Nansat Documentation

Anton Korosov, Morten W. Hansen, Aleksander Vines, Artem Mois

Jun 15, 2022

CONTENTS:

Nansat is a scientist friendly Python toolbox for processing 2D satellite earth observation data.

The main **goal** of Nansat is to facilitate:

- easy development and testing of scientific algorithms,
- easy analysis of geospatial data, and
- efficient operational processing.

You can also find a detailed description of Nansat in our paper published in Journal of Open Research Software in 2016.

… and you can join the mailing list.

CHAPTER

ONE

INSTALLATION

1.1 Quickstart

The fastest way to install nansat:

- Install Miniconda on your platform of choice.

```
# create environment with key requirements
conda create -y -n py3nansat gdal=2.4.2 numpy pillow netcdf4 scipy
# activate environment
source activate py3nansat
# install nansat
pip instal nansat
```

Nansat is now installed. For more details and other methods of installing Nansat, see below.

1.2 Requirements

Nansat requires the following packages:

- Python 2.7 or higher
- Numpy >=1.11.3
- GDAL >=2.2.3
- Pillow >=4.0.0
- netCDF4 >=1.3.1
- py-thesaurus-interface

The following packages are optional:

- Scipy 0.18.1
  - Some mappers will not work without scipy. E.g. *sentinel1_l1*
- Matplotlib >=2.1.1
  - matplotlib is required for Nansat methods *digitize_points()* and *crop_interactive()*
- Basemap >=1.0.8
  - basemap is required in *write_domain_map()*

The most tricky to compile yourself is GDAL and Basemap. But one can find pre-built binaries available for different platforms. We recommend to install all dependencies with Conda, from the conda-forge channel. See instructions on this below.

1.3 Installing Requirements

You have three main options on how to install the requirements. These are described in the following three sections.

1.3.1 Install dependencies from Anaconda

This is the recommended approach for installing dependencies.

- Download Miniconda for your platform of choice.
- Install Miniconda
  - When you install Miniconda on Windows, you will get a new app called "Anaconda Prompt". Run this to access the conda installation.
  - On Linux/OS X use a regular terminal and make sure PATH is set to contain the installation directory as explained by the installer.
- Run the following three commands:
  - *conda create -n nansat Python=3.6*
    - Or use Python version 3.5 or 2.7 if you need those versions.
  - *source activate nansat*
    - On windows you would ommit 'source' and just run *'activate nansat'*
  - *conda install –yes -c conda-forge pythesint numpy scipy=0.18.1 matplotlib basemap netcdf4 gdal pillow urllib3*

1.3.2 Install Pre-built Binaries

One can find pre-built binaries available for different platforms. We do not have an overview over all the possible repositories where you can find binaries. But if you e.g. are on Ubuntu, the following procedure can be used to install dependencies with *apt* and *pip*.

```
sudo apt install virtualenv libgdal1-dev python-dev python-gdal python-numpy python-
↪scipy \
python-matplotlib python-mpltoolkits.basemap python-requests
cd
virtualenv --no-site-packages nansat_env
source ~/nansat_env/bin/activate
export PYTHONPATH=/usr/lib/python2.7/dist-packages/
pip install pythesint pillow netcdf4 urllib3
```

1.3.3 Compile and Build Yourself

If you have the technical expertise to build all dependencies, and need to do it yourself, feel free to do so. If you need some aid, we would recommend you to look at how the corresponding conda-forge feedstocks have been built.

1.4 Installing Nansat

1.4.1 Install Nansat from source

If you want to install Nansat from source, you first need to install all requirements. Then proceed with one of the following methods

Install from git repository

git clone the master (most stable) or develop (cutting edge) branch, and install:

```
git clone https://github.com/nansencenter/nansat.git
checkout master (or develop, or a specific tag or branch)
python setup.py install
```

Nansat will then be added to your site-packages and can be used like any regular Python package.

Install with pip

Run the following command:

```
pip install nansat
```

Nansat will then be added to your site-packages and can be used like any regular Python package.

1.4.2 Special install for Nansat Developers

If you are working directly on the Nansat source, you need to install Nansat in the following way.

Git clone the develop branch (or another branch you are working on), and do:

```
python setup.py build_ext --inplace
```

The pixel functions C module is then compiled but no code is copied to site-packages and no linking is performed. Make sure to follow the Nansat conventions if you want to contribute to Nansat.

In addition to the regular dependencies, developers also need to install nose and mock. This can easily be done with

```
pip install nose mock
```

1.5 Use a self-provisioned Virtual Machine

Another option to install Nansat in a controlled environment is to use a virtual machine. Configuration for Vagrant and Ansible that brings up and provision a VirtualBox machine is provided in Nansat repository. To start the machine you need to install Vagrant and VirtualBox on your computer; clone or download the nansat source code; and start the machine:

```
# download nansat source code
git clone https://github.com/nansencenter/nansat.git
cd nansat

#start virtual machine
vagrant up
```

That's it! The virtual machine will be started and all software will be installed automatically. To start using Nansat you need to log in to the virtual machine and start Python from the conda environment:

```
vagrant ssh
source activate py3nansat
python
```

1.6 Use Docker

Docker is a platform for developers and sysadmins to develop, deploy, and run applications with containers (Get started with Docker). We have developed an image that containes compiled Nansat an a number of Python libraries needed for development and running of Nansat. A user can start using the production version of Nansat Docker image:

```
docker run --rm -it -v /path/to/data:/data akorosov/nansat ipython
```

This will mound directory /path/to/data on your host to the directory /data in the container and launch IPython where Nansat is available.

For developing Nansat you needs access to the code both from the container (to run it) and from the host (to edit it). For this purpose you should clone Nansat repository and do the following steps: 1. Build pixelfunctions inplace

```
docker run --rm -it -v `pwd`:/src akorosov/nansat python setup.py build_ext --inplace
```

2. Run container with mounting of the current directory into /src. In this case Python will use Nansat from /src/nansat (the directory shared between host and container):

```
# launch Python with Nansat in container
docker run --rm -it -v `pwd`:/src akorosov/nansat python

# ...or run nosetests
docker run --rm -it -v `pwd`:/src akorosov/nansat nosetests nansat
```

Alternatively you can run the script *build_containr.sh*. The script will build the image with Python libraries from Anaconda, compile the Nansat code inplace and create a container for running Nansat. You can then start container:

```
docker start -i nansat
# and run nosetests:
(base) root@d1625f2ce873:~# nosetests nansat
```

CHAPTER

TWO

NANSAT TUTORIALS

The package nansat-lectures contains several Jupyter notebooks with examples of how to use Nansat. Unfortunately, we have not been able to keep them fully updated. The most recently updated notebooks should, however, work.

2.1 Nansat: First Steps

2.1.1 Overview

The NANSAT package contains several classes:

- Nansat - open and read satellite data
- Domain - define grid for the region of interest
- Figure - create raster images (PNG, TIF)
- NSR - define spatial reference (SR)

2.1.2 Copy sample data

[1]:
```
import os
import shutil
import nansat
idir = os.path.join(os.path.dirname(nansat.__file__), 'tests', 'data/')
```

2.1.3 Open file with Nansat

[2]:
```
import matplotlib.pyplot as plt
%matplotlib inline


from nansat import Nansat
n = Nansat(idir+'gcps.tif')
```

2.1.4 Read information ABOUT the data (METADATA)

[4]:
```
print(n)
```

```
----------------------------------------
/opt/conda/lib/python3.7/site-packages/nansat-1.2.2-py3.7-linux-x86_64.egg/nansat/tests/
↪data/gcps.tif----------------------------------------
Mapper: genericBand : 1 L_645
  colormap: jet
  dataType: 1
  long_name: Upward spectral radiance
  minmax: 0.000 500
  name: L_645
  short_name: nLw
  SourceBand: 1
  SourceFilename: /opt/conda/lib/python3.7/site-packages/nansat-1.2.2-py3.7-linux-x86_64.
↪egg/nansat/tests/data/gcps.tif
  standard_name: surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water
  time: 2011-08-15 10:05:00
  units: W m-2 m-1 sr-1
  wkv: surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water
Band : 2 L_555
  colormap: jet
  dataType: 1
  long_name: Upward spectral radiance
  minmax: 0.000 500
  name: L_555
  short_name: nLw
  SourceBand: 2
  SourceFilename: /opt/conda/lib/python3.7/site-packages/nansat-1.2.2-py3.7-linux-x86_64.
↪egg/nansat/tests/data/gcps.tif
  standard_name: surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water
  time: 2011-08-15 10:05:00
  units: W m-2 m-1 sr-1
  wkv: surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water
Band : 3 L_469
  colormap: jet
  dataType: 1
  long_name: Upward spectral radiance
  minmax: 0.000 500
  name: L_469
  short_name: nLw
  SourceBand: 3
  SourceFilename: /opt/conda/lib/python3.7/site-packages/nansat-1.2.2-py3.7-linux-x86_64.
↪egg/nansat/tests/data/gcps.tif
  standard_name: surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water
  time: 2011-08-15 10:05:00
  units: W m-2 m-1 sr-1
  wkv: surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water
----------------------------------------
Domain:[200 x 200]
----------------------------------------
Projection(gcps):
```

(continues on next page)

(continued from previous page)

```
GEOGCS["WGS 84",
    DATUM["WGS_1984",
        SPHEROID["WGS 84",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["degree",0.0174532925199433]]
----------------------------------------
Corners (lon, lat):
         ( 28.25,  71.54)  ( 30.87,  71.17)
         ( 27.14,  70.72)  ( 29.68,  70.35)
```

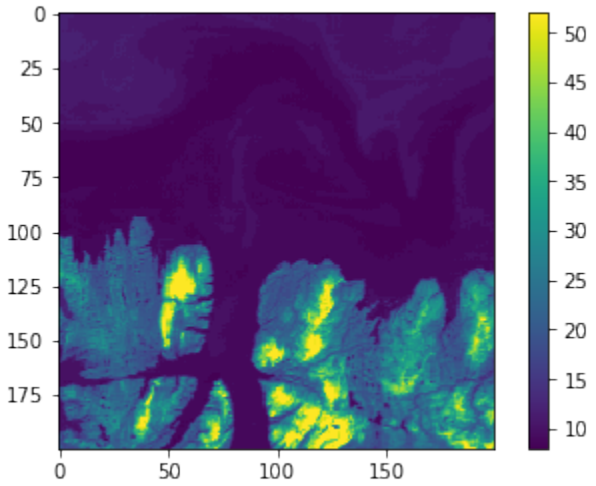
2.1.5 Read the actual DATA

[5]:
```
b1 = n[1]
```

2.1.6 Check what kind of data we have

[6]:
```
%whos
plt.imshow(b1);plt.colorbar()
plt.show()
```

```
Variable   Type       Data/Info
------------------------------
Nansat     type       <class 'nansat.nansat.Nansat'>
b1         ndarray    200x200: 40000 elems, type `uint8`, 40000 bytes
idir       str        /opt/conda/lib/python3.7/<...>64.egg/nansat/tests/data/
n          Nansat     -------------------------<...>0.72)  ( 29.68,  70.35)\n
nansat     module     <module 'nansat' from '/o<...>.egg/nansat/__init__.py'>
os         module     <module 'os' from '/opt/c<...>nda/lib/python3.7/os.py'>
plt        module     <module 'matplotlib.pyplo<...>es/matplotlib/pyplot.py'>
shutil     module     <module 'shutil' from '/o<...>lib/python3.7/shutil.py'>
```



2.1.7 Find where the image is taken

[7]:
```
n.write_figure('map.png', pltshow=True)
```

[7]:
```
<nansat.figure.Figure at 0x7fbc6034fda0>
```

CHAPTER

THREE

PACKAGES AND MODULES

3.1 nansat

3.1.1 nansat package

Subpackages

nansat.mappers package

Submodules

nansat.mappers.envisat module

class nansat.mappers.envisat.**Envisat**

Bases: object

Methods/data shared between Envisat mappers

This class is needed to read awkward N1 format of ENVISAT Mostly it support reading of variables from ADS (additional data sets) which are TIE_POINTS_ADS (MERIS) or GEOLOCATION_GRID_ADS (ASAR)

```
allADSParams = {'ASA_': {'list': {'first_line_incidence_angle': {'dataType':
gdal.GDT_Float32, 'offset': 113, 'units': 'deg'}, 'first_line_lats': {'dataType':
gdal.GDT_Int32, 'offset': 157, 'units': '(10)^-6 deg'}, 'first_line_longs':
{'dataType': gdal.GDT_Int32, 'offset': 201, 'units': '(10)^-6 deg'},
'first_line_samp_numbers': {'dataType': gdal.GDT_Float32, 'offset': 25, 'units':
''}, 'first_line_slant_range_times': {'dataType': gdal.GDT_Float32, 'offset': 69,
'units': 'ns'}, 'last_line_incidence_angle': {'dataType': gdal.GDT_Float32,
'offset': 367, 'units': 'deg'}, 'last_line_lats': {'dataType': gdal.GDT_Int32,
'offset': 411, 'units': '(10)^-6 deg'}, 'last_line_longs': {'dataType':
gdal.GDT_Int32, 'offset': 455, 'units': '(10)^-6 deg'}, 'last_line_samp_numbers':
{'dataType': gdal.GDT_Int32, 'offset': 279, 'units': ''},
'last_line_slant_range_times': {'dataType': gdal.GDT_Float32, 'offset': 323,
'units': 'ns'}, 'num_lines': {'dataType': gdal.GDT_Int16, 'offset': 13, 'units':
''}}, 'name': 'DS_NAME="GEOLOCATION GRID ADS "\n', 'width': 11}, 'MER_': {'list':
{'DME altitude': {'dataType': gdal.GDT_Int32, 'offset': 581, 'units': 'm'}, 'DME
latitude corrections': {'dataType': gdal.GDT_Int32, 'offset': 1149, 'units':
'(10)^-6 deg'}, 'DME longitude corrections': {'dataType': gdal.GDT_Int32, 'offset':
1433, 'units': '(10)^-6 deg'}, 'DME roughness': {'dataType': gdal.GDT_UInt32,
'offset': 865, 'units': 'm'}, 'latitude': {'dataType': gdal.GDT_Int32, 'offset': 13,
'units': '(10)^-6 deg'}, 'longitude': {'dataType': gdal.GDT_Int32, 'offset': 297,
'units': '(10)^-6 deg'}, 'mean sea level pressure': {'dataType': gdal.GDT_UInt16,
'offset': 3137, 'units': 'hPa'}, 'meridional winds': {'dataType': gdal.GDT_Int16,
'offset': 2995, 'units': 'm*s-1'}, 'relative humidity': {'dataType':
gdal.GDT_UInt16, 'offset': 3421, 'units': '%'}, 'sun azimuth angles': {'dataType':
gdal.GDT_Int32, 'offset': 2001, 'units': '(10)^-6 deg'}, 'sun zenith angles':
{'dataType': gdal.GDT_UInt32, 'offset': 1717, 'units': '(10)^-6 deg'}, 'total
ozone': {'dataType': gdal.GDT_UInt16, 'offset': 3279, 'units': 'DU'}, 'viewing
azimuth angles': {'dataType': gdal.GDT_Int32, 'offset': 2569, 'units': '(10)^-6
deg'}, 'viewing zenith angles': {'dataType': gdal.GDT_UInt32, 'offset': 2285,
'units': '(10)^-6 deg'}, 'zonal winds': {'dataType': gdal.GDT_Int16, 'offset': 2853,
'units': 'm*s-1'}}, 'name': 'DS_NAME="Tie points ADS "\n', 'width': 71}}
```

```
structFmt = {gdal.GDT_Int16: '>h', gdal.GDT_UInt16: '>H', gdal.GDT_Int32: '>i',
gdal.GDT_UInt32: '>I', gdal.GDT_Float32: '>f'}
```

```
lonlatNames = {'ASA_': ['first_line_longs', 'first_line_lats'], 'MER_':
['longitude', 'latitude']}
```

setup_ads_parameters(*filename*, *gdalMetadata*)
Select set of params and read offset of ADS

read_offset_from_header(*gadsDSName*)
Read offset of ADS from text header.

Find a location of gadsDSName. Adjust the location with textOffset and read the text at the location. Convert the text to integer and set it into offsetDict.

Returns **offsetDict** – offset of DS, size of DS, number of records, size of record

Return type dictionary

read_binary_line(*offset*, *fmtString*, *length*)
Read line with binary data at given offset

Open file Read values of given size, at given offset, given times Convert to list of values of given format and return

Parameters

- **offset** (*int, start of reading*) –
- **fmtString** (*str, data type format*) –
- **lentgh** (*number of values to read*) –

Returns **binaryValues** – values which are read from the file. the number of elements is length

Return type list

read_scaling_gads(*indeces*)
Read Scaling Factor GADS to get scalings of MERIS L1/L2

Parameters

- **filename** (*string*) –
- **indeces** (*list*) –

Return type list

get_array_from_ADS(*adsName*)
Create VRT with a band from Envisat ADS metadata

Read offsets of the <adsName> ADS. Read 2D matrix of binary values from ADS from file. Read last line ADS (in case of ASAR). Zoom array with ADS data to <zoomSize>. Zooming is needed to create smooth matrices. Array is zoomed to small size because it is stred in memory. Later the VRT with zoomed array is VRT.get_resized_vrt() in order to match the size of the Nansat onject. Create VRT from the ADS array.

Parameters **adsName** (*str*) – name of variable from ADS to read. should match allADSParams

Returns **adsVrt** – vrt with a band created from ADS array

Return type *VRT*

create_VRT_from_ADS(*adsName*, *zoomSize=500*)
Create VRT with a band from Envisat ADS metadata

Read offsets of the <adsName> ADS. Read 2D matrix of binary values from ADS from file. Zoom array with ADS data to <zoomSize>. Zooming is needed to create smooth matrices. Array is zoomed to small size because it is stred in memory. Later the VRT with zoomed array is VRT.get_resized_vrt() in order to match the size of the Nansat object.

Create VRT from the ADS array.

Parameters

- **adsName** (*str*) – name of variable from ADS to read. should match allADSParams
- **zoomSize** (*int, optional, 500*) – size, to which original matrix from ADSR is zoomed using scipy.zoom

Returns **adsVrt**

Return type *VRT*, vrt with a band created from ADS array

get_ads_vrts(*gdalDataset*, *adsNames*, *zoomSize=500*, *step=1*, ***kwargs*)
Create list with VRTs with zoomed and resized ADS arrays

For given names of variables (which should match self.allADSParams): Get VRT with zoomed ADS array Get resized VRT

Parameters

- **gdalDataset** (*GDAL Dataset*) – input dataset
- **adsNames** (*list with strings*) – names of varaiables from self.allADSParams['list']

- **zoomSize** (*int, 500*) – size to which the ADS array will be zoomed by scipy.zoom
- **step** (*int, 1*) – step, at which data will be given

Returns **adsVRTs** – list with resized VRT with zoomed arrays

Return type list with VRT

add_geolocation_from_ads(*gdalDataset*, *zoomSize=500*, *step=1*)
Add geolocation domain metadata to the dataset

Get VRTs with zoomed arrays of lon and lat Create geolocation object and add to the metadata

Parameters

- **gdalDataset** (*GDAL Dataset*) – input dataset
- **zoomSize** (*int, optional, 500*) – size, to which the ADS array will be zoomed using scipy array of this size will be stored in memory
- **step** (*int*) – step of pixel and line in GeolocationArrays. lat/lon grids are generated at that step
- **Modifies** –
- **--------** –
- **metadata** (*Adds Geolocation Array*) –

nansat.mappers.globcolour module

class nansat.mappers.globcolour.**Globcolour**
Bases: object

Mapper for GLOBCOLOR L3M products

varname2wkv = {'BBP_mean':
'volume_backscattering_coefficient_of_radiative_flux_in_sea_water_due_to_suspended_particles',
'CDM_mean':
'volume_absorption_coefficient_of_radiative_flux_in_sea_water_due_to_dissolved_organic_matter',
'CHL1_mean': 'mass_concentration_of_chlorophyll_a_in_sea_water', 'CHL2_mean':
'mass_concentration_of_chlorophyll_a_in_sea_water', 'KD490_mean':
'volume_attenuation_coefficient_of_downwelling_radiative_flux_in_sea_water',
'L412_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L443_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L490_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L510_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L531_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L555_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L620_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L670_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L681_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'L709_mean': 'surface_upwelling_spectral_radiance_in_air_emerging_from_sea_water',
'PAR_mean': 'surface_downwelling_photosynthetic_radiative_flux_in_air'}

make_rrsw_meta_entry(*nlwMetaEntry*)
Make metaEntry for calculation of Rrsw

nansat.mappers.hdf4_mapper module

class nansat.mappers.hdf4_mapper.**HDF4Mapper**(*x_size=1*, *y_size=1*, *metadata=None*, *nomem=False*, ***kwargs*)

Bases: *nansat.vrt.VRT*

find_metadata(*iMetadata*, *iKey*, *default=''*)
Find metadata which has similar key

Parameters

- **iMetadata** (*dict*) – input metadata, usually gdalMetadata
- **iKey** (*str*) – key to search for
- **default** (*str*) – default value

nansat.mappers.mapper_aapp_l1b module

class nansat.mappers.mapper_aapp_l1b.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for AVHRR L1B output from AAPP

nansat.mappers.mapper_aapp_l1c module

class nansat.mappers.mapper_aapp_l1c.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for AVHRR L1C output from AAPP

nansat.mappers.mapper_amsr2_l1r module

class nansat.mappers.mapper_amsr2_l1r.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *GCP_STEP=20*, *MAX_LAT=90*, *MIN_LAT=50*, *resolution='low'*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper for AMSR-2 L1 data

nansat.mappers.mapper_amsr2_l3 module

class nansat.mappers.mapper_amsr2_l3.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper for Level-3 AMSR2 data from https://gcom-w1.jaxa.jp

freqs = [6, 7, 10, 18, 23, 36, 89]

nansat.mappers.mapper_amsre_uham_leadfraction module

class nansat.mappers.mapper_amsre_uham_leadfraction.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

nansat.mappers.mapper_arome module

class nansat.mappers.mapper_arome.**Mapper**(**args*, ***kwargs*)

Bases: *nansat.mappers.mapper_netcdf_cf.Mapper*

nansat.mappers.mapper_asar module

class nansat.mappers.mapper_asar.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*, *nansat.mappers.envisat.Envisat*

VRT with mapping of WKV for ASAR Level 1

See also:

http //envisat.esa.int/handbooks/asar/CNTR6-6-9.htm#eph.asar.asardf.asarrec.ASAR_Geo_Grid_ADSR

nansat.mappers.mapper_ascat module

class nansat.mappers.mapper_ascat.**Mapper**(*filename*, *gdal_dataset*, *metadata*, *quartile=0*, **args*, ***kwargs*)

Bases: *nansat.mappers.scatterometers.Mapper*

Nansat mapper for ASCAT

times()

Get times from time variable

nansat.mappers.mapper_aster_l1a module

class nansat.mappers.mapper_aster_l1a.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *GCP_COUNT=10*, *bandNames=['VNIR_Band1', 'VNIR_Band2', 'VNIR_Band3N']*, *bandWaves=[560, 660, 820]*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper for ASTER L1A VNIR data

nansat.mappers.mapper_aster_l1b module

nansat.mappers.mapper_case2reg module

class nansat.mappers.mapper_case2reg.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *wavelengths=[None, 413, 443, 490, 510, 560, 620, 665, 681, 709, 753, None, 778, 864]*, ***kwargs*)

Bases: *nansat.mappers.mapper_generic.Mapper*

Mapping for the BEAM/Visat output of Case2Regional algorithm

nansat.mappers.mapper_cmems module

nansat.mappers.mapper_cmems.**get_gcmd_keywords_mapping**()

class nansat.mappers.mapper_cmems.**Mapper**(**args*, ***kwargs*)

Bases: *nansat.mappers.mapper_netcdf_cf.Mapper*

time_coverage()

nansat.mappers.mapper_csks module

class nansat.mappers.mapper_csks.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for Cosmo-Skymed

nansat.mappers.mapper_ecmwf_metno module

class nansat.mappers.mapper_ecmwf_metno.**Mapper**(**args*, ***kwargs*)

Bases: *nansat.mappers.mapper_netcdf_cf.Mapper*

nansat.mappers.mapper_emodnet module

class nansat.mappers.mapper_emodnet.**Mapper**(*inputFileName*, *gdalDataset*, *gdalMetadata*, *logLevel=30*, ***kwargs*)

Bases: *nansat.vrt.VRT*

nansat.mappers.mapper_generic module

class nansat.mappers.mapper_generic.**Mapper**(*inputFileName*, *gdalDataset*, *gdalMetadata*, *logLevel=30*, *rmMetadatas=['NETCDF_VARNAME', '_Unsigned', 'ScaleRatio', 'ScaleOffset', 'dods_variable']*, ***kwargs*)

Bases: *nansat.vrt.VRT*

repare_projection(*projection*)

Replace odd symbols in projection string '|' => ','; '&' => ""

add_gcps_from_metadata(*geoMetadata*)

Get GCPs from strings in metadata and insert in dataset

add_gcps_from_variables(*filename*)
Get GCPs from GCPPixel, GCPLine, GCPX, GCPY, GCPZ variables

nansat.mappers.mapper_geostationary module

nansat.mappers.mapper_geostationary.**arrays2LUTString**(*a*, *b*)

class nansat.mappers.mapper_geostationary.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)
Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for Geostationary satellite data

calibration()
get calibration data for Geostationary satellites

Returns **satDict** – calibration data for satellite 'name', 'wavelengths', 'scale', 'offset'

Return type dicts

nansat.mappers.mapper_globcolour_l3b module

class nansat.mappers.mapper_globcolour_l3b.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *latlonGrid=None*, *mask=''*, ***kwargs*)
Bases: *nansat.vrt.VRT*, *nansat.mappers.globcolour.Globcolour*

Create VRT with mapping of WKV for MERIS Level 2 (FR or RR)

nansat.mappers.mapper_globcolour_l3m module

class nansat.mappers.mapper_globcolour_l3m.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)
Bases: *nansat.vrt.VRT*, *nansat.mappers.globcolour.Globcolour*

Mapper for GLOBCOLOR L3M products

nansat.mappers.mapper_goci_l1 module

class nansat.mappers.mapper_goci_l1.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)
Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for MODIS Level 1 (QKM, HKM, 1KM)

nansat.mappers.mapper_hirlam module

class nansat.mappers.mapper_hirlam.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)
Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for HIRLAM

nansat.mappers.mapper_hirlam_wind_netcdf module

class nansat.mappers.mapper_hirlam_wind_netcdf.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *logLevel=30*, ***kwargs*)

Bases: *nansat.vrt.VRT*

nansat.mappers.mapper_kmss module

class nansat.mappers.mapper_kmss.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for KMSS TOA tiff data

nansat.mappers.mapper_landsat module

class nansat.mappers.mapper_landsat.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *resolution='low'*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper for LANDSAT5,6,7,8 .tar.gz or tif files

nansat.mappers.mapper_meris_l1 module

class nansat.mappers.mapper_meris_l1.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *geolocation=False*, *zoomSize=500*, *step=1*, ***kwargs*)

Bases: *nansat.vrt.VRT*, *nansat.mappers.envisat.Envisat*

VRT with mapping of WKV for MERIS Level 1 (FR or RR)

nansat.mappers.mapper_meris_l2 module

class nansat.mappers.mapper_meris_l2.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *geolocation=False*, *zoomSize=500*, *step=1*, ***kwargs*)

Bases: *nansat.vrt.VRT*, *nansat.mappers.envisat.Envisat*

Create VRT with mapping of WKV for MERIS Level 2 (FR or RR)

nansat.mappers.mapper_metno_hires_seaice module

class nansat.mappers.mapper_metno_hires_seaice.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Create VRT with mapping of WKV for Met.no seaice

nansat.mappers.mapper_metno_local_hires_seaice module

class nansat.mappers.mapper_metno_local_hires_seaice.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.mappers.mapper_generic.Mapper*

Create VRT with mapping of WKV for Met.no seaice

nansat.mappers.mapper_mod44w module

class nansat.mappers.mapper_mod44w.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for MOD44W produc (MODIS watermask at 250 m)

nansat.mappers.mapper_modis_l1 module

class nansat.mappers.mapper_modis_l1.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *GCP_COUNT=30*, ***kwargs*)

Bases: *nansat.mappers.hdf4_mapper.HDF4Mapper*

VRT with mapping of WKV for MODIS Level 1 (QKM, HKM, 1KM)

nansat.mappers.mapper_ncep module

class nansat.mappers.mapper_ncep.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for NCEP GFS

nansat.mappers.mapper_ncep_wind module

class nansat.mappers.mapper_ncep_wind.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for NCEP GFS

nansat.mappers.mapper_ncep_wind_online module

class nansat.mappers.mapper_ncep_wind_online.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *outFolder='/home/docs/ncep_gfs_downloads'*, ***kwargs*)

Bases: *nansat.vrt.VRT*, `object`

VRT with mapping of WKV for NCEP GFS

nansat.mappers.mapper_netcdf_cf module

Nansat NetCDF-CF mapper

Check CF-compliance of your files here: http://cfconventions.org/compliance-checker.html

exception nansat.mappers.mapper_netcdf_cf.**ContinueI**
Bases: Exception

class nansat.mappers.mapper_netcdf_cf.**Mapper**(*filename*, *gdal_dataset*, *gdal_metadata*, **args*, ***kwargs*)
Bases: *nansat.vrt.VRT*

input_filename = ''

times()
Get times from time variable

NOTE: This cannot be done with gdal because the time variable is a vector

nansat.mappers.mapper_netcdf_cf_sentinel1 module

class nansat.mappers.mapper_netcdf_cf_sentinel1.**Mapper**(*filename*, *gdal_dataset*, *gdal_metadata*, **args*, ***kwargs*)
Bases: *nansat.mappers.sentinel1.Sentinel1*, *nansat.mappers.mapper_netcdf_cf.Mapper*

add_calibrated_nrcs()

add_nrcs_VV_from_HH()

nansat.mappers.mapper_nora10_local_vpv module

class nansat.mappers.mapper_nora10_local_vpv.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *logLevel=30*, ***kwargs*)
Bases: *nansat.vrt.VRT*

nansat.mappers.mapper_obpg_l2 module

class nansat.mappers.mapper_obpg_l2.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *GCP_COUNT=10*, ***kwargs*)
Bases: *nansat.mappers.obpg.OBPGL2BaseClass*

Mapper for SeaWIFS/MODIS/MERIS/VIIRS L2 data from OBPG

TODO: * Test on SeaWIFS * Test on MODIS Terra

nansat.mappers.mapper_obpg_l2_nc module

class nansat.mappers.mapper_obpg_l2_nc.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *GCP_COUNT=10*, ***kwargs*)
Bases: *nansat.mappers.obpg.OBPGL2BaseClass*

Mapper for SeaWIFS/MODIS/MERIS/VIIRS L2 data from OBPG in NC4 format

nansat.mappers.mapper_obpg_l3 module

class nansat.mappers.mapper_obpg_l3.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper for Level-3 Standard Mapped Image from http://oceancolor.gsfc.nasa.gov

param2wkv = {'CDOM Index':
'volume_absorption_coefficient_of_radiative_flux_in_sea_water_due_to_dissolved_organic_matter',
'Chlorophyll a concentration': 'mass_concentration_of_chlorophyll_a_in_sea_water',
'Chlorophyll a concentration, Garver-Siegel-Maritorena Model':
'mass_concentration_of_chlorophyll_a_in_sea_water', 'Diffuse attenuation
coefficient':
'volume_attenuation_coefficient_of_downwelling_radiative_flux_in_sea_water',
'Instantaneous Photosynthetically Available Radiation':
'instantaneous_downwelling_photosynthetic_photon_radiance_in_sea_water', 'Particle
backscatter at 443 nm':
'volume_backscattering_coefficient_of_radiative_flux_in_sea_water_due_to_suspended_particles',
'Photosynthetically Available Radiation':
'downwelling_photosynthetic_photon_radiance_in_sea_water', 'Remote sensing
reflectance':
'surface_ratio_of_upwelling_radiance_emerging_from_sea_water_to_downwelling_radiative_flux_in_air',
'Sea Surface Salinity': 'sea_surface_salinity', 'Sea Surface Temperature':
'sea_surface_temperature'}

nansat.mappers.mapper_ocean_productivity module

class nansat.mappers.mapper_ocean_productivity.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper for Ocean Productivity website http://www.science.oregonstate.edu/ocean.productivity/

param2wkv = {'bbp': 'particle_backscatter_at_443_nm', 'chl':
'mass_concentration_of_chlorophyll_a_in_sea_water', 'ipar':
'instantaneous_downwelling_photosynthetic_photon_radiance_in_sea_water', 'par':
'downwelling_photosynthetic_photon_radiance_in_sea_water', 'sst':
'sea_surface_temperature'}

bandNames =
{'instantaneous_downwelling_photosynthetic_photon_radiance_in_sea_water': 'ipar',
'mass_concentration_of_chlorophyll_a_in_sea_water': 'algal_1',
'particle_backscatter_at_443_nm': 'bbp_443', 'sea_surface_temperature': 'SST'}

nansat.mappers.mapper_opendap_arome module

class nansat.mappers.mapper_opendap_arome.**Mapper**(*filename*, *gdal_dataset*, *gdal_metadata*, *date=None*, *ds=None*, *bands=None*, *cachedir=None*, **args*, ***kwargs*)

Bases: *nansat.mappers.opendap.Opendap*, *nansat.mappers.mapper_arome.Mapper*

baseURLs = ['http://thredds.met.no/thredds/catalog/arome25/catalog.html',
'https://thredds.met.no/thredds/dodsC/aromearcticarchive',
'http://thredds.met.no/thredds/dodsC/aromearcticarchive',
'https://thredds.met.no/thredds/dodsC/meps25epsarchive',
'http://thredds.met.no/thredds/dodsC/meps25epsarchive']

timeVarName = 'time'

xName = 'x'

yName = 'y'

timeCalendarStart = '1970-01-01'

static get_date(*filename*)

Extract date and time parameters from filename and return it as a formatted string

Parameters **filename** (*str*) – nn

Returns str, YYYY-mm-ddThh

Return type MMZ

Examples

```
>>> Mapper.get_date('/path/to/arome_arctic_full_2_5km_20171030T21Z.nc')
'2017-10-30T21:00Z'
```

convert_dstime_datetimes(*ds_time*)

Convert time variable to np.datetime64

nansat.mappers.mapper_opendap_globcurrent module

nansat.mappers.mapper_opendap_globcurrent_thredds module

nansat.mappers.mapper_opendap_mywave module

class nansat.mappers.mapper_opendap_mywave.**Mapper**(*filename*, *gdal_dataset*, *gdal_metadata*, *date=None*, *ds=None*, *bands=None*, *cachedir=None*, **args*, ***kwargs*)

Bases: *nansat.mappers.opendap.Opendap*

baseURLs = ['http://thredds.met.no/thredds/dodsC/fou-hi/mywavewam4archive',
'https://thredds.met.no/thredds/dodsC/sea/mywavewam4/mywavewam4_be']

timeVarName = 'time'

xName = 'rlon'

yName = 'rlat'

timeCalendarStart = '1970-01-01'

static get_date(*filename*)

Extract date and time parameters from filename and return it as a formatted (isoformat) string

Parameters **filename** (*str*) –

Returns str, YYYY-mm-ddThh

Return type MM:00Z

Examples

```
>>> Mapper.get_date('/path/to/MyWave_wam4_WAVE_20171029T18Z.nc')
'2017-10-29T18:00:00Z'
```

convert_dstime_datetimes(*ds_time*)
Convert time variable to np.datetime64

nansat.mappers.mapper_opendap_occci module

nansat.mappers.mapper_opendap_osisaf module

nansat.mappers.mapper_opendap_ostia module

nansat.mappers.mapper_opendap_sentinel1 module

nansat.mappers.mapper_opendap_sentinel2 module

class nansat.mappers.mapper_opendap_sentinel2.**Mapper**(*filename*, *gdal_dataset*, *gdal_metadata*, *date=None*, *ds=None*, *bands=None*, *cachedir=None*, **args*, ***kwargs*)

Bases: *nansat.mappers.opendap.Opendap*

baseURLs = ['http://nbstds.met.no/thredds/dodsC/NBS/S2A', 'http://nbstds.met.no/thredds/dodsC/NBS/S2B']

timeVarName = 'time'

xName = 'x'

yName = 'y'

timeCalendarStart = '1981-01-01'

GCP_STEP = 100

static get_date(*filename*)
Extract date and time parameters from filename and return it as a formatted (isoformat) string

Parameters **filename** (*str*) – nn

Returns **str, YYYY-mm-ddThh**

Return type MMZ

convert_dstime_datetimes(*ds_time*)
Convert time variable to np.datetime64

nansat.mappers.mapper_opendap_siwtacsst module

nansat.mappers.mapper_opendap_sstcci module

nansat.mappers.mapper_pathfinder52 module

class nansat.mappers.mapper_pathfinder52.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *minQual=4*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Mapper PATHFINDER (local files)

TODO: * remote files

nansat.mappers.mapper_quikscat module

class nansat.mappers.mapper_quikscat.**Mapper**(*filename*, *gdal_dataset*, *metadata*, *quartile=0*, **args*, ***kwargs*)

Bases: *nansat.mappers.scatterometers.Mapper*

Nansat mapper for QuikScat

nansat.mappers.mapper_radarsat2 module

class nansat.mappers.mapper_radarsat2.**Mapper**(*inputFileName*, *gdalDataset*, *gdalMetadata*, *xmlonly=False*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Create VRT with mapping of WKV for Radarsat2

init_from_xml(*productXml*, *filename*)
Fast init from metada in XML only

nansat.mappers.mapper_sentinel1_l1 module

class nansat.mappers.mapper_sentinel1_l1.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *fast=False*, *fixgcp=True*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Create VRT with mapping of Sentinel-1 (A and B) stripmap mode (S1A_SM)

Parameters

- **filename** (*str*) – name of input Sentinel-1 L1 file
- **gdalDataset** (*None*) –
- **gdalMetadata** (*None*) –
- **fast** (*bool*) – Flag that triggers faster reading of metadata from Sentinel-1 file. If True, no bands are added to the dataset and georeference is not corrected. If False, all bands are added and GCPs are corrected if necessary (see Mapper.correct_geolocation_data for details).

Note: Creates self.dataset and populates it with S1 bands (when fast=False).

read_calibration(*xml*, *vectorListName*, *variable_names*, *pol*)
Read calibration data from calibration or noise XML files :param xml: String with XML from calibration or noise files :type xml: str :param vectorListName: tag of the element that contains lists with LUT values :type vectorListName: str :param variable_names: names of LUT variable to read :type variable_names: list of str :param pol: HH, HV, etc :type pol: str

Returns **data** – Calibration or noise data. Keys: The same as variable_names + 'pixel', 'line'

Return type dict

read_annotation(*annotation_files*)
Read lon, lat, etc from annotation XML

Parameters **annotation_files** (*list*) – strings with names of annotation files

Returns

data –

geolocation data from the XML as 2D np.arrays. Keys: pixel, line, longitude, latitude, height, incidenceAngle, elevationAngle: 2D arrays shape : tuple (shape of geolocation data arrays) x_size, y_size : int pol : list

Return type dict

correct_geolocation_data()
Correct lon/lat values in geolocation data for points high above ground (incorrect)

Each GCP in Sentinel-1 L1 image (both in the GeoTIF files and Annotation LUT) have five coordinates: X, Y, Z (height), Pixel and Line. On some scenes that cover Greenland (and probably other lands) some GCPs have height above zero even over ocean. This is incorrect, because the radar signal comes actually from the surface and not from a point above the ground as stipulated in such GCPs. Correction of GCPs in this function is equivalelnt to reverse DEM correction of SAR data.

Notes

Updates 'pixel' and 'height' in self.annotation_data

static create_gcps(*x*, *y*, *z*, *p*, *l*)
Create GCPs from geolocation data

Parameters

- **x** –
- **y** –
- **z** –
- **p** –
- **l** –
- **X** (*N-D arrays with value of*) –
- **Y** –
- **Z** –
- **coordinates.** (*Pixel and Line*) –
- **lon** (*X and Y are typically*) –
- **lat** –

- **height.** (*Z -*) –

Returns gcps

Return type list with GDAL GCPs

read_manifest_data(*input_file*)
Read information (time_coverage_start, etc) manifest XML

Parameters **input_file** (*str*) – name of manifest file

Returns

data –

manifest data. Keys: time_coverage_start time_coverage_end platform_familyName platform_number

Return type dict

vrts_from_arrays(*data*, *variable_names*, *pol=''*, *resize=True*, *resample_alg=2*)
Convert input dict with arrays into dict with VRTs

Parameters

- **data** (*dict*) – 2D arrays with data from LUT
- **variable_names** (*list of str*) – variable names that should be converted to VRTs
- **pol** (*str*) – HH, HV, etc
- **resize** (*bool*) – Shall VRT be zoomed to full size?
- **resample_alg** (*int*) – Index of resampling algorithm. See VRT.get_resized_vrt()

Returns vrts

Return type dict with (resized) VRTs

nansat.mappers.mapper_sentinel1_l2 module

class nansat.mappers.mapper_sentinel1_l2.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *product_type='RVL'*, *GCP_COUNT=10*, ***kwargs*)
Bases: *nansat.vrt.VRT*

Create VRT with mapping of Sentinel-1A stripmap mode (S1A_SM)

nansat.mappers.mapper_topography module

class nansat.mappers.mapper_topography.**Mapper**(*filename*, *gdal_dataset*, **args*, ***kwargs*)
Bases: *nansat.vrt.VRT*

Mapping for the GTOPO30 (https://lta.cr.usgs.gov/GTOPO30) and the GMTED2010 (https://lta.cr.usgs.gov/GMTED2010) global elevation models.

Parameters

- **filename** (*string*) – The vrt filename, e.g., `gtopo30.vrt`
- **gdal_dataset** (*osgeo.gdal.Dataset*) – The GDAL dataset returned by `gdal.Open(filename)`

Example

You can create your own gtopo30.vrt file with gdal, e.g.:

```
gdalbuildvrt <dem>.vrt [E,W]*.DEM
```

Note: Either the name of a GTOPO30 DEM file or GMTED2010 tif file, or <path>/<dem>.vrt. The latter is an aggregation of the DEM-files available from the given DEM. The GTOPO30 vrt does not contain the Antarctic, because this is in polarstereographic projection.

Remember to update this mapper by adding allowed filenames to the list of accepted filenames (accepted_names) if you create or apply new DEM datasets.

nansat.mappers.mapper_viirs_l1 module

class nansat.mappers.mapper_viirs_l1.**Mapper**(*filename*, *gdalDataset*, *gdalMetadata*, *GCP_COUNT0=5*, *GCP_COUNT1=20*, *pixelStep=1*, *lineStep=1*, ***kwargs*)

Bases: *nansat.vrt.VRT*

VRT with mapping of WKV for VIIRS Level 1B

nansat.mappers.obpg module

class nansat.mappers.obpg.**OBPGL2BaseClass**(*x_size=1*, *y_size=1*, *metadata=None*, *nomem=False*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Base Class for Mappers for SeaWIFS/MODIS/MERIS/VIIRS L2 data from OBPG

titles = ['HMODISA Level-2 Data', 'MODISA Level-2 Data', 'HMODIST Level-2 Data', 'MERIS Level-2 Data', 'GOCI Level-2 Data', 'VIIRSN Level-2 Data', 'SeaWiFS Level-2 Data']

nansat.mappers.opendap module

class nansat.mappers.opendap.**Opendap**(*x_size=1*, *y_size=1*, *metadata=None*, *nomem=False*, ***kwargs*)

Bases: *nansat.vrt.VRT*

Methods for all OpenDAP mappers

P2S = {'D': 86400, 'H': 3600, 'M': 2592000, 'Y': 31536000}

test_mapper(*filename*)

Tests if filename fits mapper. May raise WrongMapperError

Parameters **filename** (*str*) – absolute url of input file

Raises *WrongMapperError* – if input url does not match with list of: urls for a mapper

get_dataset(*ds*)

Open Dataset

Parameters **ds** (*str or netCDF4.Dataset*) –

get_geospatial_variable_names()
Get names of variables with both spatial dimensions

get_dataset_time()
Load data from time variable

static get_layer_datetime(*date*, *datetimes*)
Get datetime of the matching layer and layer number

get_metaitem(*url*, *var_name*, *var_dimensions*)
Set metadata for creating band VRT

Parameters

- **url** (*str,*) – absolute url of an input file
- **var_name** (*str,*) – name of a variable/band from netCDF file
- **var_dimensions** – iterable array with dimensions of the variable

create_vrt(*filename*, *gdalDataset*, *gdalMetadata*, *date*, *ds*, *bands*, *cachedir*)
Create VRT

Parameters

- **filename** (*str,*) – absolute url of an input file
- **date** (*str,*) – date in format YYYY-MM-DD
- **ds** (*netDCF.Dataset*) –
- **bands** (*list*) – list of src bands
- **cachedir** (*str*) –

create_metadict(*filename*, *var_names*, *time_id*)
Create list which contains a dictionary with metadata for each single band

Parameters

- **filename** (*str,*) – full path to the file
- **var_names** (*iterable,*) – iterable object (list) with required band names (str)
- **time_id** (*int,*) – index of required slice in time dimension

Returns **meta_dict** – list which contains a dictionary with metadata for each <var_name>

Return type list

get_time_coverage_resolution()
Try to fecth time_coverage_resolution and convert to seconds

get_shape()
Get srcRasterXSize and srcRasterYSize from OpenDAP

get_geotransform()
Get first two values of X,Y variables and create geoTranform

nansat.mappers.scatterometers module

class nansat.mappers.scatterometers.**Mapper**(*filename*, *gdal_dataset*, *metadata*, *quartile=0*, **args*, ***kwargs*)

Bases: *nansat.mappers.mapper_netcdf_cf.Mapper*

Nansat mapper for scatterometers

set_gcps(*lon*, *lat*, *gdal_dataset*)
Set gcps

static shift_longitudes(*lon*)
Apply correction of longitudes (they are defined on 0:360 degrees but also contain egative values)

TODO: consider making this core to nansat - different ways of defining longitudes (-180:180 og 0:360 degrees) often cause problems…

nansat.mappers.sentinel1 module

class nansat.mappers.sentinel1.**Sentinel1**(*filename*, *flip_gcp_line=False*)
Bases: *nansat.vrt.VRT*

Mapper class to access Sentinel-1 data with netCDF4 to work for both opendap streams and local files.

timeVarName = 'time'

input_filename = ''

set_gcmd_dif_keywords()

get_gcps(*flip_gcp_line=False*)
Get Ground Control Points for the dataset.

Note that OPeNDAP streams and netCDF files are read differently by gdal. The OPeNDAP streams are read by specifying the get parameters to the OPeNDAP url. The get parameters specify the reference dimensions, e.g., x and y. Since these are specified, the raster data is correctly referenced to the GCPs. However, when gdal reads a raster band from netCDF, it reads it "blindly". This is risky, since the definition of origo may be different in gdal vs the original data (e.g., first line starts in upper left corner or in lower left corner). For Sentinel-1, the raster data is flipped in relation to the GCPs, so we need to flip the GCP line vector as well.

add_incidence_angle_band()

get_full_size_GCPs()

add_look_direction_band()

Module contents

nansat.tests package

Subpackages

Submodules

nansat.tests.nansat_test_base module

class nansat.tests.nansat_test_base.**NansatTestBase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

nansat.tests.nansat_test_data module

nansat.tests.test_domain module

class nansat.tests.test_domain.**DomainTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_dont_init_from_invalid_combination()

test_init_from_GDALDataset()

test_init_from_GDALDataset_and_srs()

test_dont_init_if_gdal_AutoCreateWarpedVRT_fails(*mock_gdal*)

test_init_from_srs_and_ext_te(*mock__get_geotransform*, *mock__create_extent_dict*)

test_init_from_srs_and_ext_lle(*mock__get_geotransform*, *mock__convert_extentDic*, *mock__create_extent_dict*)

test_init_lonlat()

test_init_from_lonlat()

test_init_from_lonlat_no_gcps()

test_repr(*mock_get_corners*)

test_write_kml()

test_get_geolocation_grids_from_GDAL_transformer(*mock_transform_points*)

test_get_geolocation_grids_from_geolocationArray()

test_convert_extentDic()

test_add_to_dict()

test_validate_ts_tr()

test_validate_te_lle()

test_check_size()

test_gen_regexp()

test_create_extent_dict()

test_get_border()

test_compound_row_col_vectors()

test_get_row_col_vector()

test_get_border_wkt()

test_get_border_geometry()

test_border_geojson()

test_overlaps_intersects_and_contains()

test_contains()

test_get_border_postgis()

test_get_corners()

test_get_min_max_lon_lat()

test_get_pixelsize_meters()

test_get_geotransform()

test_transform_tr()

test_transform_ts2()

test_transform_points()

test_transform_points_inverse()

test_transform_points_dstsrs()

test_azimuth_y(*mock_get_geolocation_grids*)

test_shape()

test_reproject_gcps()

test_reproject_gcps_auto()

test_intersects(*get_border_geometry*)

test_overlaps(*get_border_geometry*)

test_get_border_dateline()

nansat.tests.test_exporter module

class nansat.tests.test_exporter.**ExporterTest**(*methodName='runTest'*)

Bases: *nansat.tests.nansat_test_base.NansatTestBase*

test_geolocation_of_exportedNC_vs_original()

Lon/lat in original and exported file should coincide

test_special_characters_in_exported_metadata()

test_time_coverage_metadata_of_exported_equals_original()

test_export_netcdf()

Test export and following import of data with bands containing np.nan values

test_export_gcps_to_netcdf()

Should export file with GCPs and write correct bands

test_export_gcps_complex_to_netcdf()

Should export file with GCPs and write correct complex bands

test_export_gtiff()

test_export_band()

test_export_band_by_name()

test_reproject_and_export_band()

test_export_selected_bands()

test_export_option()

test_export2thredds_arctic_long_lat()

test_dont_export2thredds_gcps()

test_export2thredds_longlat_list()

test_export2thredds_longlat_dict()

test_export_netcdf_complex_remove_meta()

test_export_netcdf_arctic()

test_export_netcdf_arctic_hardcopy()

test_export_add_geoloc(*mock_add_geolocation*)

test_export2thredds_rmmetadata()

class nansat.tests.test_exporter.**TestExporter__export2thredds**(*methodName='runTest'*)

Bases: *nansat.tests.nansat_test_base.NansatTestBase*

setUp()

Hook method for setting up the test fixture before exercising it.

tearDown()

Hook method for deconstructing the test fixture after testing it.

test_example1()

test_example2()

test_example3()

nansat.tests.test_figure module

class nansat.tests.test_figure.**FigureTest**(*methodName='runTest'*)

Bases: *nansat.tests.nansat_test_base.NansatTestBase*

test_init_array()

test_get_auto_ticks_number()

test_get_auto_ticks_vector()

test_add_latlon_grids_auto()

Should create figure with lon/lat gridlines spaced automatically

test_add_latlon_grids_number()

Should create figure with lon/lat gridlines given manually

test_add_latlon_grids_list()

Should create figure with lon/lat gridlines given manually

test_get_tick_index_from_grid()
Should return indeces of pixel closest to ticks

test_apply_logarithm(*mock1*)

test_make_transparent_color(*mock1*)

nansat.tests.test_geolocation module

class nansat.tests.test_geolocation.GeolocationTest(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_init()

test_from_dataset()

test_from_filenames()

nansat.tests.test_nansat module

class nansat.tests.test_nansat.NansatTest(*methodName='runTest'*)
Bases: *nansat.tests.nansat_test_base.NansatTestBase*

test_open_gcps()

test_that_only_mappers_with_mapper_in_the_module_name_are_imported()

test_get_time_coverage_start_end()

test_from_domain_array()

test_from_domain_nansat()

test_add_band()

test_add_band_twice()

test_add_bands()

test_add_bands_no_parameter()

test_add_subvrts_only_to_one_nansat()

test_bands()

test_has_band_if_name_matches()

test_has_band_if_standard_name_matches()

test_write_fig_tif()

test_resize_by_pixelsize()

test_resize_by_factor()

test_resize_by_width()

test_resize_by_height()

test_resize_resize()

test_resize_complex_alg_average()

test_resize_complex_alg0()

test_resize_complex_alg1()

test_resize_complex_alg2()

test_resize_complex_alg3()

test_resize_complex_alg4()

test_get_GDALRasterBand()

test_get_GDALRasterBand_if_band_id_is_given()

test_list_bands_true()

test_list_bands_false()

test_reproject_domain()

test_reproject_domain_if_dst_domain_is_given()

test_reproject_domain_if_resample_alg_is_given()

test_reproject_domain_if_source_and_destination_domain_span_entire_lons(*mock_Nansat*)

test_reproject_domain_if_tps_is_given()

test_reproject_of_complex()
Should return np.nan in areas out of swath

test_add_band_and_reproject()
Should add band and swath mask and return np.nan in areas out of swath

test_reproject_no_addmask()
Should not add swath mask and return 0 in areas out of swath

test_reproject_stere()

test_reproject_gcps()

test_reproject_gcps_on_repro_gcps()

test_reproject_gcps_resize()

test_undo()

test_write_figure()

test_write_figure_band()

test_write_figure_clim()

test_write_figure_legend()

test_write_figure_logo()

test_write_geotiffimage()

test_write_geotiffimage_if_band_id_is_given()

test_get_metadata()

test_get_metadata_key()

test_get_metadata_wrong_key()

test_get_metadata_band_id()

test_set_metadata()

test_set_metadata_band_id()

test_get_band_number()

test_get_transect()

test_get_transect_outside()

test_get_transect_wrong_points()

test_get_transect_wrong_band()

test_get_transect_pixlin()

test_get_transect_data()

test_digitize_points(*mock_PointBrowser*)
shall create PointBrowser and call PointBrowser.get_points()

test_crop()

test_crop_gcpproj()

test_crop_complex()

test_crop_no_gcps_arctic()

test_crop_lonlat()

test_crop_outside()

test_watermask()
if watermask data exists: should fetch array with watermask else: should raise an error

test_watermask_fail_if_mod44path_is_wrong()
Nansat.watermask should raise an IOError

test_watermask_fail_if_mod44path_not_exist()
Nansat.watermask should raise an IOError

test_init_no_arguments()
No arguments should raise ValueError

test_get_item_basic_expressions()
Testing get_item with some basic expressions

test_get_item_inf_expressions()
inf should be replaced with nan

test_repr_basic()
repr should include some basic elements

test_getitem(*mock_Nansat*)

test_crop_interactive(*mock_digitize_points*)

test_extend()

test_open_no_mapper()

test_get_metadata_unescape(*vrt*)

test_reproject_pure_geolocation()

nansat.tests.test_node module

class nansat.tests.test_node.**NodeTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_creation()

test_getAttributeList()

test_insert()

test_create()

test_delete_attribute()

test_add_node()

test_add_nodes()

test_xml()

test_replace_node()

test_search_node()

test_str()

nansat.tests.test_nsr module

class nansat.tests.test_nsr.**NSRTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_init_empty()

test_init_from_none()

test_init_from_0()

test_init_from_EPSG()

test_init_from_proj4()

test_init_from_proj4_unicode()

test_init_from_wkt()

test_init_from_NSR()

test_dont_init_from_invalid()

nansat.tests.test_pixelfunctions module

class nansat.tests.test_pixelfunctions.**TestPixelFunctions**(*methodName='runTest'*)
Bases: unittest.case.TestCase

test_import_pixel_functions()

nansat.tests.test_pointbrowser module

class nansat.tests.test_pointbrowser.**PointBrowserTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_init()
Create Pointbrowser

test_onclick()
Mimic click

test_onclick_none()
Mimic click outside figure

test_onclick_key_z()
Mimic click with 'z' pressed

test_onclick_key()
Mimic click with 'anykey' pressed

test_convert_coordinates()
Mimic click with 'anykey' pressed

test_get_points(*plt_mock*)

nansat.tests.test_tools module

class nansat.tests.test_tools.**ToolsTest**(*methodName='runTest'*)
Bases: unittest.case.TestCase

setUp()
Hook method for setting up the test fixture before exercising it.

test_distance2coast_source_not_exists_envvar(*mock_getenv*)

test_distance2coast_source_not_exists_attribute()

test_distance2coast_integration(*Nansat*, *os*)

test_warning()

test_get_domain_map()

test_get_domain_map_no_cartopy()

test_save_domain_map()

nansat.tests.test_vrt module

class nansat.tests.test_vrt.**VRTTest**(*methodName='runTest'*)

Bases: *nansat.tests.nansat_test_base.NansatTestBase*

nsr_wkt = 'GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS 84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]], PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.0174532925199433, AUTHORITY["EPSG","9122"]],AUTHORITY["EPSG","4326"]]'

test_init(*mock_make_filename*)

test_del(*mock_make_filename*)

test_from_gdal_dataset(*_init_from_gdal_dataset*)

test_init_from_gdal_dataset(*_add_geolocation*)

test_from_dataset_params()

test_from_array()

test_from_lonlat()

test_from_lonlat_no_gcps()

test_copy_empty_vrt()

test_copy_vrt_with_band()

test_copy_vrt_pixel_func()

test_copy_geolocation()

test_export()

test_create_band()

test_make_source_bands_xml()

test_set_add_band_options()

test_remove_geotransform()

test_set_geotransform_for_resize()

test_set_gcps_geolocation_geotransform_with_geolocation()

test_set_gcps_geolocation_geotransform_with_gcps()

test_set_gcps_geolocation_geotransform_with_geotransform()

test_update_warped_vrt_xml()

test_set_fake_gcps_empty()

test_set_fake_gcps()

test_get_dst_band_data_type()

test_create_band_name_no_wkv()

test_create_band_name_wkv()

test_create_band_name_existing_name()

test_create_band_name_wkv_and_name()

test_leave_few_bands()

test_find_complex_band()

test_split_complex_bands()

test_create_geolocation_bands()

test_fix_band_metadata()

test_fix_global_metadata()

test_hardcopy_bands()

test_get_projection_dataset(*dataset*)

test_get_projection_gcps(*dataset*)

test_get_projection_geolocation(*dataset*)

test_get_projection_raises_NansatProjectionError(*dataset*)

test_repr()

test_add_swath_mask_band(*create_band*)

test_remove_strings_in_metadata_keys()

test_super_vrt_of_geolocation_bands()

test_get_shifted_vrt()

test_get_super_vrt()

test_get_super_vrt_geolocation()

test_get_super_vrt_and_copy()

test_get_sub_vrt0()

test_get_sub_vrt3()

test_get_sub_vrt_steps_0()

test_transform_points()

test_make_filename()

test_transform_coordinates_list()

test_transform_coordinates_1d_array()

test_transform_coordinates_2d_array()

test_reproject_gcps()

Module contents

Submodules

nansat.domain module

class nansat.domain.**Domain**(*srs=None*, *ext=None*, *ds=None*, ***kwargs*)

Bases: object

Container for geographical reference of a raster

A Domain object describes all attributes of geographical reference of a raster:

- width and height (number of pixels)
- pixel size (e.g. in decimal degrees or in meters)
- relation between pixel/line coordinates and geographical coordinates (e.g. a linear relation)
- type of data projection (e.g. geographical or stereographic)

Parameters

- **srs** (*PROJ4 or EPSG or WKT or NSR or osr.SpatialReference()*) – Input parameter for nansat.NSR()
- **ext** (*string*) – some gdalwarp options + additional options [http://www.gdal.org/gdalwarp.html] Specifies extent, resolution / size Available options: ((‘-te’ or ‘-lle’) and (‘-tr’ or ‘-ts’)) (e.g. ‘-lle -10 30 55 60 -ts 1000 1000’ or ‘-te 100 2000 300 10000 -tr 300 200’) -tr resolutionx resolutiony -ts sizex sizey -te xmin ymin xmax ymax -lle lonmin latmin lonmax latmax
- **ds** (*GDAL dataset*) –

Examples

```
>>> d = Domain(srs, ext) #size, extent and spatial reference is given by strings
>>> d = Domain(ds=GDALDataset) #size, extent copied from input GDAL dataset
>>> d = Domain(srs, ds=GDALDataset) # spatial reference is given by srs,
    but size and extent is determined from input GDAL dataset
```

Notes

The core of Domain is a GDAL Dataset. It has no bands, but only georeference information: rasterXsize, rasterYsize, GeoTransform and Projection or GCPs, etc. which fully describe dimentions and spatial reference of the grid.

There are three ways to store geo-reference in a GDAL dataset:

- Using GeoTransfrom to define linear relationship between raster pixel/line and geographical X/Y coordinates
- Using GCPs (set of Ground Control Points) to define non-linear relationship between pixel/line and X/Y
- Using Geolocation Array - full grids of X/Y coordinates for each pixel of a raster

The relation between X/Y coordinates of the raster and latitude/longitude coordinates is defined by projection type and projection parameters. These pieces of information are therefore stored in Domain:

- Type and parameters of projection +
  - GeoTransform, or
  - GCPs, or
  - GeolocationArrays

Domain has methods for basic operations with georeference information:

- creating georeference from input options;
- fetching corner, border or full grids of X/Y coordinates;
- making map of the georeferenced grid in a PNG or KML file;
- and some more…

The main attribute of Domain is a VRT object self.vrt. Nansat inherits from Domain and adds bands to self.vrt

:raises NansatProjectionError : occurs when Projection() is empty: despite it is required for creating extentDic.
:raises OptionError : occures when the arguments are not proper.:

See also:

Nansat.reproject()

http://www.gdal.org/gdalwarp.html

http://trac.osgeo.org/proj/

http://spatialreference.org/

http://www.gdal.org/osr_tutorial.html

```
OUTPUT_SEPARATOR = '----------------------------------------\n'
```

```
KML_BASE = '<?xml version="1.0" encoding="UTF-8"?>\n <kml
xmlns="http://www.opengis.net/kml/2.2"\n
xmlns:gx="http://www.google.com/kml/ext/2.2"\n
xmlns:kml="http://www.opengis.net/kml/2.2"\n
xmlns:atom="http://www.w3.org/2005/Atom">\n {content}\n </kml>'
```

```
logger = None
```

```
name = None
```

```
vrt = None
```

classmethod from_lonlat(*lon*, *lat*, *add_gcps=True*)

Create Domain object from input longitudes, latitudes arrays

Parameters

- **lon** (*numpy.ndarray*) – longitudes
- **lat** (*numpy.ndarray*) – latitudes
- **add_gcps** (*bool*) – Add GCPs from lon/lat arrays.

Returns d

Return type *Domain*

Examples

```
>>> lon, lat = np.meshgrid(range(10), range(10))
>>> d1 = Domain.from_lonlat(lon, lat)
>>> d2 = Domain.from_lonlat(lon, lat, add_gcps=False) # add only geolocation␣
↪arrays
```

write_kml(*xmlFileName=None*, *kmlFileName=None*)

Write KML file with domains

Convert XML-file with domains into KML-file for GoogleEarth or write KML-file with the current Domain

Parameters

- **xmlFileName** (*string, optional*) – Name of the XML-file to convert. If only this value is given - kmlFileName=xmlFileName+'.kml'

- **kmlFileName** (*string, optional*) – Name of the KML-file to generate from the current Domain

write_kml_image(*kmlFileName*, *kmlFigureName=None*)
Create KML file for already projected image

Write Domain Image into KML-file for GoogleEarth

Parameters

- **kmlFileName** (*str*) – Name of the KML-file to generate from the current Domain
- **kmlFigureName** (*str*) – Name of the projected image stored in .png format

Examples

```
>>> n.undo(100) # cancel previous reprojection
>>> lons, lats = n.get_corners() # Get corners of the image and the pixel␣
↪resolution
>>> srsString = '+proj=latlong +datum=WGS84 +ellps=WGS84 +no_defs'
>>> extentString = '-lle %f %f %f %f -ts 3000 3000'
    % (min(lons), min(lats), max(lons), max(lats))
>>> d = Domain(srs=srsString, ext=extentString) # Create Domain with
    stereographic projection, corner coordinates and resolution 1000m
>>> n.reproject(d)
>>> n.write_figure(filename=figureName, bands=[3], clim=[0,0.15],
                   cmapName='gray', transparency=0)
>>> n.write_kml_image(kmlFileName=oPath + filename + '.kml',
                   kmlFigureName=figureName) # 6.
```

get_geolocation_grids(*stepSize=1*, *dst_srs=None*)
Get longitude and latitude grids representing the full data grid

If GEOLOCATION is not present in the self.vrt.dataset then grids are generated by converting pixel/line of each pixel into lat/lon If GEOLOCATION is present in the self.vrt.dataset then grids are read from the geolocation bands.

Parameters **stepSize** (*int*) – Reduction factor if output is desired on a reduced grid size

Returns

- **longitude** (*numpy array*) – grid with longitudes
- **latitude** (*numpy array*) – grid with latitudes

get_border(*n_points=10*, *fix_lon=True*, ***kwargs*)
Generate two vectors with values of lat/lon for the border of domain

Parameters

- **n_points** (*int, optional*) – Number of points on each border
- **fix_lon** (*bool*) – Convert longitudes to positive numbers when Domain crosses dateline?

Returns **lonVec, latVec** – vectors with lon/lat values for each point at the border

Return type lists

get_border_wkt(**args*, ***kwargs*)
Creates string with WKT representation of the border polygon

Returns **WKTPolygon** – string with WKT representation of the border polygon

Return type string

`get_border_geometry`(**args*, ***kwargs*)
Get OGR Geometry of the border Polygon

Returns **OGR Geometry**

Return type Polygon

`get_border_geojson`(**args*, ***kwargs*)
Create border of the Polygon in GeoJson format

Returns **the Polygon border in GeoJson format**

Return type str

`overlaps`(*anotherDomain*)
Checks if this Domain overlaps another Domain

Returns **overlaps** – True if Domains overlaps, False otherwise

Return type bool

`intersects`(*anotherDomain*)
Checks if this Domain intersects another Domain

Returns **intersects** – True if Domains intersects, False otherwise

Return type bool

`contains`(*anotherDomain*)
Checks if this Domain fully covers another Domain

Returns **contains** – True if this Domain fully covers another Domain, False otherwise

Return type bool

`get_border_postgis`(***kwargs*)
Get PostGIS formatted string of the border Polygon

Returns **'PolygonFromText(PolygonWKT)'**

Return type str

`get_corners`()
Get coordinates of corners of the Domain

Returns **lonVec, latVec** – vectors with lon/lat values for each corner

Return type lists

`get_min_max_lon_lat`()
Get minimum and maximum of longitude and latitude geolocation grids

Returns **min_lon, max_lon, min_lat, max_lat,** – min/max lon/lat values for the Domain

Return type float

`get_pixelsize_meters`()
Returns the pixelsize (deltaX, deltaY) of the domain

For projected domains, the exact result which is constant over the domain is returned. For geographic (lon-lat) projections, or domains with no geotransform, the haversine formula is used to calculate the pixel size in the center of the domain.

Returns **delta_x, delta_y** – pixel size in X and Y directions given in meters

Return type float

transform_points(*colVector*, *rowVector*, *DstToSrc=0*, *dst_srs=None*)
Transform given lists of X,Y coordinates into lon/lat or inverse

Parameters

- **colVector** (*lists*) – X and Y coordinates in pixel/line or lon/lat coordinate system
- **DstToSrc** (*0 or 1*) –
  - 0 - forward transform (pix/line => lon/lat)
  - 1 - inverse transformation
- **dst_srs** (NSR) – destination spatial reference

Returns **X, Y** – X and Y coordinates in lon/lat or pixel/line coordinate system

Return type lists

azimuth_y(*reductionFactor=1*)
Calculate the angle of each pixel position vector with respect to the Y-axis (azimuth).

In general, azimuth is the angle from a reference vector (e.g., the direction to North) to the chosen position vector. The azimuth increases clockwise from direction to North. http://en.wikipedia.org/wiki/Azimuth

Parameters **reductionFactor** (*integer*) – factor by which the size of the output array is reduced

Returns **azimuth** – Values of azimuth in degrees in range 0 - 360

Return type numpy array

shape()
Return Numpy-like shape of Domain object (ySize, xSize)

Returns **shape** – Numpy-like shape of Domain object (ySize, xSize)

Return type tuple of two INT

reproject_gcps(*srs_string=''*)
Reproject all GCPs to a new spatial reference system

Necessary before warping an image if the given GCPs are in a coordinate system which has a singularity in (or near) the destination area (e.g. poles for lonlat GCPs)

Parameters **srs_string** (*string*) – SRS given as Proj4 string. If empty '+proj=stere' is used

Notes

Reprojects all GCPs to new SRS and updates GCPProjection

nansat.exceptions module

exception nansat.exceptions.**NansatProjectionError**
Bases: Exception

Cannot get the projection

exception nansat.exceptions.**NansatGDALError**
Bases: Exception

Error from GDAL

exception nansat.exceptions.**NansatReadError**
Bases: Exception

Exception if a file cannot be read with Nansat

exception nansat.exceptions.**NansatGeolocationError**
Bases: Exception

Exception if geolocation is wrong (e.g., all lat/lon values are 0)

exception nansat.exceptions.**NansatMissingProjectionError**
Bases: Exception

Exception raised if no (sub-) dataset has projection

exception nansat.exceptions.**WrongMapperError**
Bases: Exception

Error for handling data that does not fit a given mapper

nansat.exporter module

class nansat.exporter.**Exporter**
Bases: object

Abstract class for export functions

DEFAULT_INSTITUTE = 'NERSC'

DEFAULT_SOURCE = 'satellite remote sensing'

UNWANTED_METADATA = ['dataType', 'SourceFilename', 'SourceBand', '_Unsigned', 'FillValue', 'time', '_FillValue', 'type', 'scale', 'offset']

export(*filename=''*, *bands=None*, *rm_metadata=None*, *add_geolocation=True*, *driver='netCDF'*, *options=None*, *hardcopy=False*)
Export Nansat object into netCDF or GTiff file

Parameters

- **filename** (*str*) – output file name
- **bands** (*list (default=None)*) – Specify band numbers to export. If None, all bands are exported.
- **rm_metadata** (*list*) – metadata names for removal before export. e.g. ['name', 'colormap', 'source', 'sourceBands']
- **add_geolocation** (*bool*) – add geolocation array datasets to exported file?
- **driver** (*str*) – Name of GDAL driver (format)
- **options** (*str or list*) – GDAL export options in format of: 'OPT=VAL', or ['OPT1=VAL1', 'OP2='VAL2'] See also http://www.gdal.org/frmt_netcdf.html
- **hardcopy** (*bool*) – Evaluate all bands just before export?

Returns filename

Return type netCDF or GTiff

Notes

If number of bands is more than one, serial numbers are added at the end of each band name. It is possible to fix it by changing line.4605 in GDAL/frmts/netcdf/netcdfdataset.cpp : 'if(nBands > 1) sprintf(szBandName,"%s%d",tmpMetadata,iBand);' –> 'if(nBands > 1) sprintf(szBandName,"%s",tmpMetadata);'

CreateCopy fails in case the band name has special characters, e.g. the slash in 'HH/VV'.

Metadata strings with special characters are escaped with XML/HTML encoding.

Examples

export all the bands into a netDCF 3 file

```
>>> n.export(netcdfile)
```

export all bands into a GeoTiff

```
>>> n.export(driver='GTiff')
```

export2thredds(*filename*, *bands=None*, *metadata=None*, *mask_name=None*, *no_mask_value=64*, *rm_metadata=None*, *time=None*, *created=None*, *zlib=True*)

Export data into a netCDF formatted for THREDDS server

Parameters

- **filename** (*str*) – output file name
- **bands** (*dict*) –

  {'band_name': {'type' ['>i1',] 'scale' : 0.1, 'offset' : 1000, 'metaKey1' : 'meta value 1', 'metaKey2' : 'meta value 2'}}

  dictionary sets parameters for band creation

  - 'type' - string representation of data type in the output band
  - 'scale' - sets scale_factor and applies scaling
  - 'offset' - sets 'scale_offset and applies offsetting
  - other entries (e.g. 'units': 'K') set other metadata
- **metadata** (*dict*) – Glbal metadata to add
- **mask_name** (*str*) – if data include a mask band: give the mask name. if None: no mask is added
- **no_mask_value** (*int*) – Non-masked value is 64.
- **rm_metadata** (*list*) – unwanted metadata names which will be removed
- **time** (*list with datetime objects*) – aqcuisition time of original data. That value will be in time dim
- **created** (*datetime*) – date of creation. Will be in metadata 'created'
- **zlib** (*bool*) – compress output netCDF files?

Note: Nansat object (self) has to be projected (with valid GeoTransform and valid Spatial reference information) but not wth GCPs

Examples

create THREDDS formatted netcdf file with all bands and time variable

```
>>> n.export2thredds(filename)
```

export only one band and add global metadata

```
>>> n.export2thredds(filename, {'L_469': {'description': 'example'}})
```

export several bands and modify type, scale and offset

```
>>> bands = {'L_645' : {'type': '>i2', 'scale': 0.1, 'offset': 0},
             'L_555' : {'type': '>i2', 'scale': 0.1, 'offset': 0}}
```

```
>>> n.export2thredds(filename, bands)
```

nansat.figure module

class nansat.figure.**Figure**(*nparray*, ***kwargs*)

Bases: object

Perform operations with graphical files: create, append legend, save.

Figure instance is created in the Nansat.write_figure method. The methods below are applied consequently in order to generate a figure from one or three bands, estimate min/max, apply logarithmic scaling, convert to uint8, append legend, save to a file

Modifies: self.sizeX, self.sizeY (int), width and height of the image

Modifies: self.pilImg (PIL image), figure

Modifies: self.pilImgLegend (PIL image)

Note: If pilImgLegend is None, legend is not added to the figure. If it is replaced, pilImgLegend includes text string, color-bar, longName and units.

Parameters

- **array** (*numpy array (2D or 3D)*) – dataset from Nansat
- **cmin** (*number (int ot float) or [number, number, number]*) – 0, minimum value of varibale in the matrix to be shown
- **cmax** (*number (int ot float) or [number, number, number]*) – 1, minimum value of varibale in the matrix to be shown
- **gamma** (*float, >0*) – 2, coefficient for tone curve udjustment
- **subsetArraySize** (*int*) – 100000, size of the subset array which is used to get histogram

- **numOfColor** (*int*) – 250, number of colors for use of the palette. 254th is black and 255th is white.
- **cmapName** (*string*) – 'jet', name of Matplotlib colormaps see –> http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps
- **ratio** (*float, [0 1]*) – 1.0, ratio of pixels which are used to write the figure
- **numOfTicks** (*int*) – 5, number of ticks on a colorbar
- **titleString** (*string*) – '', title of legend (1st line)
- **caption** (*string*) – '', caption of the legend (2nd line, e.g. long name and units)
- **fontRatio** (*positive float*) – 1, factor for changing the fontSize.
- **fontSize** (*int*) – 12, size of the font of title, caption and ticks. If not given, fontSize is calculated using fontRatio: fontSize = height / 45 * fontRatio. fontSize has priority over fontRatio
- **logarithm** (*boolean, defult = False*) – If True, tone curve is used to convert pixel values. If False, linear.
- **legend** (*boolean, default = False*) – if True, information as textString, colorbar, longName and units are added in the figure.
- **mask_array** (*2D numpy array, int, the shape should be equal to*) – array.shape. If given, this array is used for masking land, clouds, etc on the output image. Value of the array are indices. LUT from mask_lut is used for coloring upon this indices.
- **mask_lut** (*dictionary*) – Look-Up-Table with colors for masking land, clouds etc. Used tgether with mask_array: {0, [0,0,0], 1, [100,100,100], 2: [150,150,150], 3: [0,0,255]} index
  - 0 - will have black color
  - 1 - dark gray
  - 2 - light gray
  - 3 - blue
- **logoFileName** (*string*) – name of the file with logo
- **logoLocation** (*list of two int, default = [0,0]*) – X and Y offset of the image If positive - offset is from left, upper edge If Negative - from right, lower edge Offset is calculated from the entire image legend inclusive
- **logoSize** (*list of two int*) – desired X,Y size of logo. If None - original size is used
- **latGrid** (*numpy array*) – full size array with latitudes. For adding lat/lon grid lines
- **lonGrid** (*numpy array*) – full size array with longitudes. For adding lat/lon grid lines
- **nGridLines** (*int*) – number of lat/lon grid lines to show
- **latlonLabels** (*int*) – number of lat/lon labels to show along each side.
- **transparency** (*int*) – transparency of the image background(mask), set for PIL alpha mask in Figure.save()
- **default** (*None*) –
- **LEGEND_HEIGHT** (*float, [0 1]*) – 0.1, legend height relative to image height
- **CBAR_HEIGHTMIN** (*int*) – 5, minimum colorbar height, pixels
- **CBAR_HEIGHT** (*float, [0 1]*) – 0.15, colorbar height relative to image height

- **CBAR_WIDTH** (*float [0 1]*) – 0.8, colorbar width relative to legend width
- **CBAR_LOCATION_X** (*float [0 1]*) – 0.1, colorbar offset X relative to legend width
- **CBAR_LOCATION_Y** (*float [0 1]*) – 0.5, colorbar offset Y relative to legend height
- **CBTICK_LOC_ADJUST_X** (*int*) – 5, colorbar tick label offset X, pixels
- **CBTICK_LOC_ADJUST_Y** (*int*) – 3, colorbar tick label offset Y, pixels
- **CAPTION_LOCATION_X** (*float, [0 1]*) – 0.1, caption offset X relative to legend width
- **CAPTION_LOCATION_Y** (*float, [0 1]*) – 0.1, caption offset Y relative to legend height
- **TITLE_LOCATION_X** (*float, [0 1]*) – 0.1, title offset X relative to legend width
- **TITLE_LOCATION_Y** – 0.3, title offset Y relative to legend height
- **DEFAULT_EXTENSION** (*string*) – '.png'

```
cmin = [0.0]
cmax = [1.0]
gamma = 2.0
subsetArraySize = 100000
numOfColor = 250
cmapName = 'jet'
ratio = 1.0
numOfTicks = 5
titleString = ''
caption = ''
fontRatio = 1
fontSize = None
logarithm = False
legend = False
mask_array = None
mask_lut = None
logoFileName = None
logoLocation = [0, 0]
logoSize = None
latGrid = None
lonGrid = None
lonTicks = 5
latTicks = 5
transparency = None
LEGEND_HEIGHT = 0.1
CBAR_HEIGHTMIN = 5
```

CBAR_HEIGHT = 0.15

CBAR_WIDTH = 0.8

CBAR_LOCATION_X = 0.1

CBAR_LOCATION_Y = 0.5

CBTICK_LOC_ADJUST_X = 5

CBTICK_LOC_ADJUST_Y = 3

CAPTION_LOCATION_X = 0.1

CAPTION_LOCATION_Y = 0.25

TITLE_LOCATION_X = 0.1

TITLE_LOCATION_Y = 0.05

DEFAULT_EXTENSION = '.png'

palette = None

pilImg = None

pilImgLegend = None

extensionList = ['png', 'PNG', 'tif', 'TIF', 'bmp', 'BMP', 'jpg', 'JPG', 'jpeg', 'JPEG']

array = None

apply_logarithm(***kwargs*)
Apply a tone curve to the array

After the normalization of the values from 0 to 1, logarithm is applied Then the values are converted to the normal scale.

Modifies: self.array (numpy array)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

apply_mask(***kwargs*)
Apply mask for coloring land, clouds, etc

If mask_array and mask_lut are provided as input parameters. The pixels in self.array which have index equal to mask_lut key in mask_array will have color equal to mask_lut value.

Modifies: self.array (numpy array)

Note: apply_mask should be called only after convert_palettesize (i.e. to uint8 data)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

add_logo(***kwargs*)
Insert logo into the PIL image

Read logo from file as PIL. Resize to the given size. Pan using the given location. Paste into pilImg.

Modifies: self.pilImg (PIL image)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

add_latlon_grids(**kwargs)
Add lat/lon grid lines into the PIL image

Compute step of the grid. Make matrices with binarized lat/lon. Find edge (make line). Convert to mask. Add mask to PIL

Modifies: self.pilImg (PIL image), added lat/lon grid lines

Parameters

- **latGrid** (*numpy array*) – array with values of latitudes
- **lonGrid** (*numpy array*) – array with values of longitudes
- **lonTicks** (*int or list*) – number of lines to draw or locations of gridlines
- **latTicks** (*int or list*) – number of lines to draw or locations of gridlines
- ****kwargs** (*dict*) – any of Figure parameters

add_latlon_labels(**kwargs)
Add lat/lon labels along upper and left side

Compute step of lables. Get lat/lon for these labels from latGrid, lonGrid Print lables to PIL in white.

Modifies: self.pilImg (PIL image), added lat/lon labels

Parameters

- **latGrid** (*numpy array*) – array with values of latitudes
- **lonGrid** (*numpy array*) – array with values of longitudes
- **lonTicks** (*int or list*) – number of lines to draw or locations of gridlines
- **latTicks** (*int or list*) – number of lines to draw or locations of gridlines
- ****kwargs** (*dict*) – Any of Figure parameters

clim_from_histogram(**kwargs)
Estimate min and max pixel values from histogram

if ratio=1.0, simply the minimum and maximum values are returned. if 0 < ratio < 1.0, get the histogram of the pixel values. Then get rid of (1.0-ratio)/2 from the both sides and return the minimum and maximum values.

Parameters **kwargs** (*dict*) – Any of Figure parameters

Returns **clim** – minimum and maximum pixel values for each band

Return type numpy array 2D ((3x2) or (1x2))

clip(**kwargs)
Convert self.array to values between cmin and cmax

if pixel value < cmin, replaced to cmin.

if pixel value > cmax, replaced to cmax.

Modifies: self.array (numpy array)

Modifies: self.cmin, self.cmax : allowed min/max values

Parameters **kwargs** (*dict*) – Any of Figure parameters

convert_palettesize(**kwargs)
Convert self.array to palette color size in uint8

Modifies: self.array (numpy array)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

create_legend(***kwargs*)
self.legend is replaced from None to PIL image

PIL image includes colorbar, caption, and titleString.

Modifies: self.legend (PIL image)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

create_pilImage(***kwargs*)
self.create_pilImage is replaced from None to PIL image

If three images are given, create a image with RGB mode. if self.pilImgLegend is not None, it is pasted.

If one image is given, create a image with P(palette) mode. if self.pilImgLegend is not None, self.array is extended before create the pilImag and then paste pilImgLegend onto it.

Modifies: self.pilImg (PIL image), PIL image with / without the legend

Modifies: self.array (replace to None)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

process(***kwargs*)
Do all common operations for preparation of a figure for saving

1. Modify default values of parameters by the provided ones (if any)
2. Clip to min/max
3. Apply logarithm if required
4. Convert data to uint8
5. Create palette
6. Apply mask for colouring land, clouds, etc if required
7. Create legend if required
8. Create PIL image
9. Add logo if required

Modifies: self.d

Modifies: self.array

Modifies: self.palette

Modifies: self.pilImgLegend

Modifies: self.pilImg

Parameters ****kwargs** (*dict*) – Any of Figure parameters

save(*fileName*, ***kwargs*)
Save self.pilImg to a physical file

If given extension is JPG, convert the image mode from Palette to RGB.

Modifies: self.pilImg (None)

Parameters

- **fileName** (*string*) – name of outputfile
- ****kwargs** (*dict*) – Any of Figure parameters

nansat.geolocation module

class nansat.geolocation.**Geolocation**(*x_vrt*, *y_vrt*, ***kwargs*)
Bases: object

Container for GEOLOCATION data

Keeps references to bands with X and Y coordinates, offset and step of pixel and line. All information is stored in dictionary self.data

Instance of Geolocation is used in VRT and ususaly created in a Mapper.

data = None

x_vrt = None

y_vrt = None

classmethod from_dataset(*dataset*)
Create geolocation from GDAL dataset :param dataset: input dataset to copy Geolocation metadata from :type dataset: gdal.Dataset

classmethod from_filenames(*x_filename*, *y_filename*, ***kwargs*)
Create geolocation from names of files with geolocation :param x_filename: name of file for X-dataset :type x_filename: str :param y_filename: name of file for Y-dataset :type y_filename: str :param **kwargs: parameters for self._init_data() :type **kwargs: dict

get_geolocation_grids()
Read values of geolocation grids

nansat.nansat module

class nansat.nansat.**Nansat**(*filename=''*, *mapper=''*, *log_level=30*, ***kwargs*)
Bases: *nansat.domain.Domain*, *nansat.exporter.Exporter*

Container for geospatial data. Performs all high-level operations.

n = Nansat(filename) opens the file with satellite or model data for reading, adds scientific metadata to bands, and prepares the data for further handling.

Parameters

- **filename** (*str*) – uri of the input file or OpeNDAP datastream
- **mapper** (*str*) – name of the mapper from nansat/mappers dir. E.g. 'sentinel1_l1', 'asar', 'hirlam', 'meris_l1', 'meris_l2', etc.
- **log_level** (*int*) – Level of logging. See: http://docs.python.org/howto/logging.html
- **kwargs** (*additional arguments for mappers*) –

Examples

```
>>> n1 = Nansat(filename)
>>> n2 = Nansat(sentinel1_filename, mapper='sentinel1_l1')
>>> array1 = n1[1]
>>> array2 = n2['sigma0_HV']
```

Notes

The instance of Nansat class (the object <n>) contains information about geographical reference of the data (e.g raster size, pixel resolution, type of projection, etc) and about bands with values of geophysical variables (e.g. water leaving radiance, normalized radar cross section, chlrophyll concentraion, etc). The object <n> has methods for high-level operations with data. E.g.: * reading data from file (Nansat.__getitem__); * visualization (Nansat.write_figure); * changing geographical reference (Nansat.reproject); * exporting (Nansat.export) * and much more...

Nansat inherits from Domain (container of geo-reference information) Nansat uses instance of VRT (wraper around GDAL VRT-files) Nansat uses instance of Figure (collection of methods for visualization)

FILL_VALUE = 9.96921e+36

ALT_FILL_VALUE = -10000.0

logger = None

filename = None

name = None

path = None

mapper = None

classmethod from_domain(*domain*, *array=None*, *parameters=None*, *log_level=30*)

Create Nansat object from input Domain [and array with data]

Parameters

- **domain** (Domain) – Defines spatial reference system and geographical extent.
- **array** (*numpy NDarray*) – Data for the first band. Shape must correspond to shape of <domain>
- **parameters** (*dict*) – Metadata for the first band. May contain 'name', 'wkv' and other keys.
- **log_level** (*int*) – Level of logging.

vrt = None

add_band(*array*, *parameters=None*, *nomem=False*)

Add band from numpy array with metadata.

Create VRT object which contains VRT and RAW binary file and append it to self.vrt.band_vrts

Parameters

- **array** (*ndarray*) – new band data. Shape should be equal to shape
- **parameters** (*dict*) – band metadata: wkv, name, etc. (or for several bands)
- **nomem** (*bool*) – saves the vrt to a tempfile on disk?

Notes

Creates VRT object with VRT-file and RAW-file. Adds band to the self.vrt.

Examples

```
>>> n.add_band(array, {'name': 'new_data'}) # add new band and metadata, keep␣
↪in memory
>>> n.add_band(array, nomem=True) # add new band, keep on disk
```

add_bands(*arrays*, *parameters=None*, *nomem=False*)

Add bands from numpy arrays with metadata.

Create VRT object which contains VRT and RAW binary file and append it to self.vrt.band_vrts

Parameters

- **arrays** (*list of ndarrays*) – new band data. Shape should be equal to shape
- **parameters** (*list of dict*) – band metadata: wkv, name, etc. (or for several bands)
- **nomem** (*bool*) – saves the vrt to a tempfile on disk?

Notes

Creates VRT object with VRT-file and RAW-file. Adds band to the self.vrt.

Examples

```
>>> n.add_bands([array1, array2]) # add new bands, keep in memory
```

bands()

Make a dictionary with all metadata from all bands

Returns **b** – key = N, value = dict with all band metadata

Return type dictionary

has_band(*band*)

Check if self has band with name <band> :param band: name or standard_name of the band to check :type band: str

Return type True/False if band exists or not

resize(*factor=None*, *width=None*, *height=None*, *pixelsize=None*, *resample_alg=- 1*)

Proportional resize of the dataset.

The dataset is resized as (x_size*factor, y_size*factor) If desired width, height or pixelsize is specified, the scaling factor is calculated accordingly. If GCPs are given in a dataset, they are also rewritten.

Parameters

- **factor** (*float, optional, default=1*) – Scaling factor for width and height
  - > 1 means increasing domain size
  - < 1 means decreasing domain size
- **width** (*int, optional*) – Desired new width in pixels

- **height** (*int, optional*) – Desired new height in pixels
- **pixelsize** (*float, optional*) – Desired new pixelsize in meters (approximate). A factor is calculated from ratio of the current pixelsize to the desired pixelsize.
- **resample_alg** (*int (GDALResampleAlg), optional*) –
  - -1 : Average (default),
  - 0 : NearestNeighbour
  - 1 : Bilinear,
  - 2 : Cubic,
  - 3 : CubicSpline,
  - 4 : Lancoz

Notes

self.vrt.dataset [VRT dataset of VRT object] raster size are modified to downscaled size. If GCPs are given in the dataset, they are also overwritten.

get_GDALRasterBand(*band_id=1*)

Get a GDALRasterBand of a given Nansat object

If str is given find corresponding band number If int is given check if band with this number exists. Get a GDALRasterBand from vrt.

Parameters **band_id** (*int or str*) –

- if int - a band number of the band to fetch
- if str band_id = {'name': band_id}

Return type GDAL RasterBand

Example

```
>>> b = n.get_GDALRasterBand(1)
>>> b = n.get_GDALRasterBand('sigma0')
```

list_bands(*do_print=True*)

Show band information of the given Nansat object

Show serial number, longName, name and all parameters for each band in the metadata of the given Nansat object.

Parameters **do_print** (*boolean*) – print on screen?

Returns **outString** – formatted string with bands info

Return type String

reproject(*dst_domain=None*, *resample_alg=0*, *block_size=None*, *tps=None*, *skip_gcps=1*, *addmask=True*, ***kwargs*)

Change projection of the object based on the given Domain

Create superVRT from self.vrt with AutoCreateWarpedVRT() using projection from the dst_domain. Modify XML content of the warped vrt using the Domain parameters. Generate warpedVRT and replace self.vrt

with warpedVRT. If current object spans from 0 to 360 and dst_domain is west of 0, the object is shifted by 180 westwards.

Parameters

- **dst_domain** (*domain*) – destination Domain where projection and resolution are set
- **resample_alg** (*int (GDALResampleAlg)*) –
  - 0 : NearestNeighbour
  - 1 : Bilinear
  - 2 : Cubic,
  - 3 : CubicSpline
  - 4 : Lancoz
- **block_size** (*int*) – size of blocks for resampling. Large value decrease speed but increase accuracy at the edge
- **tps** (*bool*) – Apply Thin Spline Transformation if source or destination has GCPs Usage of TPS can also be triggered by setting self.vrt.tps=True before calling to reproject. This options has priority over self.vrt.tps
- **skip_gcps** (*int*) – Using TPS can be very slow if the number of GCPs are large. If this parameter is given, only every [skip_gcp] GCP is used, improving calculation time at the cost of accuracy. If not given explicitly, 'skip_gcps' is fetched from the metadata of self, or from dst_domain (as set by mapper or user). [defaults to 1 if not specified, i.e. using all GCPs]
- **addmask** (*bool*) – If True, add band 'swathmask'. 1 - valid data, 0 no-data. This band is used to replace no-data values with np.nan

Notes

self.vrt : VRT object with dataset replaced to warpedVRT dataset Integer data is returnd by integer. Round off to decimal place. If you do not want to round off, convert the data types to GDT_Float32, GDT_Float64, or GDT_CFloat32.

See also:

http://www.gdal.org/gdalwarp.html

undo(*steps=1*)

Undo reproject, resize, add_band or crop of Nansat object

Restore the self.vrt from self.vrt.vrt

Parameters **steps** (*int*) – How many steps back to undo

Notes

Modifies self.vrt

watermask(*mod44path=None*, *dst_domain=None*, ***kwargs*)
Create numpy array with watermask (water=1, land=0)

250 meters resolution watermask from MODIS 44W Product: http://www.glcf.umd.edu/data/watermask/

Watermask is stored as tiles in TIF(LZW) format and a VRT file All files are stored in one directory. A tarball with compressed TIF and VRT files should be additionally downloaded from the Nansat documentation page: http://nansat.readthedocs.io/en/latest/source/features.html#differentiating-between-land-and-water

The method : Gets the directory either from input parameter or from environment variable MOD44WPATH Open Nansat object from the VRT file Reprojects the watermask onto the current object using reproject() or reproject_on_jcps() Returns the reprojected Nansat object

Parameters

- **mod44path** (*string*) – path with MOD44W Products and a VRT file
- **dst_domain** (Domain) – destination domain other than self
- **tps** (*Bool*) – Use Thin Spline Transformation in reprojection of watermask? See also Nansat.reproject()
- **skip_gcps** (*int*) – Factor to reduce the number of GCPs by and increase speed See also Nansat.reproject()

Returns watermask

Return type Nansat object with water mask in current projection

See also:

http://www.glcf.umd.edu/data/watermask/

http://nansat.readthedocs.io/en/latest/source/features.html#differentiating-between-land-and-water

write_figure(*filename=''*, *bands=1*, *clim=None*, *addDate=False*, *array_modfunc=None*, ***kwargs*)
Save a raster band to a figure in graphical format.

Get numpy array from the band(s) and band information specified either by given band number or band id. – If three bands are given, merge them and create PIL image. – If one band is given, create indexed image Create Figure object and: Adjust the array brightness and contrast using the given min/max or histogram. Apply logarithmic scaling of color tone. Generate and append legend. Save the PIL output image in PNG or any other graphical format. If the filename extension is 'tif', the figure file is converted to GeoTiff

Parameters

- **filename** (*str*) – Output file name. if one of extensions 'png', 'PNG', 'tif', 'TIF', 'bmp', 'BMP', 'jpg', 'JPG', 'jpeg', 'JPEG' is included, specified file is created. otherwise, 'png' file is created.
- **bands** (*integer or string or list (elements are integer or string),*) – default = 1 the size of the list has to be 1 or 3. if the size is 3, RGB image is created based on the three bands. Then the first element is Red, the second is Green, and the third is Blue.
- **clim** (*list with two elements or 'hist' to specify range of colormap*) – None (default) : min/max values are fetched from WKV, fallback-'hist' [min, max] : min and max are numbers, or [[min, min, min], [max, max, max]]: three bands used 'hist' : a histogram is used to calculate min and max values

- **addDate** (*boolean*) – False (default) : no date will be aded to the caption True : the first time of the object will be added to the caption
- **array_modfunc** (*None*) – None (default) : figure created using array in provided band function : figure created using array modified by provided function
- ****kwargs** (*parameters for Figure().*) –

Notes

if filename is specified, creates image file

Returns **Figure** – filename extension define format (default format is png)

Return type Figure object

Example

```
>>> n.write_figure('test.jpg') # write indexed image
>>> n.write_figure('test_rgb_hist.jpg', clim='hist', bands=[1, 2, 3]) # RGB␣
↪image
>>> n.write_figure('r09_log3_leg.jpg', logarithm=True, legend=True,
                   gamma=3, titleString='Title', fontSize=30,
                   numOfTicks=15) # add legend
>>> n.write_figure(filename='transparent.png', bands=[3],
                   mask_array=wmArray,
                   mask_lut={0: [0,0,0]},
                   clim=[0,0.15], cmapName='gray',
                   transparency=[0,0,0]) # write transparent image
```

See also:

Figure()

http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

write_geotiffimage(*filename*, *band_id=1*)

Writes an 8-bit GeoTiff image for a given band.

The colormap is fetched from the metadata item 'colormap'. Fallback colormap is 'gray'.

Color limits are fetched from the metadata item 'minmax'. If 'minmax' is not specified, min and max of the raster data is used.

The method can be replaced by using nansat.write_figure(). However, write_figure uses PIL, which does not allow Tiff compression. This gives much larger files.

Parameters

- **filename** (*str*) –
- **band_id** (*int or str*) –

property time_coverage_start

property time_coverage_end

get_metadata(*key=None*, *band_id=None*, *unescape=True*)

Get metadata from self.vrt.dataset

Parameters

- **key** (*str*) – name of the metadata key. If not givem all metadata is returned
- **band_id** (*int or str*) – number or name of band to get metadata from. If not given, global metadata is returned
- **unescape** (*bool*) – Replace '"', '&', '<' and '>' with these symbols " & < > ?

Returns

- **metadata** (*str*) – string with metadata if key is given and found
- **metadata** (*dict*) – dictionary with all metadata if key is not given

Raises **ValueError, if key is not found** –

set_metadata(*key=''*, *value=''*, *band_id=None*)
Set metadata to self.vrt.dataset

Parameters

- **key** (*string or dictionary with strings*) – name of the metadata, or dictionary with metadata names, values
- **value** (*string*) – value of metadata
- **band_id** (*int or str*) – number or name of band Without : global metadata is set

Notes

self.vrt.dataset : sets metadata in GDAL current dataset

get_band_number(*band_id*)
Return absolute band number

Check if given band_id is valid Return absolute number of the band in the VRT

Parameters **band_id** (*int or str or dict*) –

- if int : checks if such band exists and returns band_id
- if str : finds band with coresponding name
- if dict : finds first band with given metadata

Returns **absolute band number**

Return type int

get_transect(*points*, *bands*, *lonlat=True*, *smooth_radius=0*, *smooth_function=numpy.nanmedian*, *data=None*, *cornersonly=False*)
Get values from transect from given vector of poins

Parameters

- **points** (*2xN list or array, N (number of points) >= 1*) – coordinates [[x1, x2, y2], [y1, y2, y3]]
- **bands** (*list of int or string*) – elements of the list are band number or band Name
- **lonlat** (*bool*) – If the points in lat/lon, then True. If the points in pixel/line, then False.
- **smooth_radius** (*int*) – If smootRadius is greater than 0, smooth every transect pixel as the median or mean value in a circule with radius equal to the given number.
- **smooth_function** (*func*) – function for averaging values collected within smooth radius

- **data** (*ndarray*) – alternative array with data to take values from

Returns **transect**

Return type numpy record array

digitize_points(*band=1*, ***kwargs*)
Get coordinates of interactively digitized points

Parameters

- **band** (*int or str*) – ID of Nansat band
- ****kwargs** (*keyword arguments for imshow*) –

Returns **points** – list of 2xN arrays of points to be used in Nansat.get_transect()

Return type list

crop_interactive(*band=1*, *maxwidth=1000*, ***kwargs*)
Interactively select boundary and crop Nansat object

Parameters

- **band** (*int or str*) – id of the band to show for interactive selection of boundaries
- **maxwidth** (*int*) – large input data is downscaled to <maxwidth>
- ****kwargs** (*keyword arguments for imshow*) –

Notes

self.vrt [VRT] superVRT is created with modified SrcRect and DstRect

Returns **extent** – x_offset - X offset in the original dataset y_offset - Y offset in the original dataset x_size - width of the new dataset y_size - height of the new dataset

Return type (x_offset, y_offset, x_size, y_size)

Examples

```
>>> extent = n.crop_interactive(band=1) # crop a subimage interactively
```

crop_lonlat(*lonlim*, *latlim*)
Crop Nansat object to fit into given longitude/latitude limit

Parameters

- **lonlim** (*list of 2 float*) – min/max of longitude
- **latlim** (*list of 2 float*) – min/max of latitude

Notes

self.vrt [VRT] crops vrt to size that corresponds to lon/lat limits

Returns extent – x_offset - X offset in the original dataset y_offset - Y offset in the original dataset x_size - width of the new dataset y_size - height of the new dataset

Return type (x_offset, y_offset, x_size, y_size)

Examples

```
>>> extent = n.crop(lonlim=[-10,10], latlim=[-20,20]) # crop for given lon/lat␣
↪limits
```

crop(*x_offset*, *y_offset*, *x_size*, *y_size*, *allow_larger=False*)
Crop Nansat object

Create superVRT, modify the Source Rectangle (SrcRect) and Destination Rectangle (DstRect) tags in the VRT file for each band in order to take only part of the original image, create new GCPs or new GeoTransform for the cropped object.

Parameters

- **x_offset** (*int*) – pixel offset of subimage
- **y_offset** (*int*) – line offset of subimage
- **x_size** (*int*) – width in pixels of subimage
- **y_size** (*int*) – height in pizels of subimage
- **allow_larger** (*bool*) – Allow resulting extent to be larger than the original image?

Notes

self.vrt : super-VRT is created with modified SrcRect and DstRect

Returns extent – x_offset - X offset in the original dataset y_offset - Y offset in the original dataset x_size - width of the new dataset y_size - height of the new dataset

Return type (x_offset, y_offset, x_size, y_size)

Examples

```
>>> extent = n.crop(10, 20, 100, 200)
```

extend(*left=0*, *right=0*, *top=0*, *bottom=0*)
Extend domain from four sides

Parameters

- **left** (*int*) – number of pixels to add from left side
- **right** (*int*) – number of pixels to add from right side
- **top** (*int*) – number of pixels to add from top side
- **bottom** (*int*) – number of pixels to add from bottom side

Notes

Canges self.vrt by adding nexgative offset or setting size to be large that original size.

nansat.node module

class nansat.node.**Node**(*tag*, *value=None*, ***attributes*)

Bases: object

Rapidly assemble XML using minimal coding.

By Bruce Eckel, (c)2006 MindView Inc. www.MindView.net Permission is granted to use or modify without payment as long as this copyright notice is retained.

Everything is a Node, and each Node can either have a value or subnodes. Subnodes can be appended to Nodes using '+=', and a group of Nodes can be strung together using '+'.

Create a node containing a value by saying Node('tag', 'value') You can also give attributes to the node in the constructor: Node('tag', 'value', attr1 = 'attr1', attr2 = 'attr2') or without a value: Node('tag', attr1 = 'attr1', attr2 = 'attr2')

To produce xml from a finished Node n, say n.xml() (for nicely formatted output) or n.rawxml().

You can read and modify the attributes of an xml Node using getAttribute(), setAttribute(), or delAttribute().

You can find the value of the first subnode with tag == 'tag' by saying n['tag']. If there are multiple instances of n['tag'], this will only find the first one, so you should use node() or nodeList() to narrow your search down to a Node that only has one instance of n['tag'] first.

You can replace the value of the first subnode with tag == 'tag' by saying n['tag'] = newValue. The same issues exist as noted in the above paragraph.

You can find the first node with tag == 'tag' by saying node('tag'). If there are multiple nodes with the same tag at the same level, use nodeList('tag').

The Node class is also designed to create a kind of 'domain specific language' by subclassing Node to create Node types specific to your problem domain.

This implementation uses xml.dom.minidom which is available in the standard Python 2.4 library. However, it can be retargeted to use other XML libraries without much effort.

getAttribute(*name*)

Read XML attribute of this node.

setAttribute(*name*, *item*)

Modify XML attribute of this node.

delAttribute(*name*)

Remove XML attribute with this name.

replaceAttribute(*name*, *value*)

replace XML arrtibute of this node.

node(*tag*, *elemNum=0*)

Recursively find the first subnode with this tag.

Parameters **elemNum** (*int*) – if there are several same tag, specify which element to take.

replaceNode(*tag*, *elemNum=0*, *newNode=None*)

Find the first subnode with this tag and replace with given node.

Parameters

- **tag** (*str*) – node tag
- **elemNum** (*int*) – number of subnode among other subnodes with similar tag

delNode(*tag*, *options=None*)
Recursively find nodes containing subnodes with this tag and remove subnodes

Parameters **options** (*dictionary*) – if there are several tags, specify a node by their attributes.

nodeList(*tag*)
Produce a list of subnodes with the same tag.

Note: It only makes sense to do this for the immediate children of a node. If you went another level down, the results would be ambiguous, so the user must choose the node to iterate over.

---

tagList()
Produce a list of all tags of the immediate children

replaceTag(*oldTag*, *newTag*)
Replace tag name

getAttributeList()
get attributes and valuse from the node and return their lists

insert(*contents*)
return Node of the node with inserted <contents>

doc = <xml.dom.minidom.Document object>

dom()
Lazily create a minidom from the information stored in this Node object.

xml(*separator=' '*)

rawxml()

static create(*dom*)
Create a Node representation, given either a string representation of an XML doc, or a dom.

nansat.nsr module

class nansat.nsr.**NSR**(**args: Any*, ***kwargs: Any*)
Bases: osr.SpatialReference, object

Nansat Spatial Reference. Overrides constructor of osr.SpatialReference.

Parameters **srs** (*0, PROJ4 or EPSG or WKT or osr.SpatialReference,* NSR) – Specifies spatial reference system (SRS) PROJ4: string with proj4 options [http://trac.osgeo.org/proj/] e.g.: '+proj=latlong +datum=WGS84 +ellps=WGS84 +no_defs' '+proj=stere +datum=WGS84 +ellps=WGS84 +lat_0=75 +lon_0=0 +no_defs' EPSG: integer with EPSG number, [http://spatialreference.org/], e.g. 4326 WKT: string with Well Know Text of SRS. E.g.: 'GEOGCS["WGS 84",

DATUM["WGS_1984",

SPHEROID["WGS 84",6378137,298.257223563, AUTHORITY["EPSG","7030"]],

TOWGS84[0,0,0,0,0,0,0], AUTHORITY["EPSG","6326"]],

PRIMEM["Greenwich",0, AUTHORITY["EPSG","8901"]],

UNIT[“degree”,0.0174532925199433, AUTHORITY[“EPSG”,”9108”]],

AUTHORITY[“EPSG”,”4326”]]’

property wkt

Well Known Text representation of SRS

nansat.pointbrowser module

class nansat.pointbrowser.**PointBrowser**(*data*, *fmt='x-k'*, *force_interactive=True*, ***kwargs*)

Bases: object

Click on raster images shown by plt.imshow and get the X-Y coordinates.

Parameters

- **data** (*ndarray*) – image to imshow
- **transect** (*bool*) – if True, get transects / points if False, get only points
- **force_interactive** (*bool*) – force PointBrowser to interactive mode? (True for regular use, False for tests)
- ****kwargs** (*dict*) – optional parameters for imshow

Note:

- self.fig : pyplot Figure
- self.data : ndarray with data
- self.ax : axes
- self.points : plot with points
- self.line : plot with points
- self.coordinates: container for recorded coordinates

---

fig = None

data = None

fmt = None

text_ax = None

ax = None

points = None

lines = None

coordinates = None

onclick(*event*)

Process mouse onclick event Append coordinates of the click to self.coordinates, add point and 2D line to self.points If click is outside, nothing is done If click with ‘z’ pressed, nothing is done If click with ‘anykey’, new line is started

Parameters **event** (*matplotlib.mouse_event*) –

get_points()
Enables the onclick events and returns the points.

The format of the returned array: [array([[x1,...,xn],[y1,...,yn]]),array([[xn+1,...],[yn+1,...]]),...] Each 'array' element is a numpy.ndarray and represents one transect, where x1,y1 is the first point in the first transect, and xn,yn the last point in the first transect. The inner x/y-arrays are also numpy.ndarrays

Returns points

Return type array

nansat.tools module

nansat.tools.**distance2coast**(*dst_domain*, *distance_src=None*)
Estimate distance to the nearest coast (in km) for each pixcel in the domain of interest. The method utilizes NASA's OBPG group Distance to the Nearest Coast product: https://oceancolor.gsfc.nasa.gov/docs/distfromcoast/. The product is stored in GeoTiff format with pixcelsize of 0.01x0.01 degree.

Parameters

- **dst_domain** (Domain) – destination domain
- **distance_src** (*str*) – path to the NASA Distance to the Nearest coast GeoTIFF product

Returns distance

Return type Nansat object with distance to the coast mask in current projection

See also:

https://oceancolor.gsfc.nasa.gov/docs/distfromcoast/ <*http://nansat.readthedocs.io/en/latest/source/features.html#differentiating-between-land-and-water*>

nansat.tools.**get_domain_map**(*domain*, *crs=None*, *lon_margin=10.0*, *lat_margin=10.0*, *lw=1*, *linestyle='b.-'*, *fill_color='coral'*, *fill_alpha=0.5*, *draw_gridlines=True*, *draw_labels=True*, *grid_lw=2*, *grid_color='gray'*, *grid_alpha=0.5*, *grid_linestyle='--'*, ***kwargs*)
Create a pyplot figure axis with Domain map, Cartopy projection, coastlines

Parameters

- **domain** (Domain) – the desired Domain to plot
- **crs** (*cartopy CRS or None*) – projection of the map, cartopy.crs.PlateCaree by default
- **lon_margin** (*float*) – 10, horisontal border around patch (degrees of longitude)
- **lat_margin** (*float*) – 10, vertical border around patch (degrees of latitude)
- **linestyle** (*str*) – domain line style
- **lw** (*float*) – domain line width
- **fill_color** (*str*) – domain fill color
- **fill_alpha** (*float*) – domain fill transparency
- **draw_gridlines** (*bool*) – Add gridlines to the plot?
- **draw_labels** (*bool*) – Add labels to the plot?
- **grid_lw** (*float*) – gridlines line width
- **grid_color** (*str,*) – gridlines color
- **grid_alpha** (*float*) – gridlines transparency

- **grid_linestyle** (*str*) – gridlines style

Returns **ax** – Axes with the map and domain patch

Return type pyplot/cartopy axes

nansat.tools.**show_domain_map**(*domain*, ***kwargs*)
Show Domain map interactively

Parameters

- **domain** (Domain) – the Domain to show
- ****kwargs** (*dict*) – parameters for nansat.tools.get_domain_map

nansat.tools.**save_domain_map**(*domain*, *filename*, *figsize=(5, 5)*, *dpi=150*, *bbox_inches='tight'*, *pad_inches=0*, ***kwargs*)
Save Domain to PNG file

Parameters

- **domain** (Domain) – the Domain to show
- **filename** (*str*) – destination filename
- **figsize** (*(int, int)*) – size of the figure in inches
- **dpi** (*PNG resolution*) –
- **bbox_inches** (*str or int*) – see pyplot.savefig
- **pad_inches** (*int*) – see pyplot.savefig
- ****kwargs** (*dict*) – parameters for nansat.tools.get_domain_map

nansat.tools.**deprecated**(*func*)

nansat.tools.**remove_keys**(*dict*, *keys*)

nansat.tools.**register_colormaps**()

nansat.tools.**initial_bearing**(*lon1*, *lat1*, *lon2*, *lat2*)

nansat.tools.**haversine**(*lon1*, *lat1*, *lon2*, *lat2*)

nansat.tools.**add_logger**(*logName=''*, *logLevel=None*)

nansat.tools.**get_random_color**(*c0=None*, *minDist=100*, *low=0*, *high=255*)

nansat.tools.**parse_time**(*time_string*)

nansat.vrt module

class nansat.vrt.**VRT**(*x_size=1*, *y_size=1*, *metadata=None*, *nomem=False*, ***kwargs*)
Bases: object

Wrapper around GDAL VRT-file

The GDAL VRT-file is an XML-file. It contains all metadata, geo-reference information and information ABOUT each band including band metadata, reference to the bands in the source file. VRT-class performs all operation on VRT-files: create, copy, modify, read, write, add band, add GeoTransform, set Projection, etc. It uses either GDAL methods for these operations (e.g. Create, AddBand, SetMetadata, AutoCreateWarpedVRT, etc.) or reads/writes the XML-file directly (e.g. remove_geotransform, get_warped_vrt, etc).

The core of the VRT object is GDAL dataset <self.dataset> generated by the GDAL VRT-Driver. The respective VRT-file is located in /vismem and has a random name.

GDAL data model doesn't have place for geolocaion arrays therefore VRT-object has instance of Geolocation (self.geolocation) an object to keep information about Geolocation metadata: reference to file with source data, pixel and line step and offset, etc.

Domain has an instance of VRT-class <self.vrt>. It keeps only geo- reference information.

All Mappers inherit from VRT. When Nansat opens a file it loops through list of mappers, selects the one appropriate for the input file, and creates an instance of Mapper.

Nansat has one instances of Mapper-class (>=VRT-class): self.vrt. It holds VRT-file in original projection (derived from the input file). After most of the operations with Nansat object (e.g. reproject, crop, resize, add_band) self.vrt is replaced with a new VRT object which has reference to the previous VRT object inside (self.vrt.vrt).

Parameters

- **x_size** (*int*) – width of self.dataset
- **y_size** (*int*) – arguments for VRT()
- **metadata** (*dict*) – dictionray with metadata keys (str) and values (str)
- **nomem** (*bool*) – don't create VRT in VSI memory?

Notes

adds self.logger, self.driver, self.filename, self.band_vrts, self.tps, self.vrt adds self.dataset - GDAL Dataset without bands and with size=(x_zie, y_size) adds metadata to self.dataset writes VRT file content to self.filename

COMPLEX_SOURCE_XML = <string.Template object>

RAW_RASTER_BAND_SOURCE_XML = <string.Template object>

REPROJECT_TRANSFORMER = <string.Template object>

geolocation = None

classmethod from_gdal_dataset(*gdal_dataset*, ***kwargs*)

Create VRT from GDAL Dataset with the same size/georeference but wihout bands.

Parameters

- **gdal_dataset** (*gdal.Dataset*) – input GDAL dataset
- ****kwargs** (*dict*) – arguments for VRT()

Returns vrt

Return type *VRT*

classmethod from_dataset_params(*x_size*, *y_size*, *geo_transform*, *projection*, *gcps*, *gcp_projection*, ***kwargs*)

Create VRT from GDAL Dataset parameters

Create VRT with dataset wihout bands but with size/georeference corresponding to input parameters.

Parameters

- **x_size** (*int*) – X-size of dataset
- **y_size** (*int*) – Y-size of dataset
- **geotransform** (*tuple with 6 floats*) – informaton on affine transformtaion
- **projection** (*str*) – WKT representation of spatial reference system

- **gcps** (*tuple or list with GDAL GCP objects*) – GDAL Ground Control Points
- **gcp_projection** (*str*) – WKT representation of GCPs spatial reference system
- ****kwargs** (*dict*) – arguments for VRT()

Returns vrt

Return type *VRT*

classmethod from_array(*array*, ***kwargs*)

Create VRT from numpy array with dataset wih one band but without georeference.

Parameters

- **array** (*numpy.ndarray*) – array with data
- ****kwargs** (*dict*) – arguments for VRT()

Returns vrt

Return type *VRT*

classmethod from_lonlat(*lon*, *lat*, *add_gcps=True*, ***kwargs*)

Create VRT from longitude, latitude arrays

Create VRT with dataset without bands but with GEOLOCATION metadata and Geolocation object. Geolocation contains 2 2D arrays with lon/lat values given at regular pixel/line steps. GCPs can bee created from lon/lat arrays and added to the dataset

Parameters

- **lon** (*numpy.ndarray*) – array with longitudes
- **lat** (*numpy.ndarray*) – array with latitudes
- **add_gcps** (*bool*) – Create GCPs from lon/lat arrays and add to dataset
- ****kwargs** (*dict*) – arguments for VRT()

Returns vrt

Return type *VRT*

classmethod copy_dataset(*gdal_dataset*, ***kwargs*)

Create VRT with bands and georefernce as a full copy of input GDAL Dataset

Parameters

- **gdal_dataset** (*GDAL.Dataset*) – input dataset
- ****kwargs** (*dict*) – arguments for VRT()

Returns vrt

Return type *VRT*

logger = None

driver = None

filename = ''

band_vrts = None

tps = None

vrt = None

dataset = None

leave_few_bands(*bands=None*)
Leave only given bands in VRT

split_complex_bands()
Recursevly find complex bands and relace by real and imag components

create_geolocation_bands()
Create bands from Geolocation

fix_band_metadata(*rm_metadata*)
Add NETCDF_VARNAME and remove <rm_metadata> in metadata for each band

fix_global_metadata(*rm_metadata*)
Remove unwanted global metadata and escape special characters

hardcopy_bands()
Make 'hardcopy' of bands: evaluate array from band and put into original band

prepare_export_gtiff()
Prepare dataset for export using GTiff driver

prepare_export_netcdf()
Prepare dataset for export using netCDF driver

copy()
Create and return a full copy of a VRT instance with new filenames

If self.dataset has no bands, the copy is created also without bands. If self.dataset has bands, the copy is created from the dataset with all bands. If self has attribute 'vrt' (a sub-VRT object, result of get_super_vrt) it's contents is also copied into sub-VRT of the copy. Other attributes of self, such as tps flag and band_vrts are also copied.

property xml
Read XML content of the VRT-file using VSI

Returns string

Return type XMl Content which is read from the VSI file

create_bands(*metadata_dict*)
Generic function called from the mappers to create bands in the VRT dataset from an input dictionary of metadata

Parameters **metadata_dict** (*list of dict with params of input bands and generated bands.*) – Each dict has: 'src' : dictionary with parameters of the sources: 'dst' : dictionary with parameters of the generated bands

Notes

Adds bands to the self.dataset based on info in metaDict

See also:

VRT.create_band

create_band(*src*, *dst=None*)
Add band to self.dataset:

Get parameters of the source band(s) from input Generate source XML for the VRT, add options of creating Call GDALDataset.AddBand Set source and options Add metadata

Parameters

- **src** (*dict with parameters of sources*) – SourceFilename, SourceBand, ScaleRatio, ScaleOffset, NODATA, LUT, SourceType, DataType, ImageOffset (RawVRT), PixelOffset (RawVRT), LineOffset (RawVRT), ByteOrder (RawVRT), xSize, ySize
- **dst** (*dict with parameters of the created band*) – name, dataType, wkv, suffix, AnyOtherMetadata, PixelFunctionType (1) band will be a pixel function defined by the corresponding name/value. In this case src may be list of dicts with parameters for each source. 2) in case the dst band has a different datatype than the source band it is important to add a SourceTransferType parameter in dst), SourceTransferType

Returns **name**

Return type string, name of the added band

Examples

```
>>> vrt.create_band({'SourceFilename': filename, 'SourceBand': 1})
```

```
>>> vrt.create_band({'SourceFilename': filename, 'SourceBand': 2,
                     'ScaleRatio': 0.0001},
                    {'name': 'LAT', 'wkv': 'latitude'})
```

```
>>> vrt.create_band({'SourceFilename': filename, 'SourceBand': 2},
                    {'suffix': '670',
                     'wkv': 'brightness_temperature'})
```

```
>>> vrt.create_band([{'SourceFilename': filename, 'SourceBand': 1},
                     {'SourceFilename': filename, 'SourceBand': 1}],
                     {'PixelFunctionType': 'NameOfPixelFunction'})
```

write_xml(*vsi_file_content=None*)

Write XML content into a VRT dataset

Parameters **vsi_fileContent** (*string, optional*) – XML Content of the VSI file to write

Notes

self.dataset If XML content was written, self.dataset is re-opened

export(*filename*)

Export VRT file as XML into given <filename>

get_warped_vrt(*dst_srs*, *x_size*, *y_size*, *geo_transform*, *resample_alg=0*, *dst_gcps=[]*, *skip_gcps=1*, *block_size=None*, *working_data_type=None*, *resize_only=False*)

Create warped (reprojected) VRT object

1. Create a simple warped dataset using GDAL.AutoCreateWarpedVRT.
2. Create VRT from the warped dataset.
3. Modify the warped VRT according to the input options (size, geotransform, GCPs, etc)
4. Keep the original VRT in the attribute vrt

For reprojecting the function tries to use geolocation by default, if geolocation is not present it tries to use GCPs, if GCPs are not present it uses GeoTransform.

If destination image has GCPs (provided in <dst_gcps>): fake GCPs for referencing line/pixel of SRC image and X/Y of DST image are created and added to the SRC image. After warping the dst_gcps are added to the warped VRT.

Parameters

- **dst_srs** (*string*) – WKT of the destination projection
- **x_size** (*int*) – dimentions of the destination rasetr
- **y_size** (*int*) – dimentions of the destination rasetr
- **geo_transform** (*tuple with 6 floats*) – destination GDALGeoTransfrom
- **resample_alg** (*int (GDALResampleAlg)*) – 0 : NearestNeighbour, 1 : Bilinear, 2 : Cubic, 3 : CubicSpline, 4 : Lancoz
- **dst_gcps** (*list with GDAL GCPs*) – GCPs of the destination image
- **skip_gcps** (*int*) – Using TPS can be very slow if the number of GCPs are large. If this parameter is given, only every [skip_gcp] GCP is used, improving calculation time at the cost of accuracy.
- **block_size** (*int*) – BlockSize to use for resampling. Larger blocksize reduces speed but increases accuracy.
- **working_data_type** (*str*) – 'Float32', 'Int16', etc.
- **resize_only** (*bool*) – Create warped_vrt which will be used for resizing only?

Returns warped_vrt

Return type VRT object with warped dataset and with vrt

copyproj(*filename*)

Copy geoloctation data from given VRT to a figure file

Useful for adding geolocation information to figure files produced e.g. by Figure class, which contain no geolocation. Analogue to utility gdalcopyproj.py.

Parameters **filename** (*string*) – Name of file to which the geolocation data shall be written

delete_band(*band_num*)

Delete a band from the given VRT

Parameters **band_num** (*int*) – band number

delete_bands(*band_nums*)

Delete bands

Parameters **bandNums** (*list*) – elements are int

get_shifted_vrt(*shift_degree*)

Roll data in bands westwards or eastwards

Create shift_vrt which references self. Modify georeference of shift_vrt to account for the roll. Add as many bands as in self but for each band create two complex sources: for western and eastern parts. Keep self in shift_vrt.vrt

Parameters **shift_degree** (*float*) – rolling angle, how far east/west to roll

Returns shift_vrt

Return type VRT object with rolled bands

get_sub_vrt(*steps=1*)
Returns sub-VRT from given depth

Iteratively copy self.vrt into self until self.vrt is None or steps == 0

Parameters **steps** (*int*) – How many sub VRTs to restore

Returns

- **self** (*if no deeper VRTs found*)
- **self.vrt** (*if deeper VRTs are found*)

Notes

self self.vrt

get_super_vrt()
Create a new VRT object with a reference to the current object (self)

Create a new VRT (super_vrt) with exactly the same structure (number of bands, raster size, metadata) as the current object (self). Create a copy of the current object and add it as an attribute of the new object (super_vrt.vrt). Bands in the new object will refer to the same bands in the current object. Recursively copy all vrt attributes of the current object (self.vrt.vrt.vrt...) into the new object (super_vrt.vrt.vrt.vrt.vrt...).

Returns **super_vrt** – a new VRT object with copy of self in super_vrt.vrt

Return type *VRT*

get_subsampled_vrt(*new_raster_x_size*, *new_raster_y_size*, *resample_alg*)
Create VRT and replace step in the source

transform_points(*col_vector*, *row_vector*, *dst2src=0*, *dst_srs=None*, *dst_ds=None*, *options=None*)
Transform input pixel/line coordinates into lon/lat (or opposite)

Parameters

- **col_vector** (*lists*) – X and Y coordinates with any coordinate system
- **row_vector** (*lists*) – X and Y coordinates with any coordinate system
- **dst2src** (*0 or 1*) – 1 for inverse transformation, 0 for forward transformation.
- **dst_srs** (NSR) – destination SRS.
- **dst_ds** (*GDAL Dataset*) – destination dataset. The default is None. It means transform ownPixLin <–> ownXY.
- **option** (*string*) – if 'METHOD=GEOLOC_ARRAY', specify here.

Returns **lon_vector, lat_vector** – X and Y coordinates in degree of lat/lon

Return type numpy arrays

get_projection()
Get projection (spatial reference system) of the dataset

Uses dataset.GetProjection() or dataset.GetGCPProjection() or dataset.GetMetadata('GEOLOCATION')['SRS']

Returns

- **projection** (*projection WKT*)
- **source** (*str ['gcps' or 'dataset' or 'geolocation']*)

:raises NansatProjectionError : occurs when the projection is empty.:

get_resized_vrt(*x_size*, *y_size*, *resample_alg=1*)
Resize VRT

Create Warped VRT with modified RasterXSize, RasterYSize, GeoTransform. The returned VRT object has a copy of its original source VRT in its own vrt object (e.g. warpedVRT.vrt = originalVRT.copy()).

Parameters

- **x_size** (*int*) – new size of the VRT object
- **y_size** (*int*) – new size of the VRT object
- **resample_alg** (*GDALResampleAlg*) – see also gdal.AutoCreateWarpedVRT

Returns **warped_vrt**

Return type Resized VRT object

reproject_gcps(*dst_srs*)
Reproject all GCPs to a new spatial reference system

Necessary before warping an image if the given GCPs are in a coordinate system which has a singularity in (or near) the destination area (e.g. poles for lonlat GCPs)

Parameters **dst_srs** (*proj4, WKT, NSR, EPSG*) – Destiination SRS given as any NSR input parameter

Notes

Reprojects all GCPs to new SRS and updates GCPProjection

static transform_coordinates(*src_srs*, *src_points*, *dst_srs*)
Transform coordinates of points from one spatial reference to another

Parameters

- **src_srs** (nansat.NSR) – Source spatial reference system
- **src_points** (*tuple of two or three N-D arrays*) – Coordinates of points in the source spatial reference system. A tuple with (X, Y) or (X, Y, Z) coordinates arrays. Each array can be a list, 1D, 2D, N-D array.
- **dst_srs** (nansat.NSR) – Destination spatial reference

Returns **dst_points** – Coordinates of points in the destination spatial reference system. A tuple with (X, Y) or (X, Y, Z) coordinates arrays. Each array can be 1D, 2D, N-D. Shape of output arrays corrrrespond to shape of inputs.

Return type tuple of two or three N-D arrays

set_offset_size(*axis*, *offset*, *size*)
Set offset and size in VRT dataset and band attributes

Parameters

- **axis** (*str*) – name of axis ('x' or 'y')
- **offset** (*int*) – value of offset to put into VRT
- **size** (*int*) – value of size to put into VRT

Notes

Changes VRT file, sets new offset and size

shift_cropped_gcps(*x_offset*, *x_size*, *y_offset*, *y_size*)
Modify GCPs to fit the size/offset of cropped image

shift_cropped_geo_transform(*x_offset*, *x_size*, *y_offset*, *y_size*)
Modify GeoTransform to fit the size/offset of the cropped image

static read_vsi(*filename*)
Read text from input <filename:str> using VSI and return <content:str>.

nansat.warnings module

exception nansat.warnings.**NansatFutureWarning**
Bases: Warning

Module contents

class nansat.**NSR**(**args: Any*, ***kwargs: Any*)
Bases: osr.SpatialReference, object

Nansat Spatial Reference. Overrides constructor of osr.SpatialReference.

Parameters **srs** (*0, PROJ4 or EPSG or WKT or osr.SpatialReference,* NSR) – Specifies spatial reference system (SRS) PROJ4: string with proj4 options [http://trac.osgeo.org/proj/] e.g.: '+proj=latlong +datum=WGS84 +ellps=WGS84 +no_defs' '+proj=stere +datum=WGS84 +ellps=WGS84 +lat_0=75 +lon_0=0 +no_defs' EPSG: integer with EPSG number, [http://spatialreference.org/], e.g. 4326 WKT: string with Well Know Text of SRS. E.g.: 'GEOGCS["WGS 84",

DATUM["WGS_1984",

SPHEROID["WGS 84",6378137,298.257223563, AUTHORITY["EPSG","7030"]],

TOWGS84[0,0,0,0,0,0,0], AUTHORITY["EPSG","6326"]],

PRIMEM["Greenwich",0, AUTHORITY["EPSG","8901"]],

UNIT["degree",0.0174532925199433, AUTHORITY["EPSG","9108"]],

AUTHORITY["EPSG","4326"]]'

property wkt
Well Known Text representation of SRS

class nansat.**Domain**(*srs=None*, *ext=None*, *ds=None*, ***kwargs*)
Bases: object

Container for geographical reference of a raster

A Domain object describes all attributes of geographical reference of a raster:

- width and height (number of pixels)
- pixel size (e.g. in decimal degrees or in meters)
- relation between pixel/line coordinates and geographical coordinates (e.g. a linear relation)
- type of data projection (e.g. geographical or stereographic)

Parameters

- **srs** (*PROJ4 or EPSG or WKT or NSR or osr.SpatialReference()*) – Input parameter for nansat.NSR()
- **ext** (*string*) – some gdalwarp options + additional options [http://www.gdal.org/gdalwarp.html] Specifies extent, resolution / size Available options: ((‘-te’ or ‘-lle’) and (‘-tr’ or ‘-ts’)) (e.g. ‘-lle -10 30 55 60 -ts 1000 1000’ or ‘-te 100 2000 300 10000 -tr 300 200’) -tr resolutionx resolutiony -ts sizex sizey -te xmin ymin xmax ymax -lle lonmin latmin lonmax latmax
- **ds** (*GDAL dataset*) –

Examples

```
>>> d = Domain(srs, ext) #size, extent and spatial reference is given by strings
>>> d = Domain(ds=GDALDataset) #size, extent copied from input GDAL dataset
>>> d = Domain(srs, ds=GDALDataset) # spatial reference is given by srs,
    but size and extent is determined from input GDAL dataset
```

Notes

The core of Domain is a GDAL Dataset. It has no bands, but only georeference information: rasterXsize, rasterYsize, GeoTransform and Projection or GCPs, etc. which fully describe dimentions and spatial reference of the grid.

There are three ways to store geo-reference in a GDAL dataset:

- Using GeoTransfrom to define linear relationship between raster pixel/line and geographical X/Y coordinates
- Using GCPs (set of Ground Control Points) to define non-linear relationship between pixel/line and X/Y
- Using Geolocation Array - full grids of X/Y coordinates for each pixel of a raster

The relation between X/Y coordinates of the raster and latitude/longitude coordinates is defined by projection type and projection parameters. These pieces of information are therefore stored in Domain:

- Type and parameters of projection +
  - GeoTransform, or
  - GCPs, or
  - GeolocationArrays

Domain has methods for basic operations with georeference information:

- creating georeference from input options;
- fetching corner, border or full grids of X/Y coordinates;
- making map of the georeferenced grid in a PNG or KML file;
- and some more…

The main attribute of Domain is a VRT object self.vrt. Nansat inherits from Domain and adds bands to self.vrt

:raises NansatProjectionError : occurs when Projection() is empty: despite it is required for creating extentDic.
:raises OptionError : occures when the arguments are not proper.:

See also:

Nansat.reproject()

http://www.gdal.org/gdalwarp.html

http://trac.osgeo.org/proj/

http://spatialreference.org/

http://www.gdal.org/osr_tutorial.html

OUTPUT_SEPARATOR = '----------\n'

KML_BASE = '<?xml version="1.0" encoding="UTF-8"?>\n <kml xmlns="http://www.opengis.net/kml/2.2"\n xmlns:gx="http://www.google.com/kml/ext/2.2"\n xmlns:kml="http://www.opengis.net/kml/2.2"\n xmlns:atom="http://www.w3.org/2005/Atom">\n {content}\n </kml>'

logger = None

name = None

vrt = None

classmethod from_lonlat(*lon*, *lat*, *add_gcps=True*)

Create Domain object from input longitudes, latitudes arrays

Parameters

- **lon** (*numpy.ndarray*) – longitudes
- **lat** (*numpy.ndarray*) – latitudes
- **add_gcps** (*bool*) – Add GCPs from lon/lat arrays.

Returns d

Return type *Domain*

Examples

```
>>> lon, lat = np.meshgrid(range(10), range(10))
>>> d1 = Domain.from_lonlat(lon, lat)
>>> d2 = Domain.from_lonlat(lon, lat, add_gcps=False) # add only geolocation␣
↪arrays
```

write_kml(*xmlFileName=None*, *kmlFileName=None*)

Write KML file with domains

Convert XML-file with domains into KML-file for GoogleEarth or write KML-file with the current Domain

Parameters

- **xmlFileName** (*string, optional*) – Name of the XML-file to convert. If only this value is given - kmlFileName=xmlFileName+'.kml'
- **kmlFileName** (*string, optional*) – Name of the KML-file to generate from the current Domain

write_kml_image(*kmlFileName*, *kmlFigureName=None*)

Create KML file for already projected image

Write Domain Image into KML-file for GoogleEarth

Parameters

- **kmlFileName** (*str*) – Name of the KML-file to generate from the current Domain
- **kmlFigureName** (*str*) – Name of the projected image stored in .png format

Examples

```
>>> n.undo(100) # cancel previous reprojection
>>> lons, lats = n.get_corners() # Get corners of the image and the pixel␣
↪resolution
>>> srsString = '+proj=latlong +datum=WGS84 +ellps=WGS84 +no_defs'
>>> extentString = '-lle %f %f %f %f -ts 3000 3000'
    % (min(lons), min(lats), max(lons), max(lats))
>>> d = Domain(srs=srsString, ext=extentString) # Create Domain with
    stereographic projection, corner coordinates and resolution 1000m
>>> n.reproject(d)
>>> n.write_figure(filename=figureName, bands=[3], clim=[0,0.15],
                   cmapName='gray', transparency=0)
>>> n.write_kml_image(kmlFileName=oPath + filename + '.kml',
                   kmlFigureName=figureName) # 6.
```

get_geolocation_grids(*stepSize=1*, *dst_srs=None*)

Get longitude and latitude grids representing the full data grid

If GEOLOCATION is not present in the self.vrt.dataset then grids are generated by converting pixel/line of each pixel into lat/lon If GEOLOCATION is present in the self.vrt.dataset then grids are read from the geolocation bands.

Parameters **stepSize** (*int*) – Reduction factor if output is desired on a reduced grid size

Returns

- **longitude** (*numpy array*) – grid with longitudes
- **latitude** (*numpy array*) – grid with latitudes

get_border(*n_points=10*, *fix_lon=True*, ***kwargs*)

Generate two vectors with values of lat/lon for the border of domain

Parameters

- **n_points** (*int, optional*) – Number of points on each border
- **fix_lon** (*bool*) – Convert longitudes to positive numbers when Domain crosses dateline?

Returns **lonVec, latVec** – vectors with lon/lat values for each point at the border

Return type lists

get_border_wkt(**args*, ***kwargs*)

Creates string with WKT representation of the border polygon

Returns **WKTPolygon** – string with WKT representation of the border polygon

Return type string

get_border_geometry(**args*, ***kwargs*)

Get OGR Geometry of the border Polygon

Returns **OGR Geometry**

Return type Polygon

`get_border_geojson`(**args*, ***kwargs*)
Create border of the Polygon in GeoJson format

Returns **the Polygon border in GeoJson format**

Return type str

`overlaps`(*anotherDomain*)
Checks if this Domain overlaps another Domain

Returns **overlaps** – True if Domains overlaps, False otherwise

Return type bool

`intersects`(*anotherDomain*)
Checks if this Domain intersects another Domain

Returns **intersects** – True if Domains intersects, False otherwise

Return type bool

`contains`(*anotherDomain*)
Checks if this Domain fully covers another Domain

Returns **contains** – True if this Domain fully covers another Domain, False otherwise

Return type bool

`get_border_postgis`(***kwargs*)
Get PostGIS formatted string of the border Polygon

Returns **'PolygonFromText(PolygonWKT)'**

Return type str

`get_corners`()
Get coordinates of corners of the Domain

Returns **lonVec, latVec** – vectors with lon/lat values for each corner

Return type lists

`get_min_max_lon_lat`()
Get minimum and maximum of longitude and latitude geolocation grids

Returns **min_lon, max_lon, min_lat, max_lat,** – min/max lon/lat values for the Domain

Return type float

`get_pixelsize_meters`()
Returns the pixelsize (deltaX, deltaY) of the domain

For projected domains, the exact result which is constant over the domain is returned. For geographic (lon-lat) projections, or domains with no geotransform, the haversine formula is used to calculate the pixel size in the center of the domain.

Returns **delta_x, delta_y** – pixel size in X and Y directions given in meters

Return type float

`transform_points`(*colVector*, *rowVector*, *DstToSrc=0*, *dst_srs=None*)
Transform given lists of X,Y coordinates into lon/lat or inverse

Parameters

- **colVector** (*lists*) – X and Y coordinates in pixel/line or lon/lat coordinate system
- **DstToSrc** (*0 or 1*) –
  - 0 - forward transform (pix/line => lon/lat)
  - 1 - inverse transformation
- **dst_srs** (NSR) – destination spatial reference

Returns **X, Y** – X and Y coordinates in lon/lat or pixel/line coordinate system

Return type lists

azimuth_y(*reductionFactor=1*)

Calculate the angle of each pixel position vector with respect to the Y-axis (azimuth).

In general, azimuth is the angle from a reference vector (e.g., the direction to North) to the chosen position vector. The azimuth increases clockwise from direction to North. http://en.wikipedia.org/wiki/Azimuth

Parameters **reductionFactor** (*integer*) – factor by which the size of the output array is reduced

Returns **azimuth** – Values of azimuth in degrees in range 0 - 360

Return type numpy array

shape()

Return Numpy-like shape of Domain object (ySize, xSize)

Returns **shape** – Numpy-like shape of Domain object (ySize, xSize)

Return type tuple of two INT

reproject_gcps(*srs_string=''*)

Reproject all GCPs to a new spatial reference system

Necessary before warping an image if the given GCPs are in a coordinate system which has a singularity in (or near) the destination area (e.g. poles for lonlat GCPs)

Parameters **srs_string** (*string*) – SRS given as Proj4 string. If empty '+proj=stere' is used

Notes

Reprojects all GCPs to new SRS and updates GCPProjection

class nansat.**Nansat**(*filename=''*, *mapper=''*, *log_level=30*, ***kwargs*)

Bases: *nansat.domain.Domain*, *nansat.exporter.Exporter*

Container for geospatial data. Performs all high-level operations.

n = Nansat(filename) opens the file with satellite or model data for reading, adds scientific metadata to bands, and prepares the data for further handling.

Parameters

- **filename** (*str*) – uri of the input file or OpeNDAP datastream
- **mapper** (*str*) – name of the mapper from nansat/mappers dir. E.g. 'sentinel1_l1', 'asar', 'hirlam', 'meris_l1', 'meris_l2', etc.
- **log_level** (*int*) – Level of logging. See: http://docs.python.org/howto/logging.html
- **kwargs** (*additional arguments for mappers*) –

Examples

```
>>> n1 = Nansat(filename)
>>> n2 = Nansat(sentinel1_filename, mapper='sentinel1_l1')
>>> array1 = n1[1]
>>> array2 = n2['sigma0_HV']
```

Notes

The instance of Nansat class (the object <n>) contains information about geographical reference of the data (e.g raster size, pixel resolution, type of projection, etc) and about bands with values of geophysical variables (e.g. water leaving radiance, normalized radar cross section, chlrophyll concentraion, etc). The object <n> has methods for high-level operations with data. E.g.: * reading data from file (Nansat.__getitem__); * visualization (Nansat.write_figure); * changing geographical reference (Nansat.reproject); * exporting (Nansat.export) * and much more...

Nansat inherits from Domain (container of geo-reference information) Nansat uses instance of VRT (wraper around GDAL VRT-files) Nansat uses instance of Figure (collection of methods for visualization)

FILL_VALUE = 9.96921e+36

ALT_FILL_VALUE = -10000.0

logger = None

filename = None

name = None

path = None

mapper = None

classmethod from_domain(*domain*, *array=None*, *parameters=None*, *log_level=30*)

Create Nansat object from input Domain [and array with data]

Parameters

- **domain** (Domain) – Defines spatial reference system and geographical extent.
- **array** (*numpy NDarray*) – Data for the first band. Shape must correspond to shape of <domain>
- **parameters** (*dict*) – Metadata for the first band. May contain 'name', 'wkv' and other keys.
- **log_level** (*int*) – Level of logging.

vrt = None

add_band(*array*, *parameters=None*, *nomem=False*)

Add band from numpy array with metadata.

Create VRT object which contains VRT and RAW binary file and append it to self.vrt.band_vrts

Parameters

- **array** (*ndarray*) – new band data. Shape should be equal to shape
- **parameters** (*dict*) – band metadata: wkv, name, etc. (or for several bands)
- **nomem** (*bool*) – saves the vrt to a tempfile on disk?

Notes

Creates VRT object with VRT-file and RAW-file. Adds band to the self.vrt.

Examples

```
>>> n.add_band(array, {'name': 'new_data'}) # add new band and metadata, keep␣
↪in memory
>>> n.add_band(array, nomem=True) # add new band, keep on disk
```

add_bands(*arrays*, *parameters=None*, *nomem=False*)

Add bands from numpy arrays with metadata.

Create VRT object which contains VRT and RAW binary file and append it to self.vrt.band_vrts

Parameters

- **arrays** (*list of ndarrays*) – new band data. Shape should be equal to shape
- **parameters** (*list of dict*) – band metadata: wkv, name, etc. (or for several bands)
- **nomem** (*bool*) – saves the vrt to a tempfile on disk?

Notes

Creates VRT object with VRT-file and RAW-file. Adds band to the self.vrt.

Examples

```
>>> n.add_bands([array1, array2]) # add new bands, keep in memory
```

bands()

Make a dictionary with all metadata from all bands

Returns **b** – key = N, value = dict with all band metadata

Return type dictionary

has_band(*band*)

Check if self has band with name <band> :param band: name or standard_name of the band to check :type band: str

Return type True/False if band exists or not

resize(*factor=None*, *width=None*, *height=None*, *pixelsize=None*, *resample_alg=- 1*)

Proportional resize of the dataset.

The dataset is resized as (x_size*factor, y_size*factor) If desired width, height or pixelsize is specified, the scaling factor is calculated accordingly. If GCPs are given in a dataset, they are also rewritten.

Parameters

- **factor** (*float, optional, default=1*) – Scaling factor for width and height
  - > 1 means increasing domain size
  - < 1 means decreasing domain size
- **width** (*int, optional*) – Desired new width in pixels

- **height** (*int, optional*) – Desired new height in pixels
- **pixelsize** (*float, optional*) – Desired new pixelsize in meters (approximate). A factor is calculated from ratio of the current pixelsize to the desired pixelsize.
- **resample_alg** (*int (GDALResampleAlg), optional*) –
  - -1 : Average (default),
  - 0 : NearestNeighbour
  - 1 : Bilinear,
  - 2 : Cubic,
  - 3 : CubicSpline,
  - 4 : Lancoz

Notes

self.vrt.dataset [VRT dataset of VRT object] raster size are modified to downscaled size. If GCPs are given in the dataset, they are also overwritten.

get_GDALRasterBand(*band_id=1*)

Get a GDALRasterBand of a given Nansat object

If str is given find corresponding band number If int is given check if band with this number exists. Get a GDALRasterBand from vrt.

Parameters **band_id** (*int or str*) –

- if int - a band number of the band to fetch
- if str band_id = {'name': band_id}

Return type GDAL RasterBand

Example

```
>>> b = n.get_GDALRasterBand(1)
>>> b = n.get_GDALRasterBand('sigma0')
```

list_bands(*do_print=True*)

Show band information of the given Nansat object

Show serial number, longName, name and all parameters for each band in the metadata of the given Nansat object.

Parameters **do_print** (*boolean*) – print on screen?

Returns **outString** – formatted string with bands info

Return type String

reproject(*dst_domain=None*, *resample_alg=0*, *block_size=None*, *tps=None*, *skip_gcps=1*, *addmask=True*, ***kwargs*)

Change projection of the object based on the given Domain

Create superVRT from self.vrt with AutoCreateWarpedVRT() using projection from the dst_domain. Modify XML content of the warped vrt using the Domain parameters. Generate warpedVRT and replace self.vrt

with warpedVRT. If current object spans from 0 to 360 and dst_domain is west of 0, the object is shifted by 180 westwards.

Parameters

- **dst_domain** (*domain*) – destination Domain where projection and resolution are set
- **resample_alg** (*int (GDALResampleAlg)*) –
  - 0 : NearestNeighbour
  - 1 : Bilinear
  - 2 : Cubic,
  - 3 : CubicSpline
  - 4 : Lancoz
- **block_size** (*int*) – size of blocks for resampling. Large value decrease speed but increase accuracy at the edge
- **tps** (*bool*) – Apply Thin Spline Transformation if source or destination has GCPs Usage of TPS can also be triggered by setting self.vrt.tps=True before calling to reproject. This options has priority over self.vrt.tps
- **skip_gcps** (*int*) – Using TPS can be very slow if the number of GCPs are large. If this parameter is given, only every [skip_gcp] GCP is used, improving calculation time at the cost of accuracy. If not given explicitly, 'skip_gcps' is fetched from the metadata of self, or from dst_domain (as set by mapper or user). [defaults to 1 if not specified, i.e. using all GCPs]
- **addmask** (*bool*) – If True, add band 'swathmask'. 1 - valid data, 0 no-data. This band is used to replace no-data values with np.nan

Notes

self.vrt : VRT object with dataset replaced to warpedVRT dataset Integer data is returnd by integer. Round off to decimal place. If you do not want to round off, convert the data types to GDT_Float32, GDT_Float64, or GDT_CFloat32.

See also:

http://www.gdal.org/gdalwarp.html

undo(*steps=1*)

Undo reproject, resize, add_band or crop of Nansat object

Restore the self.vrt from self.vrt.vrt

Parameters **steps** (*int*) – How many steps back to undo

Notes

Modifies self.vrt

watermask(*mod44path=None*, *dst_domain=None*, ***kwargs*)
Create numpy array with watermask (water=1, land=0)

250 meters resolution watermask from MODIS 44W Product: http://www.glcf.umd.edu/data/watermask/

Watermask is stored as tiles in TIF(LZW) format and a VRT file All files are stored in one directory. A tarball with compressed TIF and VRT files should be additionally downloaded from the Nansat documentation page: http://nansat.readthedocs.io/en/latest/source/features.html#differentiating-between-land-and-water

The method : Gets the directory either from input parameter or from environment variable MOD44WPATH Open Nansat object from the VRT file Reprojects the watermask onto the current object using reproject() or reproject_on_jcps() Returns the reprojected Nansat object

Parameters

- **mod44path** (*string*) – path with MOD44W Products and a VRT file
- **dst_domain** (Domain) – destination domain other than self
- **tps** (*Bool*) – Use Thin Spline Transformation in reprojection of watermask? See also Nansat.reproject()
- **skip_gcps** (*int*) – Factor to reduce the number of GCPs by and increase speed See also Nansat.reproject()

Returns watermask

Return type Nansat object with water mask in current projection

See also:

http://www.glcf.umd.edu/data/watermask/

http://nansat.readthedocs.io/en/latest/source/features.html#differentiating-between-land-and-water

write_figure(*filename=''*, *bands=1*, *clim=None*, *addDate=False*, *array_modfunc=None*, ***kwargs*)
Save a raster band to a figure in graphical format.

Get numpy array from the band(s) and band information specified either by given band number or band id. – If three bands are given, merge them and create PIL image. – If one band is given, create indexed image Create Figure object and: Adjust the array brightness and contrast using the given min/max or histogram. Apply logarithmic scaling of color tone. Generate and append legend. Save the PIL output image in PNG or any other graphical format. If the filename extension is 'tif', the figure file is converted to GeoTiff

Parameters

- **filename** (*str*) – Output file name. if one of extensions 'png', 'PNG', 'tif', 'TIF', 'bmp', 'BMP', 'jpg', 'JPG', 'jpeg', 'JPEG' is included, specified file is created. otherwise, 'png' file is created.
- **bands** (*integer or string or list (elements are integer or string),*) – default = 1 the size of the list has to be 1 or 3. if the size is 3, RGB image is created based on the three bands. Then the first element is Red, the second is Green, and the third is Blue.
- **clim** (*list with two elements or 'hist' to specify range of colormap*) – None (default) : min/max values are fetched from WKV, fallback-'hist' [min, max] : min and max are numbers, or [[min, min, min], [max, max, max]]: three bands used 'hist' : a histogram is used to calculate min and max values

- **addDate** (*boolean*) – False (default) : no date will be aded to the caption True : the first time of the object will be added to the caption
- **array_modfunc** (*None*) – None (default) : figure created using array in provided band function : figure created using array modified by provided function
- ****kwargs** (*parameters for Figure().*) –

Notes

if filename is specified, creates image file

Returns **Figure** – filename extension define format (default format is png)

Return type Figure object

Example

```
>>> n.write_figure('test.jpg') # write indexed image
>>> n.write_figure('test_rgb_hist.jpg', clim='hist', bands=[1, 2, 3]) # RGB␣
↪image
>>> n.write_figure('r09_log3_leg.jpg', logarithm=True, legend=True,
                   gamma=3, titleString='Title', fontSize=30,
                   numOfTicks=15) # add legend
>>> n.write_figure(filename='transparent.png', bands=[3],
                   mask_array=wmArray,
                   mask_lut={0: [0,0,0]},
                   clim=[0,0.15], cmapName='gray',
                   transparency=[0,0,0]) # write transparent image
```

See also:

Figure()

http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

write_geotiffimage(*filename*, *band_id=1*)

Writes an 8-bit GeoTiff image for a given band.

The colormap is fetched from the metadata item 'colormap'. Fallback colormap is 'gray'.

Color limits are fetched from the metadata item 'minmax'. If 'minmax' is not specified, min and max of the raster data is used.

The method can be replaced by using nansat.write_figure(). However, write_figure uses PIL, which does not allow Tiff compression. This gives much larger files.

Parameters

- **filename** (*str*) –
- **band_id** (*int or str*) –

property time_coverage_start

property time_coverage_end

get_metadata(*key=None*, *band_id=None*, *unescape=True*)

Get metadata from self.vrt.dataset

Parameters

- **key** (*str*) – name of the metadata key. If not givem all metadata is returned
- **band_id** (*int or str*) – number or name of band to get metadata from. If not given, global metadata is returned
- **unescape** (*bool*) – Replace '"', '&', '<' and '>' with these symbols " & < > ?

Returns

- **metadata** (*str*) – string with metadata if key is given and found
- **metadata** (*dict*) – dictionary with all metadata if key is not given

Raises **ValueError, if key is not found** –

set_metadata(*key=''*, *value=''*, *band_id=None*)
Set metadata to self.vrt.dataset

Parameters

- **key** (*string or dictionary with strings*) – name of the metadata, or dictionary with metadata names, values
- **value** (*string*) – value of metadata
- **band_id** (*int or str*) – number or name of band Without : global metadata is set

Notes

self.vrt.dataset : sets metadata in GDAL current dataset

get_band_number(*band_id*)
Return absolute band number

Check if given band_id is valid Return absolute number of the band in the VRT

Parameters **band_id** (*int or str or dict*) –

- if int : checks if such band exists and returns band_id
- if str : finds band with coresponding name
- if dict : finds first band with given metadata

Returns **absolute band number**

Return type int

get_transect(*points*, *bands*, *lonlat=True*, *smooth_radius=0*, *smooth_function=numpy.nanmedian*, *data=None*, *cornersonly=False*)
Get values from transect from given vector of poins

Parameters

- **points** (*2xN list or array, N (number of points) >= 1*) – coordinates [[x1, x2, y2], [y1, y2, y3]]
- **bands** (*list of int or string*) – elements of the list are band number or band Name
- **lonlat** (*bool*) – If the points in lat/lon, then True. If the points in pixel/line, then False.
- **smooth_radius** (*int*) – If smootRadius is greater than 0, smooth every transect pixel as the median or mean value in a circule with radius equal to the given number.
- **smooth_function** (*func*) – function for averaging values collected within smooth radius

- **data** (*ndarray*) – alternative array with data to take values from

Returns **transect**

Return type numpy record array

digitize_points(*band=1*, ***kwargs*)
Get coordinates of interactively digitized points

Parameters

- **band** (*int or str*) – ID of Nansat band
- ****kwargs** (*keyword arguments for imshow*) –

Returns **points** – list of 2xN arrays of points to be used in Nansat.get_transect()

Return type list

crop_interactive(*band=1*, *maxwidth=1000*, ***kwargs*)
Interactively select boundary and crop Nansat object

Parameters

- **band** (*int or str*) – id of the band to show for interactive selection of boundaries
- **maxwidth** (*int*) – large input data is downscaled to <maxwidth>
- ****kwargs** (*keyword arguments for imshow*) –

Notes

self.vrt [VRT] superVRT is created with modified SrcRect and DstRect

Returns **extent** – x_offset - X offset in the original dataset y_offset - Y offset in the original dataset x_size - width of the new dataset y_size - height of the new dataset

Return type (x_offset, y_offset, x_size, y_size)

Examples

```
>>> extent = n.crop_interactive(band=1) # crop a subimage interactively
```

crop_lonlat(*lonlim*, *latlim*)
Crop Nansat object to fit into given longitude/latitude limit

Parameters

- **lonlim** (*list of 2 float*) – min/max of longitude
- **latlim** (*list of 2 float*) – min/max of latitude

Notes

self.vrt [VRT] crops vrt to size that corresponds to lon/lat limits

Returns **extent** – x_offset - X offset in the original dataset y_offset - Y offset in the original dataset x_size - width of the new dataset y_size - height of the new dataset

Return type (x_offset, y_offset, x_size, y_size)

Examples

```
>>> extent = n.crop(lonlim=[-10,10], latlim=[-20,20]) # crop for given lon/lat␣
↪limits
```

crop(*x_offset*, *y_offset*, *x_size*, *y_size*, *allow_larger=False*)
Crop Nansat object

Create superVRT, modify the Source Rectangle (SrcRect) and Destination Rectangle (DstRect) tags in the VRT file for each band in order to take only part of the original image, create new GCPs or new GeoTransform for the cropped object.

Parameters

- **x_offset** (*int*) – pixel offset of subimage
- **y_offset** (*int*) – line offset of subimage
- **x_size** (*int*) – width in pixels of subimage
- **y_size** (*int*) – height in pizels of subimage
- **allow_larger** (*bool*) – Allow resulting extent to be larger than the original image?

Notes

self.vrt : super-VRT is created with modified SrcRect and DstRect

Returns **extent** – x_offset - X offset in the original dataset y_offset - Y offset in the original dataset x_size - width of the new dataset y_size - height of the new dataset

Return type (x_offset, y_offset, x_size, y_size)

Examples

```
>>> extent = n.crop(10, 20, 100, 200)
```

extend(*left=0*, *right=0*, *top=0*, *bottom=0*)
Extend domain from four sides

Parameters

- **left** (*int*) – number of pixels to add from left side
- **right** (*int*) – number of pixels to add from right side
- **top** (*int*) – number of pixels to add from top side
- **bottom** (*int*) – number of pixels to add from bottom side

Notes

Canges self.vrt by adding nexgative offset or setting size to be large that original size.

class nansat.**Figure**(*nparray*, ***kwargs*)

Bases: object

Perform operations with graphical files: create, append legend, save.

Figure instance is created in the Nansat.write_figure method. The methods below are applied consequently in order to generate a figure from one or three bands, estimate min/max, apply logarithmic scaling, convert to uint8, append legend, save to a file

Modifies: self.sizeX, self.sizeY (int), width and height of the image

Modifies: self.pilImg (PIL image), figure

Modifies: self.pilImgLegend (PIL image)

Note: If pilImgLegend is None, legend is not added to the figure. If it is replaced, pilImgLegend includes text string, color-bar, longName and units.

Parameters

- **array** (*numpy array (2D or 3D)*) – dataset from Nansat
- **cmin** (*number (int ot float) or [number, number, number]*) – 0, minimum value of varibale in the matrix to be shown
- **cmax** (*number (int ot float) or [number, number, number]*) – 1, minimum value of varibale in the matrix to be shown
- **gamma** (*float, >0*) – 2, coefficient for tone curve udjustment
- **subsetArraySize** (*int*) – 100000, size of the subset array which is used to get histogram
- **numOfColor** (*int*) – 250, number of colors for use of the palette. 254th is black and 255th is white.
- **cmapName** (*string*) – 'jet', name of Matplotlib colormaps see –> http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps
- **ratio** (*float, [0 1]*) – 1.0, ratio of pixels which are used to write the figure
- **numOfTicks** (*int*) – 5, number of ticks on a colorbar
- **titleString** (*string*) – '', title of legend (1st line)
- **caption** (*string*) – '', caption of the legend (2nd line, e.g. long name and units)
- **fontRatio** (*positive float*) – 1, factor for changing the fontSize.
- **fontSize** (*int*) – 12, size of the font of title, caption and ticks. If not given, fontSize is calculated using fontRatio: fontSize = height / 45 * fontRatio. fontSize has priority over fontRatio
- **logarithm** (*boolean, defult = False*) – If True, tone curve is used to convert pixel values. If False, linear.
- **legend** (*boolean, default = False*) – if True, information as textString, colorbar, longName and units are added in the figure.

- **mask_array** (*2D numpy array, int, the shape should be equal to*) – array.shape. If given, this array is used for masking land, clouds, etc on the output image. Value of the array are indices. LUT from mask_lut is used for coloring upon this indices.
- **mask_lut** (*dictionary*) – Look-Up-Table with colors for masking land, clouds etc. Used tgether with mask_array: {0, [0,0,0], 1, [100,100,100], 2: [150,150,150], 3: [0,0,255]} index
  - 0 - will have black color
  - 1 - dark gray
  - 2 - light gray
  - 3 - blue
- **logoFileName** (*string*) – name of the file with logo
- **logoLocation** (*list of two int, default = [0,0]*) – X and Y offset of the image If positive - offset is from left, upper edge If Negative - from right, lower edge Offset is calculated from the entire image legend inclusive
- **logoSize** (*list of two int*) – desired X,Y size of logo. If None - original size is used
- **latGrid** (*numpy array*) – full size array with latitudes. For adding lat/lon grid lines
- **lonGrid** (*numpy array*) – full size array with longitudes. For adding lat/lon grid lines
- **nGridLines** (*int*) – number of lat/lon grid lines to show
- **latlonLabels** (*int*) – number of lat/lon labels to show along each side.
- **transparency** (*int*) – transparency of the image background(mask), set for PIL alpha mask in Figure.save()
- **default** (*None*) –
- **LEGEND_HEIGHT** (*float, [0 1]*) – 0.1, legend height relative to image height
- **CBAR_HEIGHTMIN** (*int*) – 5, minimum colorbar height, pixels
- **CBAR_HEIGHT** (*float, [0 1]*) – 0.15, colorbar height relative to image height
- **CBAR_WIDTH** (*float [0 1]*) – 0.8, colorbar width relative to legend width
- **CBAR_LOCATION_X** (*float [0 1]*) – 0.1, colorbar offset X relative to legend width
- **CBAR_LOCATION_Y** (*float [0 1]*) – 0.5, colorbar offset Y relative to legend height
- **CBTICK_LOC_ADJUST_X** (*int*) – 5, colorbar tick label offset X, pixels
- **CBTICK_LOC_ADJUST_Y** (*int*) – 3, colorbar tick label offset Y, pixels
- **CAPTION_LOCATION_X** (*float, [0 1]*) – 0.1, caption offset X relative to legend width
- **CAPTION_LOCATION_Y** (*float, [0 1]*) – 0.1, caption offset Y relative to legend height
- **TITLE_LOCATION_X** (*float, [0 1]*) – 0.1, title offset X relative to legend width
- **TITLE_LOCATION_Y** – 0.3, title offset Y relative to legend height
- **DEFAULT_EXTENSION** (*string*) – '.png'

```
cmin = [0.0]

cmax = [1.0]

gamma = 2.0

subsetArraySize = 100000
```

```
numOfColor = 250
cmapName = 'jet'
ratio = 1.0
numOfTicks = 5
titleString = ''
caption = ''
fontRatio = 1
fontSize = None
logarithm = False
legend = False
mask_array = None
mask_lut = None
logoFileName = None
logoLocation = [0, 0]
logoSize = None
latGrid = None
lonGrid = None
lonTicks = 5
latTicks = 5
transparency = None
LEGEND_HEIGHT = 0.1
CBAR_HEIGHTMIN = 5
CBAR_HEIGHT = 0.15
CBAR_WIDTH = 0.8
CBAR_LOCATION_X = 0.1
CBAR_LOCATION_Y = 0.5
CBTICK_LOC_ADJUST_X = 5
CBTICK_LOC_ADJUST_Y = 3
CAPTION_LOCATION_X = 0.1
CAPTION_LOCATION_Y = 0.25
TITLE_LOCATION_X = 0.1
TITLE_LOCATION_Y = 0.05
DEFAULT_EXTENSION = '.png'
palette = None
pilImg = None
pilImgLegend = None
```

extensionList = ['png', 'PNG', 'tif', 'TIF', 'bmp', 'BMP', 'jpg', 'JPG', 'jpeg', 'JPEG']

array = None

apply_logarithm(***kwargs*)
Apply a tone curve to the array

After the normalization of the values from 0 to 1, logarithm is applied Then the values are converted to the normal scale.

Modifies: self.array (numpy array)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

apply_mask(***kwargs*)
Apply mask for coloring land, clouds, etc

If mask_array and mask_lut are provided as input parameters. The pixels in self.array which have index equal to mask_lut key in mask_array will have color equal to mask_lut value.

Modifies: self.array (numpy array)

Note: apply_mask should be called only after convert_palettesize (i.e. to uint8 data)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

add_logo(***kwargs*)
Insert logo into the PIL image

Read logo from file as PIL. Resize to the given size. Pan using the given location. Paste into pilImg.

Modifies: self.pilImg (PIL image)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

add_latlon_grids(***kwargs*)
Add lat/lon grid lines into the PIL image

Compute step of the grid. Make matrices with binarized lat/lon. Find edge (make line). Convert to mask. Add mask to PIL

Modifies: self.pilImg (PIL image), added lat/lon grid lines

Parameters

- **latGrid** (*numpy array*) – array with values of latitudes
- **lonGrid** (*numpy array*) – array with values of longitudes
- **lonTicks** (*int or list*) – number of lines to draw or locations of gridlines
- **latTicks** (*int or list*) – number of lines to draw or locations of gridlines
- ****kwargs** (*dict*) – any of Figure parameters

add_latlon_labels(***kwargs*)
Add lat/lon labels along upper and left side

Compute step of lables. Get lat/lon for these labels from latGrid, lonGrid Print lables to PIL in white.

Modifies: self.pilImg (PIL image), added lat/lon labels

Parameters

- **latGrid** (*numpy array*) – array with values of latitudes
- **lonGrid** (*numpy array*) – array with values of longitudes
- **lonTicks** (*int or list*) – number of lines to draw or locations of gridlines
- **latTicks** (*int or list*) – number of lines to draw or locations of gridlines
- ****kwargs** (*dict*) – Any of Figure parameters

clim_from_histogram(***kwargs*)

Estimate min and max pixel values from histogram

if ratio=1.0, simply the minimum and maximum values are returned. if 0 < ratio < 1.0, get the histogram of the pixel values. Then get rid of (1.0-ratio)/2 from the both sides and return the minimum and maximum values.

Parameters ****kwargs** (*dict*) – Any of Figure parameters

Returns **clim** – minimum and maximum pixel values for each band

Return type numpy array 2D ((3x2) or (1x2))

clip(***kwargs*)

Convert self.array to values between cmin and cmax

if pixel value < cmin, replaced to cmin.

if pixel value > cmax, replaced to cmax.

Modifies: self.array (numpy array)

Modifies: self.cmin, self.cmax : allowed min/max values

Parameters ****kwargs** (*dict*) – Any of Figure parameters

convert_palettesize(***kwargs*)

Convert self.array to palette color size in uint8

Modifies: self.array (numpy array)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

create_legend(***kwargs*)

self.legend is replaced from None to PIL image

PIL image includes colorbar, caption, and titleString.

Modifies: self.legend (PIL image)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

create_pilImage(***kwargs*)

self.create_pilImage is replaced from None to PIL image

If three images are given, create a image with RGB mode. if self.pilImgLegend is not None, it is pasted.

If one image is given, create a image with P(palette) mode. if self.pilImgLegend is not None, self.array is extended before create the pilImag and then paste pilImgLegend onto it.

Modifies: self.pilImg (PIL image), PIL image with / without the legend

Modifies: self.array (replace to None)

Parameters ****kwargs** (*dict*) – Any of Figure parameters

process(***kwargs*)

Do all common operations for preparation of a figure for saving

1. Modify default values of parameters by the provided ones (if any)
2. Clip to min/max
3. Apply logarithm if required
4. Convert data to uint8
5. Create palette
6. Apply mask for colouring land, clouds, etc if required
7. Create legend if required
8. Create PIL image
9. Add logo if required

Modifies: self.d

Modifies: self.array

Modifies: self.palette

Modifies: self.pilImgLegend

Modifies: self.pilImg

Parameters **`**kwargs`** (*dict*) – Any of Figure parameters

`save`(*fileName*, ***kwargs*)

Save self.pilImg to a physical file

If given extension is JPG, convert the image mode from Palette to RGB.

Modifies: self.pilImg (None)

Parameters

- **`fileName`** (*string*) – name of outputfile
- **`**kwargs`** (*dict*) – Any of Figure parameters

CHAPTER

FOUR

PLEASE ACKNOWLEDGE NANSAT

We appreciate acknowledgments of Nansat. If you use Nansat in scientific publications, please add a reference to the following paper:

Korosov A.A., Hansen M.W., Dagestad K.-F., Yamakawa A., Vines A., Riechert M., (2016). Nansat: a Scientist-Orientated Python Package for Geospatial Data Processing. Journal of Open Research Software. 4(1), p.e39. DOI: http://doi.org/10.5334/jors.120

CHAPTER

FIVE

DIFFERENTIATING BETWEEN LAND AND WATER

To add simple land- or water-masks to your figures, you can use the watermask() method in the main Nansat class. Download the prepared MODIS 250M water-mask product from our server and add the path to the directory with this data to an environment variable named MOD44WPATH (e.g. `MOD44WPATH=/Data/sat/auxdata/mod44w`).

CHAPTER

SIX

DISTANCE TO THE NEAREST COAST

To get inforamtion about distace to the nearest coastline within the domain of interest, you can use `distance2coast` method from the `nanasat.toolbox`. Download NASA's Ocean Biology Processing Group **`0.1x0.1 degree Distance to the Nearest Coast product <https://oceancolor.gsfc.nasa.gov/docs/distfromcoast/GMT_intermediate_coast_distance_01d.zip`_** and add the path to the directory with this data to an environment variable named DIST2COAST (e.g. `DIST2COAST=/path/to/file`). For more inforamtion about the product wisit *NASA's Ocean Biology Processing Group <https://oceancolor.gsfc.nasa.gov/docs/distfromcoast/>*.

CHAPTER

SEVEN

DIGITAL ELEVATION MODELS (DEMS)

7.1 Global Multi-resolution Terrain Elevation Data 2010 (GMTED2010)

The GMTED2010 datasets are provided by the U.S. Geological Survey. We have prepared a GDAL vrt file that can be used together with mapper_topography.py to open the 30 arcseconds Digital Elevation Model (DEM) with Nansat. To use it, the vrt file must be downloaded from ftp://ftp.nersc.no/nansat/dem/gmted2010_30.vrt and stored in the same folder as the tif files of *mean elevation* available at https://edcintl.cr.usgs.gov/downloads/sciweb1/shared/topo/downloads/GMTED/Global_tiles_GMTED/300darcsec/mea/.

In case you want to use a different DEM, the procedure is as follows:

1. Download the relevant GDAL readable files to a local folder
2. Generate a vrt file using *gdalbuildvrt*, e.g.:

```
gdalbuildvrt gmted2010_30.vrt *.tif
```

3. Update *mapper_topography.py* to accept the new kind of file(s)

7.2 Global 30 Arc-Second Elevation (GTOPO30)

We have also created a vrt-file for the GTOPO30 dataset. This is available as ftp://ftp.nersc.no/nansat/dem/gtopo30.vrt. The vrt-file should be placed in the same folder as the .DEM files available at https://dds.cr.usgs.gov/srtm/version2_1/SRTM30/.

CHAPTER

EIGHT

NANSAT CONVENTIONS

8.1 Git branching and merging

We adopt the following system for branching and merging:

1. **master** branch: numbered releases of the code. Never edited. Merged from *develop* and *hot fix* branches (see notes on workflow below). Long living.
2. **develop** branch: rather stable version of code under development. Never edited. Merged from topic specific issue branches. Long living.
3. **issue<NNN>_<short-heading>** - issue specific branches (NNN = issue number). Main working area. Short living. Branched from, and merged back into develop.
4. **hotfix<NNN>_<short-heading>** - branches that are specific to a hotfix issue. Hotfixes are bugfixes on master that can not wait until next release.

Note:

1. Never edit code in the master or develop branch. Always make a new branch for your edits.
2. A new branch should be very specific to only one problem. It should be short living.
3. Commit often.
4. Branch often.
5. Branch only from master or from develop.
6. Create pull requests for your branches and **always** assign a reviewer to merge, delete the branch, and close the issue (this is easy in github)

8.2 How to report and handle new issues

If you discover a bug in Nansat or you would like to suggest improvements or new features to Nansat, the following procedure should be followed:

1. Check that no one else has reported the same issue at https://github.com/nansencenter/nansat/issues
2. If not, add a new issue at https://github.com/nansencenter/nansat/issues
3. If Nansat respository is not accessible for writing (non team members), fork Nansat and clone your own fork locally

4. Create an issue branch on your local system named **issue<NNN>_<short-heading>** where NNN is the issue number from GitHub. This will be the main (short living) working area. The issue branch should originate from develop.
5. Add tests to reproduce the bug or test the new functionality
6. Write the necessary code
7. Push the new issue branch to your own fork at GitHub
8. Create a pull request of the issue branch on your fork at GitHub. A member of our team will the review the code, merge, delete the branch, and close the issue (this is easy in github)

8.3 How to report and handle a hotfix

If a bug is relatively quick and easy to handle, we call it a hotfix. A hotfix is branched from master (by team members), and the following workflow applies:

1. Branch from master into the hotfix specific branch (**hotfix<NNN>_<short-heading>**)

a) Update the tests

b) Fix the bug

c) Increment micro version in `setup.py`

d) Commit to the hotfix branch

2. If needed rebase the hotfix branch on top of master:

a) Checkout the hotfix branch

b) Use `git rebase master`

c) Fix conflicts if any

d) Test the code

e) Push your hotfix branch to GitHub. Note: You have to use 'git push -f' in order to rewrite the history on github. Rebase will not cause problems if only one person works with the hotfix branch.

3. Go to https://github.com/nansencenter/nansat and add a pull request for the newly pushed hotfix branch and assign a reviewer
4. Let the reviewer do the following:

a) Wait for tests on Travis CI to pass

b) Check the code

c) Request changes or merge the pull request into master using 'Rebase and Merge' button in the online tool.

d) Delete the branch

e) Merge master into develop (`git checkout develop`; `git merge master`; test the code; push to GitHUB; check Travis CI status)

f) Close the related issue

Note: If you work on a project using Nansat, this project should use the master version of Nansat. Different situations may then occur:

1. You discover a **bug** that is easy to fix. This should be solved in a **hotfix** branch originating from master.

2. You discover a **bug** that is difficult to fix. This should be solved in an **issue** branch originating from develop.
3. You want to add **functionality** to Nansat. This should be solved in an **issue** branch originating from develop. When this is completed and merged back into develop, we may have a new release.
4. You want to add a mapper. This can be done by adding a package `nansat_mappers` to your project. When the mapper is completed, create an issue and an issue-branch in Nansat, and finally submit a pull request with suggestion of a reviewer. You can still use the mapper via your `nansat_mappers` package until the new mapper is implemented in a release version of Nansat.

8.4 General conventions

- Nansat coding style generally follows PEP-8 (General style guide) and PEP-257 (Docstrings)
- Max line length is set to 100 chars
- Every unit of code must be properly tested (see unit-test) and documented
- All class/function/method/variable names have to be explicit and should contain no more than 3 words
- Single quotes should be used consistently instead of double quotes (except for cases where quotes are required, and for docstrings)
- GNU v3 licence should be inserted in all files. Mappers should have a standard header like this:

```
# Name:         mapper_asar.py
# Purpose:      Mapper for Envisat/ASAR data
# Authors:      Asuka Yamakava, Anton Korosov
# Licence:      This file is part of NANSAT. You can redistribute it or modify
#               under the terms of GNU General Public License, v.3
#               http://www.gnu.org/licenses/gpl-3.0.html
#
# Additional mapper/format specific links and information
```

- Docstrings should follow the Numpy style , with an extensive example file here example.py
- Valid headers for functions and classes are, in the following order, 'Parameters', 'Attributes' (only class), 'Returns' (only function), 'Yields', 'Other parameters', 'Raises', 'Warns', 'Warnings', 'See also', 'Notes', 'References' and 'Examples'.

8.4.1 Example function with complete Docstring

```
def some_function(start = 0, stop, step = 1):
    """ Return evenly spaced values within a given interval.

    Values are generated within the half-open interval "[start, stop)"
    (in other words, the interval including 'start' but excluding 'stop').

    For integer arguments the function is equivalent to the Python built-in
    'range '_ function, but returns a ndarray rather than a list.

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    start : number, optional
        Start of interval.  The interval includes this value.  The default start value␣
↪is 0.
    stop : number
        End of interval.  The interval does not include this value.
    step : number, optional
        Spacing between values.  For any output 'out', this is the distance between two␣
↪adjacent values, "out[i+1] - out[i]". The default step size is 1. If 'step' is specified,
↪ 'start' must also be given.
    dtype : dtype
        The type of the output array. If 'dtype' is not given, infer the data type from␣
↪the other input arguments.

    Returns
    -------
    out : ndarray
        Array of evenly spaced values.

        For floating point arguments, the length of the result is "ceil((stop - start)/
↪step)". Because of floating point overflow, this rule may result in the last element␣
↪of 'out' being greater than 'stop'.

    See Also
    --------
    linspace : Evenly spaced numbers with careful handling of endpoints
    ogrid: Arrays of evenly spaced numbers in N-dimensions
    mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensions

    Examples
    --------
    >>> np.arange(3)
    array([0, 1, 2])

    """
```

8.5 Naming conventions

- when a variable points to the GDALDataset, GDALDriver, etc. its name must always contain word “dataset”, “driver”, etc. representatively (raw_dataset, src_dataset, example_driver)
- when a variable points to a string with name it should contain ‘name’ (band_name)
- when longitude and latitude are input to (or output from) a function, they should be given in this order: (lon, lat). These variables should always be named ‘lon’ and ‘lat’ (i.e. never ‘long’).
- source and destination are prefixed as ‘src’ and ‘dst’ (src_dataset, dst_raster_xsize)
- band numbers should be called ‘band_number’
- GDAL bands should be called ‘band’ or, e.g., ‘dst_band’ when prefixed (GDAL is actually in-consistent here: gdal.Dataset.!GetRasterBand returns a ‘Band’-object; hence ‘Band’ is the name of the class and the Python datatype)
- We use ‘filename’ (as in Python standard library)

8.6 Style checking

In your IDE/editor, it is highly recommended to activate/install a plugin for/script a save hook for doing automatic style checks and/or corrections, eg autopep8, pylint, pyflakes.

8.7 Tests

In general:

- Every function must be accompanied with a test suite
- Tests should be both positive (testing that the function work as intended with valid data) and negative (testing that the function behaves as expected with invalid data e.g. that correct exceptions are thrown)
- If a function has optional arguments, separate tests for all options should be created

8.7.1 Testing core Nansat functionality

- Tests for Nansat, Domain, etc should be added to nansat/tests/test_<module_name>.py file;
- These tests should be added as functions of classes inheriting from unittest.TestCase (e.g. DomainTest);
- Tests sharing similar set-up may inherit from the same class which has a setUp function;
- The core tests are run at Travis CI (continuous integration) which integrates with Coveralls for providing test coverage

8.7.2 Integration testing

Products read by Nansat mappers are tested in modules within the nansat_integration_tests folder in the repository root. These tests should have access to all the kinds of data read by nansat. Since this is a very large amount of data, and since we cannot share every data product openly, these tests are not presently executed at Travis CI. Every developer should add new end-to-end tests and execute them when new mappers or workflows are added. Unavailable test data will lead to fewer tests being executed, i.e. they won't fail because of missing data. If possible, datasets used in new tests should be made available to the Nansen Center such that we can run the full test suite.

8.7.3 Testing mappers

General tests checking that the mappers don't violate the functionality of nansat and checks that some specific metadata is added, are collected in the nansat_integration_tests.mapper_tests module.

Also, we aim to create proper unit tests that use mock object for all the mappers. This will help to significantly increase the test coverage.

8.7.4 Testing specific data products or workflows

In typical scientific workflows, a data product is opened with Nansat and some operations are performed, e.g., adding new derived bands and exporting the results to a netcdf, or creating figures. To make sure that new versions of nansat do not harm these workflows with bugs or sudden interface changes, we collect tests for typical workflows in separate modules within the nansat_integration_tests package, e.g. test_sar, test_radarsat2, etc. We encourage users and developers to add such tests to avoid such potential problems

8.7.5 Doctests

TODO: add information about how to use doctests

CHAPTER

NINE

ABOUT NANSAT MAPPERS

9.1 Concept

GDAL can read most satellite EO and NetCDF-CF compliant raster data relevant for Earth sciences. However, GDAL does not attach meaning to the contained information, i.e., it does not specify what kind of data is contained in a given band, e.g., if it is *water-leaving-radiance* used for monitoring of water quality. Nansat provides a mapping between geophysical variables of known meaning and the raster bands provided in the "Datasets" returned by GDAL.

The modules within the mappers package each contain a class defining the mapping between the bands returned from GDAL and metadata vocabularies provided via the py-thesaurus-interface package. For example, the simplest mapper for Meris level-1 data explicitly states that the first 15 bands are *upwelling_spectral_radiance* at 15 wavelengths and that the 16th band contains quality flags. The description of these bands require some compulsory metadata attributes to be defined in the mapper. These attributes follow certain given conventions:

- NetCDF-CF
- NASA GCMD keywords
- nersc-vocabularies

By using these conventions, the mappers thus attach unambiguous geophysical meaning to variables following the given standards. This allows the user to open and use a geo-refenced raster dataset with Nansat without depending on detailed apriori knowledge about the origin or type of the data.

9.2 Workflow

When we open a file with Nansat:

```
#!/usr/bin/env python
n = Nansat(filename)
```

these steps follow:

- The Nansat constructor calls gdal.Open(filename) to open the file with GDAL, and returns a GDAL Dataset with a list of available raster bands
- The Nansat constructor loops through available mappers and parses the Dataset to the mapper
  - Each mapper checks if the input Dataset is appropriate for the mapper, i.e., if the format, the metadata and the set of bands in the Dataset corresponds to what is expected in the mapper
    - If the Dataset is not valid, the mapper silently fails and the next mapper is tested
    - If the Dataset fits the mapper:

  - the mapper creates a GDAL VRT file with georeference and raster bands corresponding to the "well known variables" in nersc-vocabularies and adds respective metadata to each band (standard_name, units, etc).

- The mapper object, which is an instance of the VRT-class, is then returned to the Nansat instance as an attribute named `vrt` (`Nansat.vrt`)

The VRT has the following properties:

- we can use any available GDAL API functions, e.g., warping or exporting
- it contains georeferencing recognised by GDAL
- we can add PixelFunctions for, e.g., calculation of *speed* given two vector components of wind or current
- still it contains only Raster Bands with metadata which correspond to any of the NANSAT "Well Known Variables"

The Dataset may, e.g., be subsetted, reprojected, merged, etc., by simply modifying the VRT-file, either automatically by the GDAL high level applications/functions, or with NANSAT-specific Python logic. An important benefit of this approach is that we employ the *lazy processing concept* in GDAL.

Note: No processing or file reading/writing is performed before it is needed.

The VRT file defines a set of operations in xml format. When information is needed, data is extracted as numpy arrays for further processing or plotting. As such, we basically use the GDAL Datamodel and do not need to design our own.

9.3 Technical details

- The VRT-file is stored in memory using GDAL VSI-approach
- The VRT-class is a wrapper around the VRT-file. It has methods for generating, modifying, copying and other operations with VRT-files. VRT-class uses both GDAL methods and direct writing for modifying the VRT-file.
- Each mapper inherits the VRT-class.

9.4 Where to put new mappers?

If you have created a new mapper, you can either submit a pull request for inclusion in the nansat mappers package, or you can make a namespace package to let nansat discover your mapper automatically. This is done the following way:

1. Create a directory called *nansat_mappers* within a directory on your $PYTHONPATH
2. Inside *nansat_mappers*, create the file *__init__.py* with the following lines:

```
# __init__.py
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

3. Add your mapper module (the filename should start with *mapper_* and end with *.py*) to the *nansat_mappers* folder
4. Reload your shell and start Python
5. Nansat should now find you mapper.

Note that user defined mappers have higher priority than standard mappers.

9.5 Required metadata added in the mappers

- TODO: add list of required metadata

9.6 Adding mapper tests

TODO: add documentation about how to write mapper tests

CHAPTER

TEN

RELEASING NANSAT

10.1 General release procedure

In setup.py, Make sure the version is correct, by checking MAJOR, MINOR and MICRO, and that 'dev' is removed from VERSION.

Releases should be made from the master branch. **Make sure that Nansat works on all operating systems (Windows, Linux and OS X), with all combinations of python (py27, py35, py36) before releasing.**

When you are ready to release a new version, create a tag for the release and push it. The following creates a tag for version 0.6.17 and pushes it.

```
git tag "v0.6.17"
git push origin --tags
```

Go to https://github.com/nansencenter/nansat/releases and click "Draft a new release".

Select the tag version you just created, create a release title, for consistency, use the format of Nansat-0.6.17

Write some release notes that describes the changes done in this release.

10.2 Releasing on PiPy

First, wait until Nansat passes all tests on Travi-CI, Appveyor and Coverals. Then execute:

```
conda create -n release_nansat -c conda-forge -y python pythesint scipy basemap netcdf4␣
↪gdal pillow mock nose urllib3 twine
source activate release_nansat
python setup.py sdist
# Check the dist file that was just created
ls dist
# Should be a file on this format 'nansat-1.0.20.tar.gz'
twine upload dist/nansat-1.0.20.tar.gz
```

Packaging documentation is found at PyPA Packaging and Distributing Projects

To avoid having to enter password when uploading, you can set $HOME/.pypirc as described in the above link.

10.3 Releasing on Anaconda

We are releasing Nansat through the conda-forge channel on Anaconda. First, wait until Nansat passes all tests on Travi-CI, Appveyor and Coverals. Then execute:

```
# install (or update) conda-smithy
conda install -n root -c conda-forge conda-smithy
git clone git@github.com:conda-forge/nansat-feedstock.git
cd nansat-feedstock
conda smithy rerender -c auto
git push
```

Information how to use Conda-Smithy can be found at The tool for managing conda-forge feedstocks

10.4 Releasing on Docker

To build an image with stable version of Nansat you need to build the image and push it to Docker Hub:

:: # build the base image with conda docker build . -t nansat:conda –target conda # build the image for compiling Nansat docker build . -t nansat:dev –target dev # build the image for distributing Nansat docker build . -t nansat:latest –target latest # find the ide of the nansat:latest image (e.g. bb38976d03cf) docker images # tag the image (bb38976d03cf is the id of the nansat:latest image) docker tag bb38976d03cf akorosov/nansat:latest # authenticate on Docker Hub docker login –username=yourhubusername –email=youremail@company.com # push the image to Docker Hub docker push akorosov/nansat:latest

CHAPTER

ELEVEN

DOCUMENTING NANSAT

Documentation should follow the conventions.

Note: Documentation for classes should be given after the class definition, not within the `__init__`-method.

To build documentation locally, the best is to create a virtual environment with the sphinx environment installed. This is done as follows:

```
cd docs
conda env create -n build_docs --file environment.yml
source activate build_docs
```

Then, the following commands should build the documentation:

```
make clean
sphinx-apidoc -fo source/ ../nansat
make html
```

Some documentation remains to be written. This is marked by `TODO` in the rst source files. Find open tasks by:

```
cd docs/source
grep TODO *
```

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y