable this with `tiled=False`.

See the GDAL GeoTIFF documentation for more information on the creation options available including other compression choices.

Table 7: Satpy Readers

Description	Reader name	Status	fsspec support
GOES-R ABI imager Level 1b data in netcdf format	abi_l1b	Nominal	true
SCMI ABI L1B in netCDF4 format	abi_l1b_scmi	Beta	false
GOES-R ABI Level 2 products in netCDF4 format	abi_l2_nc	Beta	true
NOAA Level 2 ACSPO SST data in netCDF4 format	acspo	Nominal	false
FY-4A AGRI L1 data in HDF5 format	agri_fy4a_l1	Beta	false
	agri_fy4b_l1		false
Himawari (8 + 9) AHI Level 1 (HRIT)	ahi_hrit	Nominal	false
Himawari (8 + 9) AHI Level 1b (HSD)	ahi_hsd	Nominal	false
Himawari (8 + 9) AHI Level 1b (gridded)	ahi_l1b_gridded_bin	Nominal	false
Himawari-8/9 AHI Level 2 products in netCDF4 format from NOAA enterprise	ahi_l2_nc	Beta	true
GEO-KOMPSAT-2 AMI Level 1b	ami_l1b	Beta	true
GCOM-W1 AMSR2 data in HDF5 format	amsr2_l1b	Nominal	false
GCOM-W1 AMSR2 Level 2 (HDF5)	amsr2_l2	Beta	false
GCOM-W1 AMSR2 Level 2 GAASP (NetCDF4)	amsr2_l2_gaasp	Beta	false
AAPP L1C AMSU-B format	amsub_l1c_aapp	Beta	false
METOP ASCAT Level 2 SOILMOISTURE BUFR	ascat_l2_soilmoisture_	Defunct	false
S-NPP and JPSS-1 ATMS L1B (NetCDF4)	atms_l1b_nc	Beta	false
S-NPP and JPSS ATMS SDR (hdf5)	atms_sdr_hdf5	Beta	false
NOAA 15 to 19, Metop A to C AVHRR data in AAPP format	avhrr_l1b_aapp	Nominal	false
Metop A to C AVHRR in native level 1 format	avhrr_l1b_eps	Nominal	false
Tiros-N, NOAA 7 to 19 AVHRR data in GAC and LAC format	avhrr_l1b_gaclac	Nominal	false
NOAA 15 to 19 AVHRR data in raw HRPT format	avhrr_l1b_hrpt	Alpha	false
EUMETCSAT GAC FDR NetCDF4	avhrr_l1c_eum_gac_	Defunct	false
Callipso Caliop Level 2 Cloud Layer data (v3) in EOS-hdf4 format	caliop_l2_cloud	Alpha	false
The Clouds from AVHRR Extended (CLAVR-x)	clavrx	Nominal	false
CMSAF CLAAS-2 data for SEVIRI-derived cloud products	cmsaf-claas2_l2_nc	Beta	false
Electro-L N2 MSU-GS data in HRIT format	electrol_hrit	Nominal	false
DSCOVR EPIC L1b hdf5	epic_l1b_h5	Beta	false
MTG FCI Level-1c NetCDF	fci_l1c_nc	Beta for full-disc FDHSI and HRFI, RSS not supported yet	true
MTG FCI L2 data in netCDF4 format	fci_l2_nc	Alpha	false

continues on next page

Table 7 – continued from previous page

Description	Reader name	Status	fsspec support
Generic Images e.g. GeoTIFF	generic_image	Nominal	false
GEOstationary Cloud Algorithm Test-bed	geocat	Nominal	false
Meteosat Second Generation Geostationary Earth Radiation Budget L2 High-Resolution	gerb_l2_hr_h5	Beta	false
	ghi_l1		false
Sentinel-3 SLSTR SST data in netCDF4 format	ghrsst_l2	Beta	false
GOES-R GLM Level 2	glm_l2	Beta	false
GMS-5 VISSR Level 1b	gms5-vissr_l1b	Alpha	true
GOES Imager Level 1 (HRIT)	goes-imager_hrit	Nominal	false
GOES Imager Level 1 (netCDF)	goes-imager_nc	Beta	false
GPM IMERG level 3 precipitation data in HDF5 format	gpm_imerg	Nominal	false
GRIB2 format	grib	Beta	false
Hydrology SAF products in GRIB format	hsaf_grib	Beta, only h03, h03b, h05 and h05b currently supported	false
Hydrology SAF products in HDF5 format	hsaf_h5	Beta, only h10 currently supported	false
HY-2B Scatterometer level 2b data in HDF5 format from both EUMETSAT and NSOAS	hy2_scat_l2b_h5	Beta	false
IASI Level 2 data in HDF5 format	iasi_l2	Alpha	false
IASI All Sky Temperature and Humidity Profiles - Climate Data Record Release 1.1 - Metop-A and -B	iasi_l2_cdr_nc	Alpha	True
METOP IASI Level 2 SO2 in BUFR format	iasi_l2_so2_bufr	Beta	false
EPS-SG ICI L1B Radiance (NetCDF4)	ici_l1b_nc	Beta	false
Insat 3d IMG L1B HDF5	insat3d_img_l1b_h5	Beta, navigation still off	false
MTSAT-1R JAMI Level 1 data in JMA HRIT format	jami_hrit	Beta	false
LI Level-2 NetCDF Reader	li_l2_nc	Beta	false
AAPP MAIA VIIRS and AVHRR products in HDF5 format	maia	Nominal	false
	meris_nc_sen3		false
MERSI-2 L1B data in HDF5 format	mersi2_l1b	Beta	false
	mersi_ll_l1b		false
AAPP L1C in MHS format	mhs_l1c_aapp	Nominal	false
MIMIC Total Precipitable Water Product Reader in netCDF format	mimicTPW2_comp	Beta	false
MiRS Level 2 Precipitation and Surface Swath Product Reader in netCDF4 format	mirs	Beta	false

continues on next page

Table 7 – continued from previous page

Description	Reader name	Status	fsspec support
Terra and Aqua MODIS data in EOS-hdf4 level-1 format as produced by IMAPP and IPOPP or downloaded from LAADS	modis_l1b	Nominal	false
MODIS Level 2 (mod35) data in HDF-EOS format	modis_l2	Beta	false
MODIS Level 3 (mcd43) data in HDF-EOS format	modis_l3	Beta	false
Sentinel-2 A and B MSI data in SAFE format	msi_safe	Nominal	false
Arctica-M (N1) MSU-GS/A data in HDF5 format	msu_gsa_l1b	Beta	false
MTSAT-2 Imager Level 1 data in JMA HRIT format	mtsat2-imager_hrit	Beta	false
MFG (Meteosat 2 to 7) MVIRI data in netCDF format (FIDUCEO FCDR)	mviri_l1b_fiduceo_n	Beta	false
EPS-SG MWI L1B Radiance (NetCDF4)	mwi_l1b_nc	Beta	false
EPS-SG MWS L1B Radiance (NetCDF4)	mws_l1b_nc	Beta	false
NUCAPS EDR Retrieval data in NetCDF4 format	nucaps	Nominal	false
NWCSAF GEO 2016 products in netCDF4 format (limited to SEVIRI)	nwcsaf-geo	Alpha	false
NWCSAF GEO 2013 products in HDF5 format (limited to SEVIRI)	nwcsaf-msg2013-hdf5	Defunct	false
NWCSAF PPS 2014, 2018 products in netCDF4 format	nwcsaf-pps_nc	Alpha, only standard swath based ouput supported (remapped netCDF and CPP products not supported yet)	false
Ocean color CCI Level 3S data reader	oceancolor-cci_l3_nc	Nominal	false
Sentinel-3 A and B OLCI Level 1B data in netCDF4 format	olci_l1b	Nominal	true
Sentinel-3 A and B OLCI Level 2 data in netCDF4 format	olci_l2	Nominal	true
OMPS EDR data in HDF5 format	omps_edr	Beta	false
OSI-SAF data in netCDF4 format	osisaf_nc	Beta	true
SAR Level 2 OCN data in SAFE format	safe_sar_l2_ocn	Defunct	false
Sentinel-1 A and B SAR-C data in SAFE format	sar-c_safe	Nominal	false
Reader for CF conform netCDF files written with Satpy	satpy_cf_nc	Nominal	false
Scatsat-1 Level 2b Wind field data in HDF5 format	scatsat1_l2b	defunct	false
SEADAS L2 Chlorphyll A product in HDF4 format	seadas_l2	Beta	false

continues on next page

Table 7 – continued from previous page

Description	Reader name	Status	fsspec support
MSG SEVIRI Level 1b (HRIT)	seviri_l1b_hrit	Nominal	true
MSG SEVIRI Level 1b in HDF format from ICARE (Lille)	seviri_l1b_icare	Defunct	false
MSG (Meteosat 8 to 11) SEVIRI data in native format	seviri_l1b_native	Nominal	false
MSG SEVIRI Level 1b NetCDF4	seviri_l1b_nc	Beta, HRV channel not supported	true
MSG (Meteosat 8 to 11) Level 2 products in BUFR format	seviri_l2_bufr	Alpha	false
MSG (Meteosat 8 to 11) SEVIRI Level 2 products in GRIB2 format	seviri_l2_grib	Nominal	false
Sentinel-3 A and B SLSTR data in netCDF4 format	slstr_l1b	Alpha	false
Sentinel-3 SLSTR Level 2 data in netCDF format	slstr_l2	defunct	false
SMOS level 2 wind data in NetCDF4 format	smos_l2_wind	Beta	false
TROPOMI Level 2 data in NetCDF4 format	tropomi_l2	Beta	false
Vaisala Global Lightning Dataset GLD360 data in ASCII format	vaisala_gld360	Beta	false
EPS-SG Visual Infrafred Imager (VII) Level 1B Radiance data in netCDF4 format	vii_l1b_nc	Beta	false
EPS-SG Visual Infrared Imager (VII) Level 2 data in netCDF4 format	vii_l2_nc	Beta	false
SNPP VIIRS SDR data in HDF5 Compact format	viirs_compact	Nominal	false
	viirs_edr		false
VIIRS EDR Active Fires data in netCDF4 & CSV .txt format	viirs_edr_active_fires	Beta	false
VIIRS EDR Flood data in HDF4 format	viirs_edr_flood	Beta	false
SNPP VIIRS Level 1b data in netCDF4 format	viirs_l1b	Nominal	false
SNPP VIIRS data in HDF5 SDR format	viirs_sdr	Nominal	false
VIIRS Global Area Coverage from VIIRS Reflected Solar Band and Thermal Emission Band data for both Moserate resolution and Imager resolution channels.	viirs_vgac_l1c_nc		false
VIRR data in HDF5 format	virr_l1b	Beta	false

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z