PyEPR Documentation

Release 0.9.5.dev0

Antonio Valentino

Aug 23, 2018

Contents

List of Figures

HomePage http://avalentino.github.io/pyepr

Author Antonio Valentino

Contact antonio.valentino@tiscali.it

Version 0.9.5.dev0

CHAPTER 1

User Manual

1.1 Quick start

PyEPR provides Python bindings for the ENVISAT Product Reader C API (EPR API) for reading satellite data from ENVISAT ESA (European Space Agency) mission.

PyEPR, as well as the EPR API for C, supports ENVISAT MERIS, AATSR Level 1B and Level 2 and also ASAR data products. It provides access to the data either on a geophysical (decoded, ready-to-use pixel samples) or on a raw data layer. The raw data access makes it possible to read any data field contained in a product file.

Full access to the Python EPR API is provided by the *epr* module that have to be imported by the client program e.g. as follows:

```
import epr
```

The following snippet open an ASAR product and dumps the "Main Processing Parameters" record to the standard output:

```
import epr

product = epr.Product(
    'ASA_IMP_1PNUPA20060202_062233_000000152044_00435_20529_3110.N1
↪')
dataset = product.get_dataset('MAIN_PROCESSING_PARAMS_ADS')
record = dataset.read_record(0)
print(record)
product.close()
```

Since version 0.9 PyEPR also include *update* features that are not available in the EPR C API. The user can open a product in update mode ('rb+') and call the *epr.Field.set_elem()*

and *epr.Field.set_elems()* methods of *epr.Field* class to update its elements and write changes to disk.

See also:

Update support and *Update Field elements* tutorial for details.

1.2 Requirements

In order to use PyEPR it is needed that the following software are correctly installed and configured:

- Python2 >= 2.6 or Python3 >= 3.1 (including PyPy)
- numpy >= 1.5.0
- EPR API >= 2.2 (optional, since PyEPR 0.7 the source tar-ball comes with a copy of the EPR C API sources)
- a reasonably updated C compiler[1] (build only)
- Cython >= 0.19[2] (optional and build only)
- unittest2 (only required for Python < 3.4)

1.3 Download

Official source tar-balls can be downloaded form PyPi:

https://pypi.org/project/pyepr

The source code of the development versions is available on the GitHub project page

https://github.com/avalentino/pyepr

To clone the git repository the following command can be used:

```
$ git clone https://github.com/avalentino/pyepr.git
```

1.4 Installation

The easier way to install PyEPR is using tools like pip:

[1] PyEPR has been developed and tested with gcc 4.

[2] The source tarball of official releases also includes the C extension code generated by cython so users don't strictly need cython to install PyEPR.
It is only needed to re-generate the C extension code (e.g. if one wants to build a development version of PyEPR).

```
$ pip install pyepr
```

For a user specific installation please use:

```
$ pip install --user pyepr
```

To install PyEPR in a non-standard path:

```
$ pip install --install-option="--prefix=<TARGET_PATH>" pyepr
```

just make sure that `<TARGET_PATH>/lib/pythonX.Y/site-packages` is in the `PYTHONPATH`.

PyEPR can be installed from sources using the following command:

```
$ python setup.py install
```

The `setup.py` script by default checks for the availability of the EPR C API source code in the `<package-root>/epr-api-src` directory and tries to build PyEPR in *standalone mode*, i.e. without linking an external dynamic library of EPR-API.

If no EPR C API sources are found then the `setup.py` script automatically tries to link the EPR-API dynamic library. This can happen, for example, if the user is using a copy of the PyEPR sources cloned from a git repository. In this case it is assumed that the EPR API C library is properly installed in the system (see the *Requirements* section).

It is possible to control which EPR API C sources to use by means of the *–epr-api-src* option of the `setup.py` script:

```
$ python setup.py install --epr-api-src=../epr-api/src
```

Also it is possible to switch off the *standalone mode* and force the link with the system EPR API C library:

```
$ python setup.py install --epr-api-src=None
```

1.5 Testing

PyEPR package comes with a complete test suite. The test suite can be run using the following command in the `tests` directory:

```
$ python test_all.py
```

or from the package root directory:

```
$ python setup.py test
```

The test script automatically downloads and decompresses the ENVISAT sample product necessary for testing, MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_0001.N1, if it is not already available in the `tests` directory.

Note: please note that, unless the user already have a copy of the specified sample product correctly installed, an **internet connection** is necessary the first time that the test suite is run.

After the first run the sample product remains in the `tests` directory so the internet access is no longer necessary.

1.6 Python vs C API

The Python EPR API is fully object oriented. The main structures of the C API have been implemented as objects while C function have been logically grouped and mapped onto object methods.

The entire process of defining an object oriented API for Python has been quite easy and straightforward thanks to the good design of the C API,

Of course there are also some differences that are illustrated in the following sections.

1.7 Memory management

Being Python a very high level language uses have never to worry about memory allocation/de-allocation. They simply have to instantiate objects:

```
product = epr.Product('filename.N1')
```

and use them freely.

Objects are automatically destroyed when there are no more references to them and memory is de-allocated automatically.

Even better, each object holds a reference to other objects it depends on so the user never have to worry about identifiers validity or about the correct order structures have to be freed.

For example: the C *EPR_DatasetId* structure has a field (*product_id*) that points to the *product* descriptor *EPR_productId* to which it belongs to.

The reference to the parent product is used, for example, when one wants to read a record using the *epr_read_record* function:

```
EPR_SRecord* epr_read_record(EPR_SDatasetId* dataset_id, ...);
```

The function takes a *EPR_SDatasetId* as a parameter and assumes all fields (including `dataset->product_id`) are valid. It is responsibility of the programmer to keep all structures valid and free them at the right moment and in the correct order.

This is the standard way to go in C but not in Python.

In Python all is by far simpler, and the user can get a *dateset* object instance:

```
dataset = product.get_dataset('MAIN_PROCESSING_PARAMS_ADS')
```

and then forget about the *product* instance it depends on. Even if the *product* variable goes out of scope and it is no more directly accessible in the program the *dataset* object keeps staying valid since it holds an internal reference to the *product* instance it depends on.

When *record* is destroyed automatically also the parent *epr.Product* object is destroyed (assumed there is no other reference to it).

The entire machinery is completely automatic and transparent to the user.

Note: of course when a *product* object is explicitly closed using the *epr.Product.close()* any I/O operation on it and on other objects (bands, datasets, etc) associated to it is no more possible.

1.8 Arrays

PyEPR uses numpy in order to manage efficiently the potentially large amount of data contained in ENVISAT products.

- *epr.Field.get_elems()* return an 1D array containing elements of the field
- the *Raster.data* property is a 2D array exposes data contained in the *epr.Raster* object in form of `numpy.ndarray`

  Note: *epr.Raster.data* directly exposes *epr.Raster* i.e. shares the same memory buffer with *epr.Raster*:

  ```
  >>> raster.get_pixel(i, j)
  5
  >>> raster.data[i, j]
  5
  >>> raster.data[i, j] = 3
  >>> raster.get_pixel(i, j)
  3
  ```

- *epr.Band.read_as_array()* is an additional method provided by the Python EPR API (does not exist any correspondent function in the C API). It is mainly a facility method that allows users to get access to band data without creating an intermediate *epr.Raster* object. It read a slice of data from the *epr.Band* and returns it as a 2D `numpy.ndarray`.

1.9 Enumerators

Python does not have *enumerators* at language level (at least this is true for Python < 3.4). Enumerations are simply mapped as module constants that have the same name of the C enumerate but are spelled all in capital letters.

For example:

C	Pythn
e_tid_double	E_TID_DOUBLE
e_smod_1OF1	E_SMOD_1OF1
e_smid_log	E_SMID_LOG

1.10 Error handling and logging

Currently error handling and logging functions of the EPR C API are not exposed to python.

Internal library logging is completely silenced and errors are converted to Python exceptions. Where appropriate standard Python exception types are use in other cases custom exception types (e.g. *epr.EPRError*, *epr.EPRValueError*) are used.

1.11 Library initialization

Differently from the C API library initialization is not needed: it is performed internally the first time the *epr* module is imported in Python.

1.12 High level API

PyEPR provides some utility method that has no correspondent in the C API:

- *epr.Record.fields()*
- *epr.Record.get_field_names()*
- *epr.Dataset.records()*
- *epr.Product.get_dataset_names()*
- *epr.Product.get_band_names()*
- *epr.Product.datasets()*
- *epr.Product.bands()*

Example:

```
for dataset in product.datasets():
    for record in dataset.records():
        print(record)
        print()
```

Another example:

```
if 'proc_data' in product.band_names():
    band = product.get_band('proc_data')
    print(band)
```

1.13 Special methods

The Python EPR API also implements some special method in order to make EPR programming even handy and, in short, pythonic.

The `__repr__` methods have been overridden to provide a little more information with respect to the standard implementation.

In some cases `__str__` method have been overridden to output a verbose string representation of the objects and their contents.

If the EPR object has a `print_` method (like e.g. *epr.Record.print_()* and *epr.Field.print_()*) then the string representation of the object will have the same format used by the `print_` method. So writing:

```
fd.write(str(record))
```

giver the same result of:

```
record.print_(fd)
```

Of course the *epr.Record.print_()* method is more efficient for writing to file.

Also *epr.Dataset* and *epr.Record* classes implement the `__iter__` special method for iterating over records and fields respectively. So it is possible to write code like the following:

```
for record in dataset:
    for index, field in enumerate(record):
        print(index, field)
```

epr.DSD and *epr.Field* classes implement the `__eq__` and `__ne__` methods for objects comparison:

```
if filed1 == field2:
    print('field 1 and field2 are equal')
    print(field1)
else:
```

(continues on next page)

(continued from previous page)

```
    print('field1:', field1)
    print('field2:', field2)
```

epr.Field object also implement the __len__ special method that returns the number of elements in the field:

```
if field.get_type() != epr.E_TID_STRING:
    assert field.get_num_elems() == len(field)
else:
    assert len(field) == len(field.get_elem())
```

Note: differently from the *epr.Field.get_num_elems()* method `len(field)` return the number of elements if the field type is not *epr.E_TID_STRING*. If the field contains a string then the string length is returned.

Finally the *epr.Product* class acts as a context manager (i.e. it implements the __enter__ and __exit__ methods).

This allows the user to write code like the following:

```
with epr.open('ASA_IMS_ ... _4650.N1') as product:
    print(product)
```

that ensure that the product is closed as soon as the program exits the `with` block.

1.14 Update support

It is not possible to create new ENVISAT products for scratch with the EPR API. Indeed EPR means "**E**NVISAT **P**roduct **R**eaeder". Anyway, since version 0.9, PyEPR also include basic *update* features. This means that, while it is still not possible to create new `Products`, the user can *update* existing ones changing the contents of any `Field` in any record with the only exception of MPH and SPH `Fields`.

The user can open a product in update mode ('rb+'):

```
product = epr.open('ASA_IMS_ ... _4650.N1', 'rb+')
```

and update the *epr.Field* element at a specific index:

```
field.set_elem(new_value, index)
```

or also update all elements ol the *epr.Field* in one shot:

```
field.set_elems(new_values)
```

Note: unfortunately there are some limitations to the update support. Many of the internal structures of the EPR C API are loaded when the `Product` is opened and are not automatically updated when the *epr.Field.set_elem()* and *epr.Field.set_elems()* methods are called. In particular *epr.Band*s contents may depend on several *epr.Field* values, e.g. the contents of *Scaling_Factor_GADS* *epr.Dataset*. For this reason the user may need to close and re-open the *epr.Product* in order to have all changes effectively applied.

See also:

Update Field elements

CHAPTER 2

Tutorials

2.1 Interactive use of PyEPR

In this tutorial it is showed an example of how to use PyEPR interactively to open, browse and display data of an ENVISAT ASAR product.

For the interactive session it is used the Jupyter console started with the *–pylab* option to enable the interactive plotting features provided by the matplotlib package.

The ASAR product used in this example is a free sample available at the ESA web site.

2.1.1 `epr` module and classes

After starting the jupyter console with the following command:

```
$ jupyter console -- --pylab
```

one can import the *epr* module and start start taking confidence with available classes and functions:

```
Jupyter console 5.2.0

Python 3.6.5 (default, Apr  1 2018, 05:46:30)
Type "copyright", "credits" or "license" for more information.

IPython 5.5.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
```

(continues on next page)

(continued from previous page)

```
object?   -> Details about 'object', use 'object??' for extra␣
↪details.

In [1]: import epr

In [2]: epr?

Base Class:        <type 'module'>
String Form:       <module 'epr' from 'epr.so'>
Namespace:         Interactive
File:              /home/antonio/projects/pyepr/epr.so
Docstring:
    Python bindings for ENVISAT Product Reader C API

    PyEPR_ provides Python_ bindings for the ENVISAT Product Reader␣
↪C API
    (`EPR API`_) for reading satellite data from ENVISAT_ ESA_␣
↪(European
    Space Agency) mission.

    PyEPR_ is fully object oriented and, as well as the `EPR API`_␣
↪for C,
    supports ENVISAT_ MERIS, AATSR Level 1B and Level 2 and also␣
↪ASAR data
    products. It provides access to the data either on a geophysical
    (decoded, ready-to-use pixel samples) or on a raw data layer.
    The raw data access makes it possible to read any data field␣
↪contained
    in a product file.

    .. _PyEPR: http://avalentino.github.io/pyepr
    .. _Python: https://www.python.org
    .. _`EPR API`: https://github.com/bcdev/epr-api
    .. _ENVISAT: https://envisat.esa.int
    .. _ESA: https://earth.esa.int

In [3]: epr.__version__, epr.EPR_C_API_VERSION
Out[3]: ('0.9.1', '2.3dev')
```

Docstrings are available for almost all classes, methods and functions in the *epr* and they can be displayed using the `help()` python command or the `?` Jupyter shortcut as showed above.

Also Jupyter provides a handy tab completion mechanism to automatically complete commands or to display available functions and classes:

```
In [4]: product = epr. [TAB]
epr.Band                        epr.E_TID_STRING
epr.DSD                         epr.E_TID_TIME
epr.Dataset                     epr.E_TID_UCHAR
```

(continues on next page)

(continued from previous page)

```
epr.EPRError                    epr.E_TID_UINT
epr.EPRTime                     epr.E_TID_UNKNOWN
epr.EPRValueError               epr.E_TID_USHORT
epr.EPR_C_API_VERSION           epr.EprObject
epr.E_SMID_LIN                  epr.Field
epr.E_SMID_LOG                  epr.Product
epr.E_SMID_NON                  epr.Raster
epr.E_SMOD_1OF1                 epr.Record
epr.E_SMOD_1OF2                 epr.create_bitmask_raster
epr.E_SMOD_2OF2                 epr.create_raster
epr.E_SMOD_2TOF                 epr.data_type_id_to_str
epr.E_SMOD_3TOI                 epr.get_data_type_size
epr.E_TID_CHAR                  epr.get_sample_model_name
epr.E_TID_DOUBLE                epr.get_scaling_method_name
epr.E_TID_FLOAT                 epr.np
epr.E_TID_INT                   epr.open
epr.E_TID_SHORT                 epr.so
epr.E_TID_SPARE                 epr.sys
```

2.1.2 `epr.Product` navigation

The first thing to do is to use the *`epr.open()`* function to get an instance of the desired ENVISAT *`epr.Product`*:

```
In [4]: product = epr.open(\
'ASA_IMP_1PNUPA20060202_062233_000000152044_00435_20529_3110.N1')

In [4]: product.
product.bands                   product.get_mph
product.close                   product.get_num_bands
product.closed                  product.get_num_datasets
product.datasets                product.get_num_dsds
product.file_path               product.get_scene_height
product.get_band                product.get_scene_width
product.get_band_at             product.get_sph
product.get_band_names          product.id_string
product.get_dataset             product.meris_iodd_version
product.get_dataset_at          product.read_bitmask_raster
product.get_dataset_names       product.tot_size
product.get_dsd_at

In [5]: product.tot_size / 1024.**2
Out[5]: 132.01041889190674

In [6]: print(product)
epr.Product(ASA_IMP_1PNUPA20060202_ ...) 7 datasets, 5 bands
```

(continues on next page)

(continued from previous page)

```
epr.Dataset(MDS1_SQ_ADS) 1 records
epr.Dataset(MAIN_PROCESSING_PARAMS_ADS) 1 records
epr.Dataset(DOP_CENTROID_COEFFS_ADS) 1 records
epr.Dataset(SR_GR_ADS) 1 records
epr.Dataset(CHIRP_PARAMS_ADS) 1 records
epr.Dataset(GEOLOCATION_GRID_ADS) 11 records
epr.Dataset(MDS1) 8192 records

epr.Band(slant_range_time) of epr.Product(ASA_IMP_1PNUPA20060202_ ..
↪.)
epr.Band(incident_angle) of epr.Product(ASA_IMP_1PNUPA20060202_ ...)
epr.Band(latitude) of epr.Product(ASA_IMP_1PNUPA20060202 ...)
epr.Band(longitude) of epr.Product(ASA_IMP_1PNUPA20060202 ...)
epr.Band(proc_data) of epr.Product(ASA_IMP_1PNUPA20060202 ...)
```

A short summary of product contents can be displayed simply printing the *epr.Product* object as showed above. Being able to display contents of each object it is easy to keep browsing and get all desired information from the product:

```
In [7]: dataset = product.get_dataset('MAIN_PROCESSING_PARAMS_ADS')

In [8]: dataset
Out[8]: epr.Dataset(MAIN_PROCESSING_PARAMS_ADS) 1 records

In [9]: record = dataset.[TAB]
dataset.create_record    dataset.get_dsd_name     dataset.product
dataset.description      dataset.get_name         dataset.read_
↪record
dataset.get_dsd          dataset.get_num_records  dataset.records

In [9]: record = dataset.read_record(0)

In [10]: record
Out[10]: <epr.Record object at 0x33570f0> 220 fields

In [11]: record.get_field_names()[:20]
Out[11]:
['first_zero_doppler_time',
 'attach_flag',
 'last_zero_doppler_time',
 'work_order_id',
 'time_diff',
 'swath_id',
 'range_spacing',
 'azimuth_spacing',
 'line_time_interval',
 'num_output_lines',
 'num_samples_per_line',
 'data_type',
```

(continues on next page)

(continued from previous page)

```
 'spare_1',
 'data_analysis_flag',
 'ant_elev_corr_flag',
 'chirp_extract_flag',
 'srgr_flag',
 'dop_cen_flag',
 'dop_amb_flag',
 'range_spread_comp_flag']

In [12]: field = record.get_field('range_spacing')

In [13]: field.get [TAB]
field.get_description  field.get_name         field.get_unit
field.get_elem         field.get_num_elems
field.get_elems        field.get_type

In [13]: field.get_description()
Out[13]: 'Range sample spacing'

In [14]: epr.data_type_id_to_str(field.get_type())
Out[14]: 'float'

In [15]: field.get_num_elems()
Out[15]: 1

In [16]: field.get_unit()
Out[16]: 'm'

In [17]: print(field)
range_spacing = 12.500000
```

2.1.3 Iterating over `epr` objects

epr.Record objects are also iterable so one can write code like the following:

```
In [18]: for field in record:
             if field.get_num_elems() == 4:
                 print('%s: %d elements' % (field.get_name(),␣
↪len(field)))

        ....:
nominal_chirp.1.nom_chirp_amp: 4 elements
nominal_chirp.1.nom_chirp_phs: 4 elements
nominal_chirp.2.nom_chirp_amp: 4 elements
nominal_chirp.2.nom_chirp_phs: 4 elements
nominal_chirp.3.nom_chirp_amp: 4 elements
nominal_chirp.3.nom_chirp_phs: 4 elements
```

(continues on next page)

(continued from previous page)

```
nominal_chirp.4.nom_chirp_amp: 4 elements
nominal_chirp.4.nom_chirp_phs: 4 elements
nominal_chirp.5.nom_chirp_amp: 4 elements
nominal_chirp.5.nom_chirp_phs: 4 elements
beam_merge_sl_range: 4 elements
beam_merge_alg_param: 4 elements
```

2.1.4 Image data

Dealing with image data is simple as well:

```
In [19]: product.get_band_names()
Out[19]: ['slant_range_time',
          'incident_angle',
          'latitude',
          'longitude',
          'proc_data']

In [19]: band = product.get_band('proc_data')

In [20]: data = band. [TAB]
band.bm_expr                    band.read_raster
band.create_compatible_raster   band.sample_model
band.data_type                  band.scaling_factor
band.description                band.scaling_method
band.get_name                   band.scaling_offset
band.lines_mirrored             band.spectr_band_index
band.product                    band.unit
band.read_as_array

In [20]: data = band.read_as_array(1000, 1000, xoffset=100, \
yoffset=6500, xstep=2, ystep=2)

In [21]: data
Out[21]:
array([[ 146.,  153.,  134., ...,   51.,   55.,   72.],
       [ 198.,  163.,  146., ...,   26.,   54.,   57.],
       [ 127.,  205.,  105., ...,   64.,   76.,   61.],
       ...,
       [  64.,   78.,   52., ...,   96.,  176.,  159.],
       [  66.,   41.,   45., ...,  200.,  153.,  203.],
       [  64.,   71.,   88., ...,  289.,  182.,  123.]],␣
 ↪dtype=float32)

In [22]: data.shape
Out[22]: (500, 500)
```

(continues on next page)

(continued from previous page)

```
In [23]: imshow(data, cmap=cm.gray, vmin=0, vmax=1000)
Out[23]: <matplotlib.image.AxesImage object at 0x60dcf10>

In [24]: title(band.description)
Out[24]: <matplotlib.text.Text object at 0x67e9950>

In [25]: colorbar()
Out[25]: <matplotlib.colorbar.Colorbar instance at 0x6b18cb0>
```

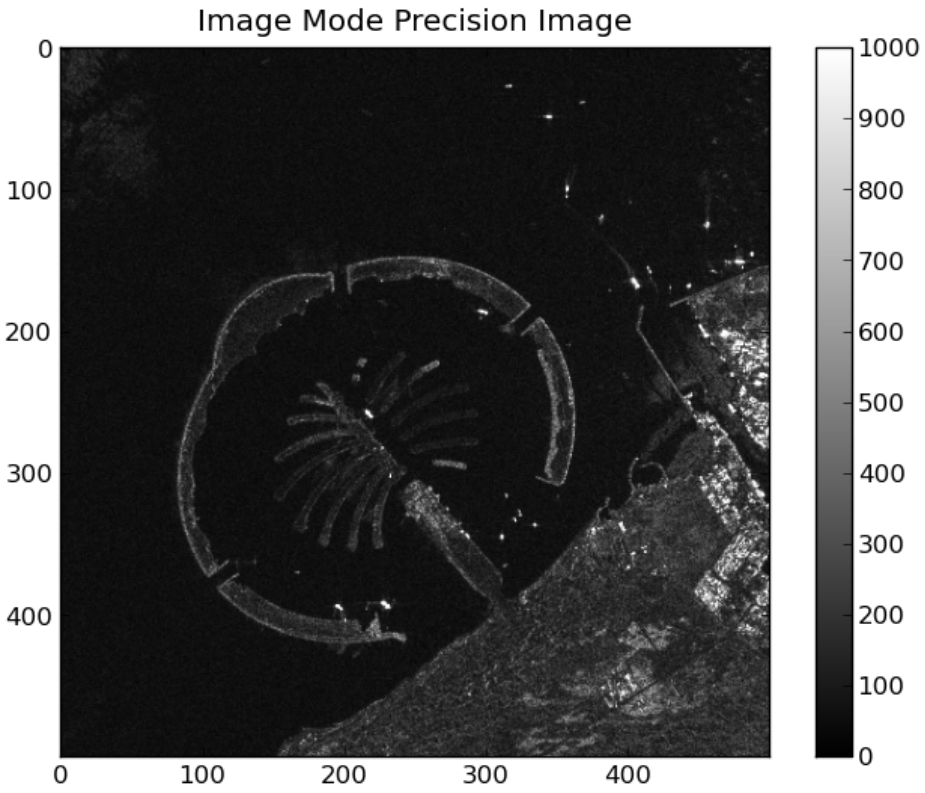

Fig. 1: Image data read from the "proc_data" band

2.1.5 Closing the epr.Product

Finally the *epr.Product* can be closed using the *epr.Product.close()* method:

```
In [26]: product.close()
```

After a product is closed no more I/O operations can be performed on it. Any attempt to do it will raise a ValueError:

```
In [27]: product.tot_size / 1024.**2
---------------------------------------------------------------------------
↪------
```

(continues on next page)

(continued from previous page)

```
ValueError                                Traceback (most recent call␣
↪last)
<ipython-input-13-6420c80534dc> in <module>()
----> 1 product.tot_size / 1024.**2

epr.so in epr.Product.tot_size.__get__ (src/epr.c:16534)()

epr.so in epr.Product.check_closed_product (src/epr.c:16230)()

ValueError: I/O operation on closed file
```

At any time the user can check whenever a *epr.Product* is closed or not using the *epr.Product.closed* property:

```
In [28]: product.closed
Out[28]: True
```

2.2 Exporting band data

This tutorial shows how to convert ENVISAT raster information from dataset and generate flat binary rasters using PyEPR.

The program generates as many raster as the dataset specified in input.

The example code (`examples/write_bands.py`) is a direct translation of the C sample program write_bands.c bundled with the EPR API distribution.

The program is invoked as follows:

```
$ python write_bands.py <envisat-product> \
<output directory for the raster file> <dataset name 1> \
[<dataset name 2> ... <dataset name N>]
```

2.2.1 Import section

To use the Python EPR API one have to import *epr* module.

At first import time the underlaying C library is opportunely initialized.

```
#!/usr/bin/env python3

# This program is a direct translation of the sample program
# "write_bands.c" bundled with the EPR-API distribution.
#
# Source code of the C program is available at:
# https://github.com/bcdev/epr-api/blob/master/src/examples/write_bands.c


from __future__ import print_function

import os
import sys
import epr
```

2.2.2 The main program

The main program in quite simple (this is just an example).

```
def main(*argv):
    '''A program for converting producing ENVI raster information from
    dataset.

    It generates as many raster as there are dataset entrance parameters.

    Call::
```

(continues on next page)

(continued from previous page)

```
        $ write_bands.py <envisat-product>
                         <output directory for the raster file>
                         <dataset name 1>
                         [<dataset name 2> ... <dataset name N>]

    Example::

        $ write_bands.py \
        MER_RR__1PNPDK20020415_103725_000002702005_00094_00649_1059.N1 \
        . latitude

    '''

    if not argv:
        argv = sys.argv

    if len(argv) <= 3:
        print('Usage: write_bands.py <envisat-product> <output-dir> '
              '<dataset-name-1>')
        print('                      [<dataset-name-2> ... <dataset-name-N>
↪]')
        print('  where envisat-product is the input filename')
        print('  and output-dir is the output directory')
        print('  and dataset-name-1 is the name of the first band to be '
              'extracted (mandatory)')
        print('  and dataset-name-2 ... dataset-name-N are the names of '
              'further bands to be extracted (optional)')
        print('Example:')
        print('  write_bands MER_RR__2P_TEST.N1 . latitude')
        print()
        sys.exit(1)

    product_file_path = argv[1]
    output_dir_path = argv[2]

    # Open the product; an argument is a path to product data file
    with epr.open(product_file_path) as product:
        for band_name in argv[3:]:
            write_raw_image(output_dir_path, product, band_name)


if __name__ == '__main__':
    main()
```

It performs some basic command line arguments handling and then open the input product.

```
    # Open the product; an argument is a path to product data file
    with epr.open(product_file_path) as product:
```

Finally the core function (`write_raw_image()`) is called on each band specified on the command:

```
        for band_name in argv[3:]:
            write_raw_image(output_dir_path, product, band_name)
```

2.2.3 The `write_raw_image()` core function

The core function is `write_raw_image()`.

```
def write_raw_image(output_dir, product, band_name):
    '''Generate the ENVI binary pattern image file for an actual DS.

    The first parameter is the output directory path.

    '''

    # Build ENVI file path, DS name specifically
    image_file_path = os.path.join(output_dir, band_name + '.raw')

    band = product.get_band(band_name)
    source_w = product.get_scene_width()
    source_h = product.get_scene_height()
    source_step_x = 1
    source_step_y = 1

    raster = band.create_compatible_raster(source_w, source_h,
                                           source_step_x, source_step_y)

    print('Reading band "%s"...' % band_name)
    raster = band.read_raster(0, 0, raster)

    out_stream = open(image_file_path, 'wb')

    for line in raster.data:
        out_stream.write(line.tostring())
    # or better: raster.data.tofile(out_stream)

    out_stream.close()

    print('Raw image data successfully written to "%s".' % image_file_path)
    print('C data type is "%s", element size %u byte(s), '
          'raster size is %u x %u pixels.' % (
          epr.data_type_id_to_str(raster.data_type),
          raster.get_elem_size(),
          raster.get_width(),
          raster.get_height()))
```

It generates a flat binary file with data of a single band whose name is specified as input parameter.

First the output file name is computed using the `os` module.

```
    # Build ENVI file path, DS name specifically
    image_file_path = os.path.join(output_dir, band_name + '.raw')
```

Then the desired band is retrieved using the *`epr.Product.get_band()`* method and some of its parameters are loaded in to local variables:

```
    band = product.get_band(band_name)
```

Band data are accessed by means of a *epr.Raster* object.

See also:

epr.Band.read_as_array()

The *epr.Band.create_compatible_raster()* is a facility method that allows to instantiate a *epr.Raster* object with a data type compatible with the band data:

```
    raster = band.create_compatible_raster(source_w, source_h,
                                           source_step_x, source_step_y)
```

Then data are read using the *epr.Band.read_raster()* method:

```
    print('Reading band "%s"...' % band_name)
    raster = band.read_raster(0, 0, raster)
```

Then the output file object is created (in binary mode of course)

```
    out_stream = open(image_file_path, 'wb')
```

and data are copied to the output file one line at time

```
    for line in raster.data:
        out_stream.write(line.tostring())
```

Please note that it has been used *epr.Raster.data* attribute of the *epr.Raster* objects that exposes *epr.Raster* data with the powerful `numpy.ndarray` interface.

Note: copying one line at time is not the best way to perform the task in Python. It has been done just to mimic the original C code:

```
out_stream = fopen(image_file_path, "wb");
if (out_stream == NULL) {
    printf("Error: can't open '%s'\n", image_file_path);
    return 3;
}

for (y = 0; y < (uint)raster->raster_height; ++y) {
    numwritten = fwrite(epr_get_raster_line_addr(raster, y),
                        raster->elem_size,
                        raster->raster_width,
                        out_stream);

    if (numwritten != raster->raster_width) {
        printf("Error: can't write to %s\n", image_file_path);
        return 4;
    }
}
fclose(out_stream);
```

A by far more pythonic solution would be:

```
raster.data.tofile(out_stream)
```

2.3 Exporting bitmasks

This tutorial shows how to generate bit masks from ENVISAT flags information as "raw" image using PyEPR.

The example code (`examples/write_bitmask.py`) is a direct translation of the C sample program write_bitmask.c bundled with the EPR API distribution.

The program is invoked as follows:

```
$ python write_bitmask.py <envisat-product> <bitmask-expression> \
<output-file>
```

The `examples/write_bitmask.py` code consists in a single function that also includes command line arguments handling:

```
#!/usr/bin/env python3

# This program is a direct translation of the sample program
# "write_bitmask.c" bundled with the EPR-API distribution.
#
# Source code of the C program is available at:
# https://github.com/bcdev/epr-api/blob/master/src/examples/write_bitmask.c

'''Generates bit mask from ENVISAT flags information as "raw" image
for (e.g.) Photoshop

Call::

    $ python write_bitmask.py <envisat-product> <bitmask-expression>
    <output-file>

Example to call the main function::

    $ python write_bitmask.py MER_RR__2P_TEST.N1 \
    'l2_flags.LAND and !l2_flags.BRIGHT' my_flags.raw

'''

from __future__ import print_function

import sys
import epr


def main(*argv):
    if not argv:
        argv = sys.argv

    if len(argv) != 4:
        print('Usage: write_bitmask <envisat-product> <bitmask-expression>
↪'
              '<output-file>')
        print('  where envisat-product is the input filename')
        print('  and bitmask-expression is a string containing the bitmask
↪'
```

(continues on next page)

(continued from previous page)

```
                'logic')
        print('  and output-file is the output filename.')
        print('Example:')
        print("  MER_RR__2P_TEST.N1 'l2_flags.LAND and not l2_flags.BRIGHT
↪' "
              "my_flags.raw")
        print
        sys.exit(1)

    product_file_path = argv[1]
    bm_expr = argv[2]
    image_file_path = argv[3]

    # Open the product; an argument is a path to product data file
    with epr.open(product_file_path) as product:
        offset_x = 0
        offset_y = 0
        source_width = product.get_scene_width()
        source_height = product.get_scene_height()
        source_step_x = 1
        source_step_y = 1

        bm_raster = epr.create_bitmask_raster(source_width, source_height,
                                              source_step_x, source_step_y)

        product.read_bitmask_raster(bm_expr, offset_x, offset_y, bm_raster)

        with open(image_file_path, 'wb') as out_stream:
            bm_raster.data.tofile(out_stream)

    print('Raw image data successfully written to "%s".' % image_file_path)
    print('Data type is "byte", size is %d x %d pixels.' % (source_width,
                                                            source_height))


if __name__ == '__main__':
    main()
```

In order to use the Python EPR API the *epr* module is imported:

```
import epr
```

As usual the ENVISAT product is opened using the *epr.open()* function that returns an *epr.Product* instance. In this case the *epr.open()* is used together with a `with` statement so that the *epr.Product* instance is closed automatically when the program exits the `with` block.

```
    # Open the product; an argument is a path to product data file
    with epr.open(product_file_path) as product:
```

Scene size parameters are retrieved form the *epr.Product* object using the *epr.Product.get_scene_width()* and *epr.Product.get_scene_height()* methods:

```
        source_width = product.get_scene_width()
        source_height = product.get_scene_height()
```

The EPR API allows to manage data by means of *epr.Raster* objects, so the function *epr.create_bitmask_raster()*, specific for bitmasks, is used to create a *epr.Raster* instance.

See also:

epr.create_raster()

Data are actually read using the *epr.Product.read_bitmask_raster()* method of the *epr.Product* class:

```
        product.read_bitmask_raster(bm_expr, offset_x, offset_y, bm_raster)
```

The *epr.Product.read_bitmask_raster()* method receives in input the *bm_expr* parameter that is set via command line:

```
    bm_expr = argv[2]
```

bm_expr is a string that define the logical expression for the definition of the bit-mask. In a bit-mask expression, any number of the flag-names (found in the DDDB) can be composed with "(", ")", "NOT", "AND", "OR".

Valid bit-mask expression are for example:

```
flags.LAND OR flags.CLOUD
```

or:

```
NOT flags.WATER AND flags.TURBID_S
```

Finally data are written to disk as a flat binary file using the `numpy.ndarray.tofile()` method of the *epr.Raster.data* attribute of the *epr.Raster* objects that exposes data via the `numpy.ndarray` interface:

```
        with open(image_file_path, 'wb') as out_stream:
            bm_raster.data.tofile(out_stream)
```

2.4 NDVI computation

This tutorial shows how to use PyEPR to open a MERIS L1B product, compute the *Normalized Difference Vegetation Index* (NDVI) and store it into a flat binary file.

The example code (`examples/write_ndvi.py`) is a direct translation of the C sample program write_ndvi.c bundled with the EPR API distribution.

The program is invoked as follows:

```
$ python write_ndvi.py <envisat-oroduct> <output-file>
```

The code have been kept very simple and it consists in a single function (`main()`) that also performs a minimal command line arguments handling.

```
#!/usr/bin/env python3

# This program is a direct translation of the sample program
# "write_ndvi.c" bundled with the EPR-API distribution.
#
# Source code of the C program is available at:
# https://github.com/bcdev/epr-api/blob/master/src/examples/write_ndvi.c


'''Example for using the epr-api

Demonstrates how to open a MERIS L1b product and calculate the NDVI.

This example does not demonstrate how to write good and safe code.
It is reduced to the essentials for working with the epr-api.

Calling sequence::

    $ python write_ndvi.py <envisat-product> <output-file>

for example::

    $ python write_ndvi.py MER_RR__1P_test.N1 my_ndvi.raw

'''

from __future__ import pront_function

import sys
import struct
import logging

import epr


def main(*argv):
    if not argv:
        argv = sys.argv

    if len(argv) != 3:
```

(continues on next page)

(continued from previous page)

```
        print('Usage: write_ndvi <envisat-product> <output-file>')
        print('  where envisat-product is the input filename')
        print('  and output-file is the output filename.')
        print('Example: MER_RR__1P_TEST.N1 my_ndvi.raw')
        print
        sys.exit(1)

    # Open the product
    with epr.open(argv[1]) as product:

        # The NDVI shall be calculated using bands 6 and 8.
        band1_name = 'radiance_6'
        band2_name = 'radiance_10'

        band1 = product.get_band(band1_name)
        band2 = product.get_band(band2_name)

        # Allocate memory for the rasters
        width = product.get_scene_width()
        height = product.get_scene_height()
        subsampling_x = 1
        subsampling_y = 1
        raster1 = band1.create_compatible_raster(width, height,
                                                 subsampling_x,␣
↪subsampling_y)
        raster2 = band2.create_compatible_raster(width, height,
                                                 subsampling_x,␣
↪subsampling_y)

        # Read the radiance into the raster.
        offset_x = 0
        offset_y = 0

        logging.info('read "%s" data' % band1_name)
        band1.read_raster(offset_x, offset_y, raster1)

        logging.info('read "%s" data' % band2_name)
        band2.read_raster(offset_x, offset_y, raster2)

        # Open the output file
        logging.info('write ndvi to "%s"' % argv[2])
        with open(argv[2], 'wb') as out_stream:

            # Loop over all pixel and calculate the NDVI.
            #
            # @NOTE: looping over data matrices is not the best soluton.
            #        It is done here just for demostrative purposes
            for j in range(height):
                for i in range(width):
                    rad1 = raster1.get_pixel(i, j)
                    rad2 = raster2.get_pixel(i, j)
                    if (rad1 + rad2) != 0.0:
                        ndvi = (rad2 - rad1) / (rad2 + rad1)
                    else:
                        ndvi = -1.0
                    out_stream.write(struct.pack('f', ndvi))
```

(continues on next page)

(continued from previous page)

```
            logging.info('ndvi was written success')


if __name__ == '__main__':
    main()
```

The ENVISAT *epr.Product* is opened using the *epr.open()* function.

```
    with epr.open(argv[1]) as product:
```

As usual in modern python programs the *with* statement has been used to ensure that the product is automatically closed as soon as the program exits the block. Of course it is possible to use a simple assignment form:

```
product = open(argv[1])
```

but in this case the user should take care of manually call:

```
product.close()
```

when appropriate.

The name of the product is in the first argument passed to the program. In order to keep the code simple no check is performed to ensure that the product is a valid L1B product.

The NDVI is calculated using bands 6 and 8 (the names of these bands are "radiance_6" and "radiance_10"). *epr.Band* objects are retrieved using the *epr.Product.get_band()* method:

```
        # The NDVI shall be calculated using bands 6 and 8.
        band1_name = 'radiance_6'
        band2_name = 'radiance_10'

        band1 = product.get_band(band1_name)
        band2 = product.get_band(band2_name)
```

band1 and *band2* are used to read the calibrated radiances into the *epr.Raster* objects that allow to access data matrices with the radiance values.

Before reading data into the *epr.Raster* objects they have to be instantiated specifying their size and data type in order to allow the library to allocate the correct amount of memory.

For sake of simplicity *epr.Raster* object are created with the same size of the whole product (with no sub-sampling) using the *epr.Band.create_compatible_raster()* method of the *epr.Band* class:

```
        # Allocate memory for the rasters
        width = product.get_scene_width()
        height = product.get_scene_height()
        subsampling_x = 1
        subsampling_y = 1
        raster1 = band1.create_compatible_raster(width, height,
                                                 subsampling_x,␣
↪subsampling_y)
```

(continues on next page)

(continued from previous page)

```
        raster2 = band2.create_compatible_raster(width, height,
                                                 subsampling_x,␣
↪subsampling_y)
```

Then data are actually loaded into memory using the *epr.Band.read_raster()* method. Since *epr.Raster* objects have been defined to match the whole product, offset parameters are set to zero (data are read starting from specified offset):

```
        # Read the radiance into the raster.
        offset_x = 0
        offset_y = 0

        logging.info('read "%s" data' % band1_name)
        band1.read_raster(offset_x, offset_y, raster1)

        logging.info('read "%s" data' % band2_name)
        band2.read_raster(offset_x, offset_y, raster2)
```

Note: in this simplified example it is assumed that there is enough system memory to hold the two *epr.Raster* objects.

After opening (in binary mode) the stream for the output

```
        # Open the output file
        logging.info('write ndvi to "%s"' % argv[2])
        with open(argv[2], 'wb') as out_stream:
```

the program simply loops over all pixel and calculate the NDVI with the following formula:

$$NDVI = \frac{radiance_{10} - radiance_8}{radiance_{10} + radiance_8}$$

```
            # Loop over all pixel and calculate the NDVI.
            #
            # @NOTE: looping over data matrices is not the best soluton.
            #        It is done here just for demostrative purposes
            for j in range(height):
                for i in range(width):
                    rad1 = raster1.get_pixel(i, j)
                    rad2 = raster2.get_pixel(i, j)
                    if (rad1 + rad2) != 0.0:
                        ndvi = (rad2 - rad1) / (rad2 + rad1)
                    else:
                        ndvi = -1.0
                    out_stream.write(struct.pack('f', ndvi))
            logging.info('ndvi was written success')
```

This part of the code tries to mimic closely the original C code (write_ndvi.c)

```
out_stream = fopen(argv[2], "wb");
for (j = 0; j < height; ++j) {
    for (i = 0; i < width; ++i) {
        rad1 = epr_get_pixel_as_float(raster1, i, j);
        rad2 = epr_get_pixel_as_float(raster2, i, j);
        if ((rad1 + rad2) != 0.0) {
            ndvi = (rad2 - rad1) / (rad2 + rad1);
        } else {
            ndvi = -1.0;
        }
        status = fwrite( & ndvi, sizeof(float), 1, out_stream);
    }
}
epr_log_message(e_log_info, "ndvi was written success");
```

and uses the *epr.Raster.get_pixel()* method to access pixel values and perform computation.

The Python `struct.pack()` function together with `file.write()` is used to write the NDVI of the pixel n the file in binary format.

```
out_stream.write(struct.pack('f', ndvi))
```

Note: the entire solution is quite not pythonic. As an alternative implementation it could be used the `numpy.ndarray` interface of *epr.Raster* objects available via the *epr.Raster.data* property. The NDVI index is computed on all pixels altogether using vectorized expressions:

```
# Initialize the entire matrix to -1
ndvi = numpy.zeros((height, width), 'float32') - 1

aux = raster2.data + raster1.data

# indexes of pixel with non null denominator
idx = numpy.where(aux != 0)

# actual NDVI computation
ndvi[idx] = (raster2.data[idx] - raster1.data[idx]) / aux[idx]
```

Finally data can be saved to file simply using the `numpy.ndarray.tofile()` method:

```
ndvi.tofile(out_stream)
```

2.5 Update `Field` elements

The EPR C API has been designed to provide read-only features.

PyEPR provides and extra capability consisting in the possibility to modify (*update*) an existing ENVISAT `Product`.

Lets consider a MERIS Level 2 low resolution product (*MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_0001.N1*). It has a `Band` named *water_vapour* containing the water vapour content at a specific position.

One can load water vapour and compute an histogram using the following instructions:

```
FILENAME = 'MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_0001.N1'

# load original data
with epr.open(FILENAME) as product:
    band = product.get_band('water_vapour')
    wv_orig_histogram, orig_bins = np.histogram(band.read_as_array().flat,␣
↪50)
```

The resulting histogram can be plot using Matplotlib:

```
# plot water vapour histogram
plt.figure()
plt.bar(orig_bins[:-1], wv_orig_histogram, 0.02, label='original')
plt.grid(True)
plt.title('Water Vapour Histogram')
```

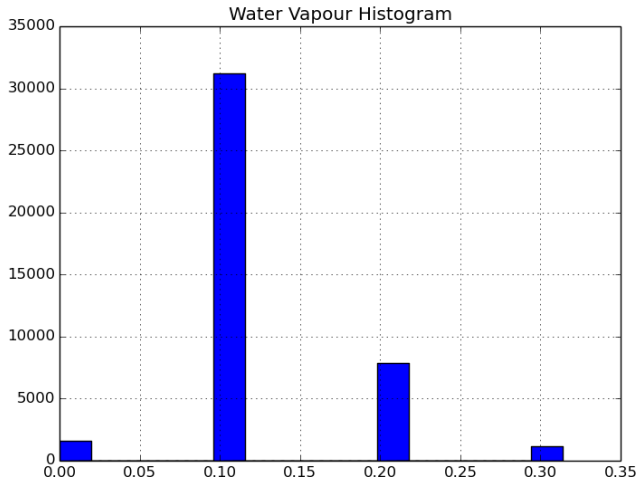

Fig. 2: Histogram of the original water vapour content

The actual values of the water vapour content `Band` are computed starting form data stored in the *Vapour_Content* `Dataset` using scaling factors contained in the *Scaling_Factor_GADS* `Dataset`. In particular `Fields` *sf_wvapour* and *off_wvapour* are used:

```
In [21]: dataset = product.get_dataset('Scaling_Factor_GADS')

In [22]: print(dataset)
epr.Dataset(Scaling_Factor_GADS) 1 records

sf_cl_opt_thick = 1.000000
sf_cloud_top_press = 4.027559
sf_wvapour = 0.100000
off_cl_opt_thick = -1.000000
off_cloud_top_press = -4.027559
off_wvapour = -0.100000
spare_1 = <<unknown data type>>
```

Now suppose that for some reason one needs to update the *sf_wvapour* scaling factor for the water vapour content. Changing the scaling factor, of course, will change all values in the *water_vapour* `Band`.

The change can be performed using the `Field.set_elem()` and `Field.set_elems()` methods of `Field` objects:

```
# modily scaling facotrs
with epr.open(FILENAME, 'rb+') as product:
    dataset = product.get_dataset('Scaling_Factor_GADS')
    record = dataset.read_record(0)

    field = record.get_field('sf_wvapour')
    scaling = field.get_elem()
    scaling *= 1.1
    field.set_elem(scaling)
```

Now the *sf_wvapour* scaling factor has been changed and it is possible to compute and display the histogram of modified data in the *water_vapour* `Band`:

```
# re-open the product and load modified data
with epr.open(FILENAME) as product:
    band = product.get_band('water_vapour')
    unit = band.unit
    new_data = band.read_as_array()
    wv_new_histogram, new_bins = np.histogram(new_data.flat, 50)

# plot histogram of modified data
plt.figure()
plt.bar(orig_bins[:-1], wv_orig_histogram, 0.02, label='original')
plt.grid(True)
plt.title('Water Vapour Histogram')
plt.hold(True)
plt.bar(new_bins[:-1], wv_new_histogram, 0.02, color='red', label='new')
plt.legend()
```

Figure above shows the two different histograms, original data in blue and modified data in red, demonstrating the effect of the change of the scaling factor.

The new map of water vapour is showed in the following picture:

Important: it is important to stress that it is necessary to close and re-open the `Product` in

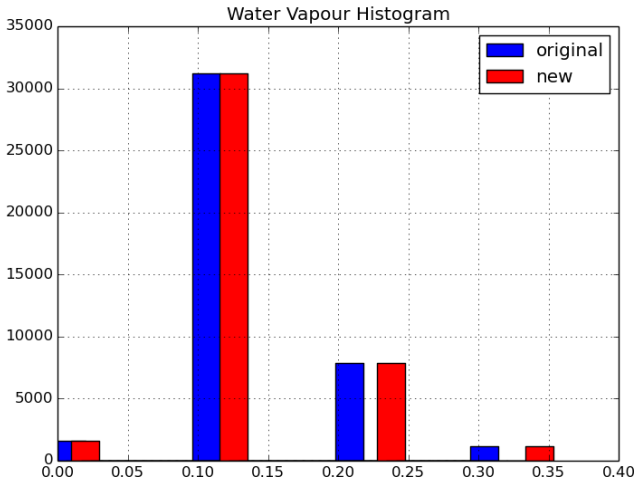

Fig. 3: Histogram of the water vapour content (original and modified)

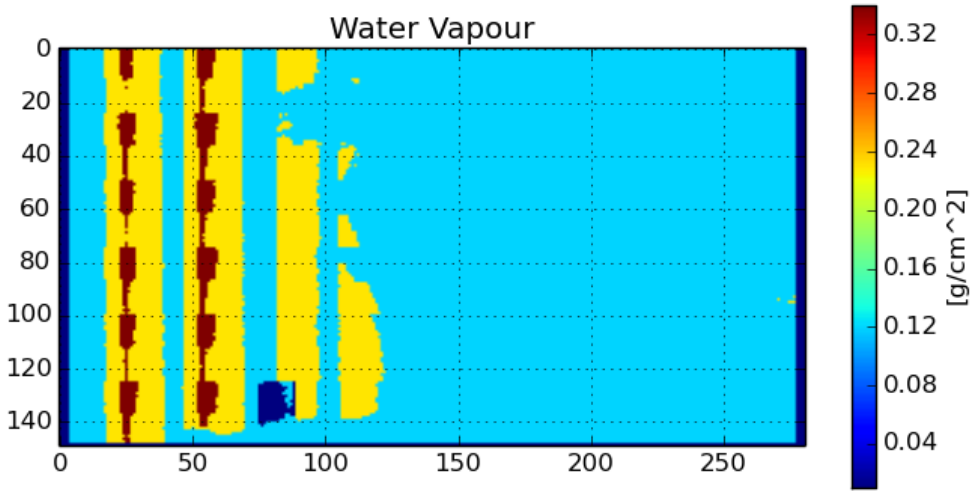

Fig. 4: Modified water vapour content map

order to see changes in the scaling factors applied to the *water_vapour*`Band` data.

This is a limitation of the current implementation that could be removed in future versions of the PyEPR package.

It has been showed that changing the *sf_wvapour* scaling factor modifies all values of the *water_vapour* `Band`.

Now suppose that one needs to modify only a specific area. It can be done changing the contents of the *Vapour_Content* `Dataset`.

The `Dataset` size can be read form the `Product`:

```
In [44]: product.get_scene_height(), product.get_scene_width()
Out[44]: (149, 281)
```

while information about the fields in each record can be retrieved introspecting the `Record` object:

```
In [49]: record = dataset.read_record(0)

In [50]: record.get_field_names()
Out[50]: ['dsr_time', 'quality_flag', 'wvapour_cont_pix']

In [51]: record.get_field('wvapour_cont_pix')
Out[51]: epr.Field("wvapour_cont_pix") 281 uchar elements
```

So the name of the `Field` we need to change is the *wvapour_cont_pix*, and its index is *2*.

It is possible to change a small box inside the `Dataset` as follows:

```
# modify the "Vapour_Content" dataset
with epr.open(FILENAME, 'rb+') as product:
    dataset = product.get_dataset('Vapour_Content')
    for line in range(70, 100):
        record = dataset.read_record(line)
        field = record.get_field_at(2)
        elems = field.get_elems()
        elems[50:100] = 0
        field.set_elems(elems)
```

Please note that when one modifies the content of a `Dataset` he/she should also take into account id the corresponding band has lines mirrored or not:

```
In [59]: band = p.get_band('water_vapour')

In [60]: band.lines_mirrored
Out[60]: True
```

Finally the `Product` can be re-opened to load and display the modified `Band`:

```
with epr.open(FILENAME) as product:
    band = product.get_band('water_vapour')
    unit = band.unit
```

(continues on next page)

(continued from previous page)

```
    data = band.read_as_array()

# plot the water vapour map
plt.figure(figsize=(8, 4))
plt.imshow(data)
plt.grid(True)
plt.title('Water Vapour with box')
cb = plt.colorbar()
cb.set_label('[{}]'.format(unit))
```

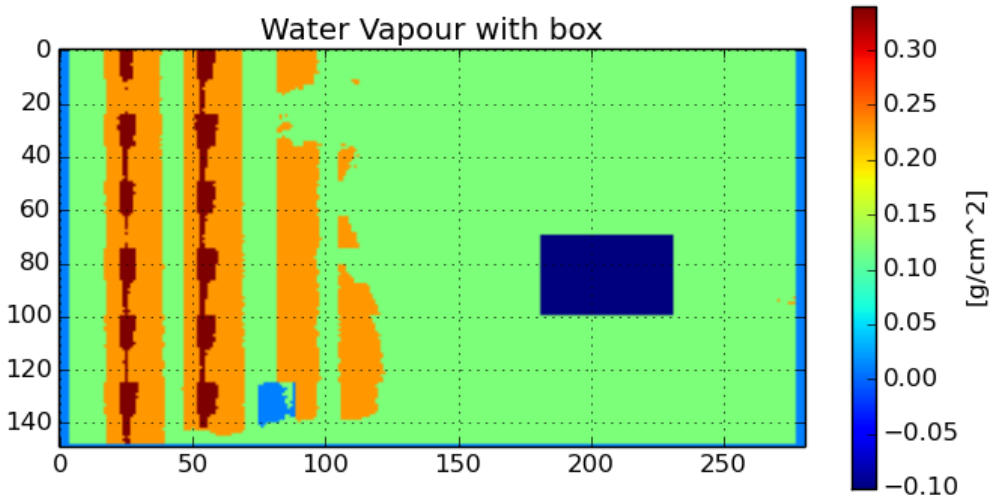

Fig. 5: Modified water vapour content map with zeroed box

Of course values in the box that has been set to zero in the `Dataset` are transformed according to the scaling factor and offset parameters associated to *water_vapour* `Band`.

The complete code of the example can be found at `examples/update_elements.py`.

2.6 GDAL export example

This tutorial explains how to use PyEPR to generate a file in GDAL Virtual Format (VRT) that can be used to access data with the powerful and popular GDAL library. GDAL already has support for ENVISAT products but this example is interesting for two reasons:

- it exploits some low level feature (like e.g. offset management) that are rarely used but that can be very useful in some cases
- the generated VRT file uses raw raster access and it can be opened in update mode to modify the ENVISAT product data. This feature is not supported by the native ENVISAT driver of the GDAL library

2.6.1 export_gdalvrt module

The complete code of the example is available in the `examples/export_gdalvrt.py` file. It is organized so that it can also be imported as a module in any program.

The *export_gdalvrt* module provides two functions:

`export_gdalvrt.`**`epr2gdal_band`**(*band*, *vrt*)
Takes in input an *epr.Band* object and a VRT dataset and add a GDAL band to the VRT dataset

`export_gdalvrt.`**`epr2gdal`**(*product*, *vrt*, *overwrite_existing=False*)
Takes in input a PyEPR `Product` (or a filename) and the file name of the output VRT file and generates the VRT file itself containing a band for each *epr.Band* present in the original *epr.Product* and also associated metadata.

The *epr2gdal()* function first creates the VRT dataset

```
    if isinstance(product, str):
        filename = product
        product = epr.open(filename)

    ysize = product.get_scene_height()
    xsize = product.get_scene_width()

    if os.path.exists(vrt) and not overwrite_existing:
        raise ValueError('unable to create "{0}". Already exists'.
↪format(vrt))

    driver = gdal.GetDriverByName('VRT')
    if driver is None:
        raise RuntimeError('unable to get driver "VRT"')

    gdal_ds = driver.Create(vrt, xsize, ysize, 0)
    if gdal_ds is None:
        raise RuntimeError('unable to create "{}" dataset'.format(vrt))
```

and then loops on all *epr.Band*s of the PyEPR *epr.Product* calling the *epr2gdal_band()* function on each of them:

```
    for band in product.bands():
        epr2gdal_band(band, gdal_ds)
```

The *export_gdalvrt* module also provides a epr_to_gdal_type mapping between EPR and GDAL data type identifiers.

2.6.2 Generating *VRTRawRasterBand*s

The core of the example is the part of the code in the *epr2gdal_band()* function that generates the GDAL *VRTRawRasterBand*. It is a description of a raster file that the GDAL library uses for low level data access. Of course the entire machinery works because data in *epr.Band*s and *epr.Dataset*s of ENVISAT products are stored as contiguous rasters.

```
    filename = os.pathsep.join(product.file_path.split('/'))  # denormalize
    offset = dataset.get_dsd().ds_offset + field.get_offset()
    line_offset = record.tot_size
    pixel_offset = epr.get_data_type_size(field.get_type())

    if band.sample_model == epr.E_SMOD_1OF2:
        pixel_offset *= 2
    elif band.sample_model == epr.E_SMOD_2OF2:
        offset += pixel_offset
        pixel_offset *= 2

    options = [
        'subClass=VRTRawRasterBand',
        'SourceFilename={}'.format(filename),
        'ImageOffset={}'.format(offset),
        'LineOffset={}'.format(line_offset),
        'PixelOffset={}'.format(pixel_offset),
        'ByteOrder=MSB',
    ]

    gtype = epr_to_gdal_type[field.get_type()]
    ret = gdal_ds.AddBand(gtype, options=options)
    if ret != gdal.CE_None:
        raise RuntimeError(
            'unable to add VRTRawRasterBand to "{}"'.format(vrt))
```

The fundamental part is the computation of the:

ImageOffset:

> the offset in bytes to the beginning of the first pixel of data with respect to the beginning of the file.
>
> In the example it is computed using
>
> - the *epr.DSD.ds_offset* attribute, that represents the offset in bytes of the *epr.Dataset* from the beginning of the file, and
> - the *epr.Field.get_offset()* method that returns the offset in bytes of the *epr.Field* containing *epr.Band* data from the beginning of the *epr.Record*

```
offset = dataset.get_dsd().ds_offset + field.get_offset()
```

LineOffset:

the offset in bytes from the beginning of one *scanline* of data and the next scanline of data. In the example it is set to the *epr.Record* size in bytes using the *epr.Record.tot_size* attribute:

```
line_offset = record.tot_size
```

PixelOffset:

the offset in bytes from the beginning of one pixel and the next on the same line. Usually it corresponds to the size in bytes of the elementary data type. It is set using the *epr.Field.get_type()* method and the *epr.get_data_type_size()* function:

```
pixel_offset = epr.get_data_type_size(field.get_type())
```

The band size in lines and columns of the GDAL bands is fixed at GDAL dataset level when it is created:

```
        gdal_ds = driver.Create(vrt, xsize, ysize, 0)
        if gdal_ds is None:
            raise RuntimeError('unable to create "{}" dataset'.format(vrt))
```

Please note that in case of *epr.Dataset*s storing complex values, like in *MDS1* *epr.Dataset* of ASAR IMS *epr.Product*s, pixels of real and imaginary parts are interleaved, so to represent *epr.Band*s of the two components the pixel offset have to be doubled and an additional offset (one pixel) must be added to the *ImageOffset* of the *epr.Band* representing the imaginary part:

```
    if band.sample_model == epr.E_SMOD_1OF2:
        pixel_offset *= 2
    elif band.sample_model == epr.E_SMOD_2OF2:
        offset += pixel_offset
        pixel_offset *= 2
```

Note: the PyEPR API does not supports complex Bands. *epr.Dataset*s containing complex data, like the *MDS1* *epr.Dataset* of ASAR IMS *epr.Product*s, are associated to two distinct *epr.Band*s containing the real (I) and the imaginary (Q) component respectively.

GDAL, instead, supports complex data types, so it is possible to map a complex ENVISAT *epr.Dataset* onto a single GDAL bands with complex data type.

This case is not handled in the example.

2.6.3 Metadata

The *epr2gdal_band()* function also stores a small set of metadata for each *epr.Band*:

```
    gdal_band = gdal_ds.GetRasterBand(gdal_ds.RasterCount)
    gdal_band.SetDescription(band.description)
    metadata = {
        'name': band.get_name(),
        'dataset_name': dataset.get_name(),
        'dataset_description': dataset.description,
        'lines_mirrored': str(band.lines_mirrored),
        'sample_model': epr.get_sample_model_name(band.sample_model),
        'scaling_factor': str(band.scaling_factor),
        'scaling_offset': str(band.scaling_offset),
        'scaling_method': epr.get_scaling_method_name(band.scaling_method),
        'spectr_band_index': str(band.spectr_band_index),
        'unit': band.unit if band.unit else '',
        'bm_expr': band.bm_expr if band.bm_expr else '',
    }
    gdal_band.SetMetadata(metadata)
```

Metadata are also stored at GDAL dataset level by the *epr2gdal()* function:

```
    metadata = {
        'id_string': product.id_string,
        'meris_iodd_version': str(product.meris_iodd_version),
        'dataset_names': ','.join(product.get_dataset_names()),
        'num_datasets': str(product.get_num_datasets()),
        'num_dsds': str(product.get_num_dsds()),
    }
    gdal_ds.SetMetadata(metadata)
```

The *epr2gdal()* function also stores the contents of the *MPH* and the *SPH* records as GDAL dataset matadata in custom domains:

```
    mph = product.get_mph()
    metadata = str(mph).replace(' = ', '=').split('\n')
    gdal_ds.SetMetadata(metadata, 'MPH')

    sph = product.get_sph()
    metadata = str(sph).replace(' = ', '=').split('\n')
    gdal_ds.SetMetadata(metadata, 'SPH')
```

2.6.4 Complete listing

```
#!/usr/bin/env python3

import os
import epr
from osgeo import gdal


epr_to_gdal_type = {
    epr.E_TID_UNKNOWN: gdal.GDT_Unknown,
    epr.E_TID_UCHAR: gdal.GDT_Byte,
    epr.E_TID_CHAR: gdal.GDT_Byte,
    epr.E_TID_USHORT: gdal.GDT_UInt16,
```

(continues on next page)

(continued from previous page)

```
    epr.E_TID_SHORT: gdal.GDT_Int16,
    epr.E_TID_UINT: gdal.GDT_UInt32,
    epr.E_TID_INT: gdal.GDT_Int32,
    epr.E_TID_FLOAT: gdal.GDT_Float32,
    epr.E_TID_DOUBLE: gdal.GDT_Float64,
    #epr.E_TID_STRING: gdal.GDT_Unknown,
    #epr.E_TID_SPARE: gdal.GDT_Unknown,
    #epr.E_TID_TIME: gdal.GDT_Unknown,
}


def epr2gdal_band(band, vrt):
    product = band.product
    dataset = band.dataset
    record = dataset.read_record(0)
    field = record.get_field_at(band._field_index - 1)

    ysize = product.get_scene_height()
    xsize = product.get_scene_width()

    if isinstance(vrt, gdal.Dataset):
        if (vrt.RasterYSize, vrt.RasterXSize) != (ysize, xsize):
            raise ValueError('dataset size do not match')
        gdal_ds = vrt
    elif os.path.exists(vrt):
        gdal_ds = gdal.Open(vrt, gdal.GA_Update)
        if gdal_ds is None:
            raise RuntimeError('unable to open "{}"'.format(vrt))
        driver = gdal_ds.GetDriver()
        if driver.ShortName != 'VRT':
            raise TypeError('unexpected GDAL driver ({}). '
                            'VRT driver expected'.format(driver.ShortName))
    else:
        driver = gdal.GetDriverByName('VRT')
        if driver is None:
            raise RuntimeError('unable to get driver "VRT"')

        gdal_ds = driver.Create(vrt, xsize, ysize, 0)
        if gdal_ds is None:
            raise RuntimeError('unable to create "{}" dataset'.format(vrt))

    filename = os.pathsep.join(product.file_path.split('/'))  # denormalize
    offset = dataset.get_dsd().ds_offset + field.get_offset()
    line_offset = record.tot_size
    pixel_offset = epr.get_data_type_size(field.get_type())

    if band.sample_model == epr.E_SMOD_1OF2:
        pixel_offset *= 2
    elif band.sample_model == epr.E_SMOD_2OF2:
        offset += pixel_offset
        pixel_offset *= 2

    options = [
        'subClass=VRTRawRasterBand',
        'SourceFilename={}'.format(filename),
        'ImageOffset={}'.format(offset),
```

(continues on next page)

(continued from previous page)

```
        'LineOffset={}'.format(line_offset),
        'PixelOffset={}'.format(pixel_offset),
        'ByteOrder=MSB',
    ]

    gtype = epr_to_gdal_type[field.get_type()]
    ret = gdal_ds.AddBand(gtype, options=options)
    if ret != gdal.CE_None:
        raise RuntimeError(
            'unable to add VRTRawRasterBand to "{}"'.format(vrt))

    gdal_band = gdal_ds.GetRasterBand(gdal_ds.RasterCount)
    gdal_band.SetDescription(band.description)
    metadata = {
        'name': band.get_name(),
        'dataset_name': dataset.get_name(),
        'dataset_description': dataset.description,
        'lines_mirrored': str(band.lines_mirrored),
        'sample_model': epr.get_sample_model_name(band.sample_model),
        'scaling_factor': str(band.scaling_factor),
        'scaling_offset': str(band.scaling_offset),
        'scaling_method': epr.get_scaling_method_name(band.scaling_method),
        'spectr_band_index': str(band.spectr_band_index),
        'unit': band.unit if band.unit else '',
        'bm_expr': band.bm_expr if band.bm_expr else '',
    }
    gdal_band.SetMetadata(metadata)

    return gdal_ds


def epr2gdal(product, vrt, overwrite_existing=False):
    if isinstance(product, str):
        filename = product
        product = epr.open(filename)

    ysize = product.get_scene_height()
    xsize = product.get_scene_width()

    if os.path.exists(vrt) and not overwrite_existing:
        raise ValueError('unable to create "{0}". Already exists'.
↪format(vrt))

    driver = gdal.GetDriverByName('VRT')
    if driver is None:
        raise RuntimeError('unable to get driver "VRT"')

    gdal_ds = driver.Create(vrt, xsize, ysize, 0)
    if gdal_ds is None:
        raise RuntimeError('unable to create "{}" dataset'.format(vrt))

    metadata = {
        'id_string': product.id_string,
        'meris_iodd_version': str(product.meris_iodd_version),
        'dataset_names': ','.join(product.get_dataset_names()),
        'num_datasets': str(product.get_num_datasets()),
```

(continues on next page)

(continued from previous page)

```
        'num_dsds': str(product.get_num_dsds()),
    }
    gdal_ds.SetMetadata(metadata)

    mph = product.get_mph()
    metadata = str(mph).replace(' = ', '=').split('\n')
    gdal_ds.SetMetadata(metadata, 'MPH')

    sph = product.get_sph()
    metadata = str(sph).replace(' = ', '=').split('\n')
    gdal_ds.SetMetadata(metadata, 'SPH')

    for band in product.bands():
        epr2gdal_band(band, gdal_ds)

    # @TODO: set geographic info

    return gdal_ds


if __name__ == '__main__':
    filename = 'MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_
↪0001.N1'
    vrtfilename = os.path.splitext(filename)[0] + '.vrt'

    gdal_ds = epr2gdal(filename, vrtfilename)

    with epr.open(filename) as product:
        band_index = product.get_band_names().index('water_vapour')
        band = product.get_band('water_vapour')
        eprdata = band.read_as_array()
        unit = band.unit
        lines_mirrored = band.lines_mirrored
        scaling_offset = band.scaling_offset
        scaling_factor = band.scaling_factor

    gdal_band = gdal_ds.GetRasterBand(band_index + 1)
    vrtdata = gdal_band.ReadAsArray()

    if lines_mirrored:
        vrtdata = vrtdata[:, ::-1]

    vrtdata = vrtdata * scaling_factor + scaling_offset

    print('Max absolute error:', abs(vrtdata - eprdata).max())

    # plot
    from matplotlib import pyplot as plt

    plt.figure()
    plt.subplot(2, 1, 1)
    plt.imshow(eprdata)
    plt.grid(True)
    cb = plt.colorbar()
    cb.set_label(unit)
    plt.title('EPR data')
```

(continues on next page)

(continued from previous page)

```
    plt.subplot(2, 1, 2)
    plt.imshow(vrtdata)
    plt.grid(True)
    cb = plt.colorbar()
    cb.set_label(unit)
    plt.title('VRT data')
    plt.show()
```

CHAPTER 3

API Reference

PyEPR provides Python bindings for the ENVISAT Product Reader C API (EPR API) for reading satellite data from ENVISAT ESA (European Space Agency) mission.

PyEPR is fully object oriented and, as well as the EPR API for C, supports ENVISAT MERIS, AATSR Level 1B and Level 2 and also ASAR data products. It provides access to the data either on a geophysical (decoded, ready-to-use pixel samples) or on a raw data layer. The raw data access makes it possible to read any data field contained in a product file.

3.1 Classes

3.1.1 Product

class epr.**Product**

ENVISAT product.

The Product class provides methods and properties to get information about an ENVISAT product file.

See also:

open()

Attributes

file_path

The file's path including the file name.

mode
String that specifies the mode in which the file is opened.

Possible values: *rb* for read-only mode, *rb+* for read-write mode.

id_string
The product identifier string obtained from the MPH parameter 'PRODUCT'.

The first 10 characters of this string identify the product type, e.g. "MER_1P__FR" for a MERIS Level 1b full resolution product. The rest of the string decodes product instance properties.

meris_iodd_version
For MERIS L1b and RR and FR to provide backward compatibility.

tot_size
The total size in bytes of the product file.

Methods

get_band(*name*)
Gets the band corresponding to the specified name.

Parameters **name** – the name of the band

Returns the requested *Band* instance, or raises a *EPRValueError* if not found

get_band_at(*index*)
Gets the *Band* at the specified position within the `product`.

Parameters **index** – the index identifying the position of the *Band*, starting with 0, must not be negative

Returns the requested *Band* instance, or raises a *EPRValueError* if not found

get_dataset(*name*)
Gets the *Dataset* corresponding to the specified dataset name.

Parameters **name** – the *Dataset* name

Returns the requested *Dataset* instance

get_dataset_at(*index*)
Gets the *Dataset* at the specified position within the *Product*.

Parameters **index** – the index identifying the position of the *Dataset*, starting with 0, must not be negative

Returns the requested *Dataset*

get_dsd_at(*index*)
Gets the *DSD* at the specified position.

Gets the *DSD* (*Dataset* descriptor) at the specified position within the *Product*.

Parameters **index** – the index identifying the position of the *DSD*, starting with 0, must not be negative

Returns the requested *DSD* instance

get_num_bands()
Gets the number of all *Band*s contained in a *Product*.

get_num_datasets()
Gets the number of all *Dataset*s contained in a *Product*.

get_num_dsds()
Gets the number of all *DSD*s (*Dataset* descriptors) contained in the *Product*.

get_scene_height()
Gets the *Product* scene height in pixels.

get_scene_width()
Gets the *Product* scene width in pixels.

get_mph()
The *Record* representing the main product header (MPH).

get_sph()
The *Record* representing the specific product header (SPH).

read_bitmask_raster(*bm_expr*, *xoffset*, *yoffset*, *raster*)

Calculates a bit-mask raster.

Calculates a bit-mask, composed of flags of the given *Product* and combined as described in the given bit-mask expression, for the a certain dimension and sub-sampling as defined in the given raster.

param bm_expr a string holding the logical expression for the definition of the bit-mask. In a bit-mask expression, any number of the flag-names (found in the DDDB) can be composed with "(", ")", "NOT", "AND", "OR". Valid bit-mask expression are for example `flags.LAND OR flags.CLOUD` or `NOT flags.WATER AND flags.TURBID_S`

param xoffset across-track co-ordinate in pixel co-ordinates (zero-based) of the upper right corner of the source-region

param yoffset along-track co-ordinate in pixel co-ordinates (zero-based) of the upper right corner of the source-region

param raster the raster for the bit-mask. The data type of the raster must be either *E_TID_UCHAR* or *E_TID_CHAR*

returns zero for success, an error code otherwise

See also:

create_bitmask_raster().

close()
Closes the *Product* product and free the underlying file descriptor.

This method has no effect if the *Product* is already closed. Once the *Product* is closed, any operation on it will raise a ValueError.

As a convenience, it is allowed to call this method more than once; only the first call, however, will have an effect.

flush()
Flush the file stream.

High level interface methods

Note: the following methods are part of the *high level* Python API and do not have any corresponding function in the C API.

---

closed
True if the *Product* is closed.

get_dataset_names()
Return the list of names of the *Dataset*s in the *Product*.

get_band_names()
Return the list of names of the *Band*s in the *Product*.

datasets()
Return the list of *Dataset*s in the *Product*.

bands()
Return the list of *Band*s in the *Product*.

Special methods

The *Product* class provides a custom implementation of the following *special methods*:

- __repr__
- __str__
- __enter__
- __exit__

3.1.2 Dataset

class epr.**Dataset**
ENVISAT dataset.

The Dataset class contains information about a dataset within an ENVISAT product file which has been opened with the *open()* function.

A new Dataset instance can be obtained with the *Product.get_dataset()* or *Product.get_dataset_at()* methods.

Attributes

description
A short description of the *Band* contents.

product
The *Product* instance to which this *Dataset* belongs to.

Methods

get_name()
Gets the name of the *Dataset*.

get_dsd()
Gets the *Dataset* descriptor (*DSD*).

get_dsd_name()
Gets the name of the *DSD* (*Dataset* descriptor).

get_num_records()
Gets the number of *Record*s of the *Dataset*.

create_record()
Creates a new *Record*.

Creates a new, empty *Record* with a structure compatible with the *Dataset*. Such a *Record* is typically used in subsequent calls to *Dataset.read_record()*.

Returns the new *Record* instance

read_record(*index*[, *record*])
Reads specified *Record* of the *Dataset*.

The *Record* is identified through the given zero-based *Record* index. In order to reduce memory reallocation, a *Record* (pre-)created by the method *Dataset.create_record()* can be passed to this method. Data is then read into this given *Record*.

If no *Record* (None) is given, the method initiates a new one.

In both cases, the *Record* in which the data is read into will be returned.

Parameters

- **index** – the zero-based *Record* index (default: 0)

- **record** – a pre-created *Record* to reduce memory reallocation, can be None (default) to let the function allocate a new *Record*

Returns the record in which the data has been read into or raises an exception (*EPRValueError*) if an error occurred

Changed in version 0.9: The *index* parameter now defaults to zero.

High level interface methods

Note: the following methods are part of the *high level* Python API and do not have any corresponding function in the C API.

---

records()
Return the list of *Record*s contained in the *Dataset*.

Special methods

The *Dataset* class provides a custom implementation of the following *special methods*:

- __repr__
- __str__
- __iter__

3.1.3 Record

class epr.**Record**
Represents a record read from an ENVISAT dataset.

A record is composed of multiple fields.

See also:

Field

Attributes

dataset_name
The name of the *Dataset* to which this *Record* belongs to.

New in version 0.9.

tot_size
The total size in bytes of the *Record*.

It includes all data elements of all *Field*s of a *Record* in a *Product* file.

tot_size is a derived variable, it is computed at run-time and not stored in the DSD-DB.

New in version 0.9.

index

Index of the *Record* within the *Dataset*.

It is *None* for empty *Record*s (created with *Dataset.create_record()* but still not read) and for *MPH* (see *Product.get_mph()*) and *SPH* (see *Product.get_sph()*) *Record*s.

See also:

Dataset.read_record()

New in version 0.9.

Methods

get_field(*name*)

Gets a *Field* specified by name.

The *Field* is here identified through the given name. It contains the *Field* info and all corresponding values.

Parameters **name** – the the name of required *Field*

Returns the specified *Field* or raises an exception (*EPRValueError*) if an error occurred

get_field_at(*index*)

Gets a *Field* at the specified position within the *Record*.

Parameters **index** – the zero-based index (position within *Record*) of the *Field*

Returns the *Field* or raises and exception (*EPRValueError*) if an error occurred

get_num_fields()

Gets the number of *Field*s contained in the *Record*.

print_([*ostream*])

Write the *Record* to specified file (default: `sys.stdout`).

This method writes formatted contents of the *Record* to specified *ostream* text file or (default) the ASCII output is be printed to standard output (`sys.stdout`).

Parameters **ostream** – the (opened) output file object

Note: the *ostream* parameter have to be a *real* file not a generic stream object like `StringIO.StringIO` instances.

---

print_element (*field_index*, *element_index*[, *ostream*])
Write the specified field element to file (default: `sys.stdout`).

This method writes formatted contents of the specified *Field* element to the *ostream* text file or (default) the ASCII output will be printed to standard output (`sys.stdout`).

Parameters

- **field_index** – the index of *Field* in the *Record*
- **element_index** – the index of element in the specified *Field*
- **ostream** – the (opened) output file object

Note: the *ostream* parameter have to be a *real* file not a generic stream object like `StringIO.StringIO` instances.

---

get_offset ()
Record offset in bytes within the *Dataset*.

New in version 0.9.

High level interface methods

Note: the following methods are part of the *high level* Python API and do not have any corresponding function in the C API.

---

get_field_names ()
Return the list of names of the *Field*s in the *Record*.

fields ()
Return the list of *Field*s contained in the *Record*.

Special methods

The *Record* class provides a custom implementation of the following *special methods*:

- __repr__
- __str__
- __iter__

3.1.4 Field

class epr.**Field**
Represents a field within a record.

A *Field* is composed of one or more data elements of one of the types defined in the internal `field_info` structure.

See also:

Record

Attributes

tot_size

The total size in bytes of all data elements of a *Field*.

tot_size is a derived variable, it is computed at run-time and not stored in the DSD-DB.

New in version 0.9.

get_description()
Gets the description of the *Field*.

get_name()
Gets the name of the *Field*.

get_num_elems()
Gets the number of elements of the *Field*.

get_type()
Gets the type of the *Field*.

get_unit()
Gets the unit of the *Field*.

get_elem([*index*])
Field single element access.

This function is for getting the elements of a *Field*.

Parameters **index** – the zero-based index of element to be returned, must not be negative. Default: 0.

Returns the typed value from given *Field*

get_elems()
Field array element access.

This function is for getting an array of field elements of the *Field*.

Returns the data array (`numpy.ndarray`) having the type of the *Field*

Changed in version 0.9: the returned `numpy.ndarray` shares the data buffer with the C `Field` structure so any change in its contents is also reflected to the `Filed` object

set_elem(*elem*[, *index*])
Set *Field* array element.

This function is for setting an array of field element of the *Field*.

Parameters

- **elem** – value of the element to set
- **index** – the zero-based index of element to be set, must not be negative. Default: 0.

Note: this method does not have any corresponding function in the C API.

New in version 0.9.

set_elems(*elems*)
Set *Field* array elements.

This function is for setting an array of *Field* elements of the *Field*.

Parameters **elems** – np.ndarray of elements to set

Note: this method does not have any corresponding function in the C API.

New in version 0.9.

print_([*ostream*])
Write the *Field* to specified file (default: `sys.stdout`).

This method writes formatted contents of the *Field* to specified *ostream* text file or (default) the ASCII output is be printed to standard output (`sys.stdout`).

Parameters **ostream** – the (opened) output file object

Note: the *ostream* parameter have to be a *real* file not a generic stream object like `StringIO.StringIO` instances

---

get_offset()
Field offset in bytes within the *Record*.

New in version 0.9.

Special methods

The *Field* class provides a custom implementation of the following *special methods*:

- __repr__
- __str__

- __eq__
- __ne__
- __len__[1]

3.1.5 DSD

class epr.**DSD**

Dataset descriptor.

The DSD class contains information about the properties of a *Dataset* and its location within an ENVISAT *Product* file.

Attributes

ds_name
The *Dataset* name.

ds_offset
The offset of *Dataset* in the *Product* file.

ds_size
The size of *Dataset* in the *Product* file.

ds_type
The *Dataset* type descriptor.

dsr_size
The size of dataset record for the given *Dataset* name.

filename
The filename in the DDDB with the description of this *Dataset*.

index
The index of this *DSD* (zero-based).

num_dsr
The number of dataset records for the given *Dataset* name.

Special methods

The *DSD* class provides a custom implementation of the following *special methods*:

- __repr__
- __eq__
- __ne__

[1] if the field is a *E_TID_STRING* field then the `__len__()` method returns the string length, otherwise the number of elements of the field is returned (same as *Field.get_num_elems()*)

3.1.6 Band

class epr.**Band**

The band of an ENVISAT *Product*.

The Band class contains information about a band within an ENVISAT *Product* file which has been opened with the *open()* function.

A new Band instance can be obtained with the *Product.get_band()* method.

Attributes

bm_expr

A bit-mask expression used to filter valid pixels.

All others are set to zero.

data_type

The data type of the *Band* pixels.

Possible values are:

- `*` –> the datatype remains unchanged.
- `uint8_t` –> 8-bit unsigned integer
- `uint32_t` –> 32-bit unsigned integer
- `Float` –> 32-bit IEEE floating point

description

A short description of the *Band* contents.

lines_mirrored

Mirrored lines flag.

If true (=1) lines will be mirrored (flipped) after read into a raster in order to ensure a pixel ordering in raster X direction from WEST to EAST.

product

The *Product* instance to which this *Band* belongs to.

sample_model

The sample model operation.

The sample model operation applied to the source *Dataset* for getting the correct samples from the MDS (for example MERIS L2).

Possible values are:

- `*` –> no operation (direct copy)
- `1OF2` –> first byte of 2-byte interleaved MDS
- `2OF2` –> second byte of 2-byte interleaved MDS
- `0123` –> combine 3-bytes interleaved to 4-byte integer

scaling_factor
The scaling factor.

Possible values are:

- `*` –> no factor provided (implies scaling_method=*)
- `const` –> a floating point constant
- `GADS.field[.field2]` –> value is provided in global annotation *Dataset* with name *GADS* in *Field* *field*. Optionally a second element index for multiple-element fields can be given too

scaling_method
The scaling method which must be applied to the raw source data in order to get the 'real' pixel values in geo-physical units.

Possible values are:

- `*` –> no scaling applied
- `Linear_Scale` –> linear scaling applied:

```
y = offset + scale * x
```

- `Log_Scale` –> logarithmic scaling applied:

```
y = log10(offset + scale * x)
```

scaling_offset
Possible values are:

- `*` –> no offset provided (implies scaling_method=*)
- `const` –> a floating point constant
- `GADS.field[.field2]` ``--> value is provided in global annotation :class:`Dataset` with name ``GADS`` in *Field* `field`. Optionally a second element index for multiple-element fields can be given too

spectr_band_index
The (zero-based) spectral *Band* index.

-1 if this is not a spectral *Band*.

unit
The geophysical unit for the *Band* pixel values.

dataset
The source *Dataset*.

The source *Dataset* containing the raw data used to create the *Band* pixel values.

New in version 0.9.

Methods

get_name()
Gets the name of the *Band*.

create_compatible_raster([*src_width*, *src_height*, *xstep*, *ystep*])
Creates a *Raster* which is compatible with the data type of the *Band*.

The created *Raster* is used to read the data in it (see *Band.read_raster()*).

The *Raster* is defined on the grid of the *Product*, from which the data are read. Spatial subsets and under-sampling are possible) through the parameter of the method.

A *Raster* is an object that allows direct access to data of a certain portion of the ENVISAT *Product* that are read into the it. Such a portion is called the source. The complete ENVISAT *Product* can be much greater than the source. One can move the *Raster* over the complete ENVISAT *Product* and read in turn different parts (always of the size of the source) of it into the *Raster*. The source is specified by the parameters *height* and *width*.

A typical example is a processing in blocks. Lets say, a block has 64x32 pixel. Then, my source has a width of 64 pixel and a height of 32 pixel.

Another example is a processing of complete image lines. Then, my source has a widths of the complete product (for example 1121 for a MERIS RR product), and a height of 1). One can loop over all blocks read into the *Raster* and process it.

In addition, it is possible to defined a sub-sampling step for a *Raster*. This means, that the source is not read 1:1 into the *Raster*, but that only every 2nd or 3rd pixel is read. This step can be set differently for the across track (source_step_x) and along track (source_step_y) directions.

Parameters

- **src_width** – the width (across track dimension) of the source to be read into the *Raster*. Default: scene width (see *Product.get_scene_width*)
- **src_height** – the height (along track dimension) of the source to be read into the *Raster*. Default: scene height (see *Product.get_scene_height*)
- **xstep** – the sub-sampling step across track of the source when reading into the *Raster*. Default: 1.
- **ystep** – the sub-sampling step along track of the source when reading into the *Raster*. Default: 1.

Returns the new *Raster* instance or raises an exception (*EPRValueError*) if an error occurred

---

Note: *src_width* and *src_height* are the dimantion of the of the source area. If one specifies a *step* parameter the resulting *Raster* will have a size that is smaller that

the specifies source size:

```
raster_size = src_size // step
```

read_raster ([*xoffset*, *yoffset*, *raster*])

Reads (geo-)physical values of the *Band* of the specified source-region.

The source-region is a defined part of the whole ENVISAT *Product* image, which shall be read into a *Raster*. In this routine the co-ordinates are specified, where the source-region to be read starts. The dimension of the region and the sub-sampling are attributes of the *Raster* into which the data are read.

Parameters

- **xoffset** – across-track source co-ordinate in pixel co-ordinates (zero-based) of the upper right corner of the source-region. Default 0.
- **yoffset** – along-track source co-ordinate in pixel co-ordinates (zero-based) of the upper right corner of the source-region. Default 0.
- **raster** – *Raster* instance set with appropriate parameters to read into. If not provided a new *Raster* is instantiated

Returns the *Raster* instance in which data are read

This method raises an instance of the appropriate *EPRError* sub-class if case of errors

See also:

Band.create_compatible_raster() and *create_raster()*

High level interface methods

Note: the following methods are part of the *high level* Python API and do not have any corresponding function in the C API.

read_as_array ([*width*, *height*, *xoffset*, *yoffset*, *xstep*, *ystep*])

Reads the specified source region as an `numpy.ndarray`.

The source-region is a defined part of the whole ENVISAT *Product* image, which shall be read into a *Raster*. In this routine the co-ordinates are specified, where the source-region to be read starts. The dimension of the region and the sub-sampling are attributes of the *Raster* into which the data are read.

Parameters

- **src_width** – the width (across track dimension) of the source to be read into the *Raster*. If not provided reads as much as possible

- **src_height** – the height (along track dimension) of the source to be read into the *Raster*, If not provided reads as much as possible
- **xoffset** – across-track source co-ordinate in pixel co-ordinates (zero-based) of the upper right corner of the source-region. Default 0.
- **yoffset** – along-track source co-ordinate in pixel co-ordinates (zero-based) of the upper right corner of the source-region. Default 0.
- **xstep** – the sub-sampling step across track of the source when reading into the *Raster*. Default: 1
- **ystep** – the sub-sampling step along track of the source when reading into the *Raster*. Default: 1

Returns the `numpy.ndarray` instance in which data are read

This method raises an instance of the appropriate *EPRError* sub-class if case of errors

See also:

Band.create_compatible_raster(), *create_raster()* and *Band.read_raster()*

Special methods

The *Band* class provides a custom implementation of the following *special methods*:

- __repr__

3.1.7 Raster

class epr.**Raster**

Represents a raster in which data will be stored.

All 'size' parameter are in PIXEL.

Attributes

data_type

The data type of the *Band* pixels.

All `E_TID_*` types are possible.

source_height

The height of the source.

source_width

The width of the source.

source_step_x
The sub-sampling for the across-track direction in pixel.

source_step_y
The sub-sampling for the along-track direction in pixel.

High level interface attributes

Note: the following attributess are part of the *high level* Python API and do not have a counterpart in the C API.

---

data
Raster data exposed as `numpy.ndarray` object.

Note: this property shares the data buffer with the *Raster* object so any change in its contents is also reflected to the *Raster* object

---

Note: the *Raster* objects do not have a *Field* named *data* in the corresponding C structure. The *EPR_SRaster* C structure have a *Field* named *buffer* that is a raw pointer to the data buffer and it is not exposed as such in the Python API.

Methods

get_pixel (*x*, *y*)
Single pixel access.

This function is for getting the values of the elements of a *Raster* (i.e. pixel)

Parameters

- **x** – the (zero-based) X coordinate of the pixel
- **y** – the (zero-based) Y coordinate of the pixel

Returns the typed value at the given co-ordinate

get_elem_size()
The size in byte of a single element (sample) of this *Raster* buffer.

get_height()
Gets the *Raster* height in pixels.

get_width()
Gets the *Raster* width in pixels.

Special methods

The *Raster* class provides a custom implementation of the following *special methods*:

- __repr__

3.1.8 EPRTime

class epr.**EPRTime**

Convenience class for time data exchange.

EPRTime is a `collections.namedtuple` with the following fields:

days

seconds

microseconds

3.2 Functions

epr.**open**(*filename*, *mode='rb'*)

Open the ENVISAT product.

Opens the ENVISAT *Product* file with the given file path, reads MPH, SPH and all *DSD*s, organized the table with parameter of line length and tie points number.

Parameters

- **product_file_path** – the path to the ENVISAT *Product* file
- **mode** – string that specifies the mode in which the file is opened. Allowed values: *rb* for read-only mode, *rb+* for read-write mode. Default: mode='rb'.

Returns the *Product* instance representing the specified product. An exception (`exceptions.ValueError`) is raised if the file could not be opened.

The *Product* class supports context management so the recommended way to ensure that a product is actually closed as soon as a task is completed is to use the `with` statement:

```
with open('ASA_IMP_1PNUPA20060202_ ... _3110.N1') as product:
    dataset = product.get_dataset('MAIN_PROCESSING_PARAMS_ADS')
    record = dataset.read_record(0)
    print(record)
```

epr.**data_type_id_to_str**(*type_id*)

Gets the 'C' data type string for the given data type.

epr.**get_data_type_size**(*type_id*)
Gets the size in bytes for an element of the given data type.

epr.**get_numpy_dtype**(*type_id*)
Return the numpy data-type specified EPR type ID.

New in version 0.9.

epr.**get_sample_model_name**(*model*)
Return the name of the specified sample model.

epr.**get_scaling_method_name**(*method*)
Return the name of the specified scaling method.

epr.**create_raster**(*data_type*, *src_width*, *src_height*[, *xstep*, *ystep*])
Creates a *Raster* of the specified data type.

This function can be used to create any type of raster, e.g. for later use as a bit-mask.

Parameters

- **data_type** – the type of the data to stored in the *Raster*, must be one of E_TID_*.

  See also:

  Data type Identifiers

- **src_width** – the width (across track dimension) of the source to be read into the *Raster*. See description of *Band.create_compatible_raster()*
- **src_height** – the height (along track dimension) of the source to be read into the *Raster*. See description of *Band.create_compatible_raster()*
- **xstep** – the sub-sampling step across track of the source when reading into the *Raster*. Default: 1.
- **ystep** – the sub-sampling step along track of the source when reading into the *Raster*. Default: 1.

Returns the new *Raster* instance

See also:

description of *Band.create_compatible_raster()*

epr.**create_bitmask_raster**(*src_width*, *src_height*[, *xstep*, *ystep*])
Creates a *Raster* to be used for reading bitmasks.

The *Raster* returned always is of type `byte`.

Parameters

- **src_width** – the width (across track dimension) of the source to be read into the *Raster*

- **src_height** – the height (along track dimension) of the source to be read into the *Raster*
- **xstep** – the sub-sampling step across track of the source when reading into the *Raster*. Default: 1.
- **ystep** – the sub-sampling step along track of the source when reading into the *Raster*. Default: 1.

Returns the new *Raster* instance or raises an exception (*EPRValueError*) if an error occurred

See also:

the description of *Band.create_compatible_raster()*

3.3 Exceptions

3.3.1 EPRError

exception epr.**EPRError**
EPR API error.

code
EPR API error code.

__init__(*[message[, code, *args, **kwargs]]*)
Initializer.

Parameters **message** – error message

Pram code EPR error code

3.3.2 EPRValueError

exception epr.**EPRValueError**
Inherits both *EPRError* and standard exceptions.ValueError.

3.4 Data

epr.**__version__**
Version string of PyEPR.

epr.**EPR_C_API_VERSION**
Version string of the wrapped EPR API C library.

3.4.1 Data type identifiers

epr.**E_TID_UNKNOWN**

epr.**E_TID_UCHAR**

epr.**E_TID_CHAR**

epr.**E_TID_USHORT**

epr.**E_TID_SHORT**

epr.**E_TID_UINT**

epr.**E_TID_INT**

epr.**E_TID_FLOAT**

epr.**E_TID_DOUBLE**

epr.**E_TID_STRING**

epr.**E_TID_SPARE**

epr.**E_TID_TIME**

3.4.2 Sample Models

epr.**E_SMOD_1OF1**

epr.**E_SMOD_1OF2**

epr.**E_SMOD_2OF2**

epr.**E_SMOD_3TOI**

epr.**E_SMOD_2TOF**

3.4.3 Scaling Methods

epr.**E_SMID_NON**
: No scaling.

epr.**E_SMID_LIN**
: Linear pixel scaling.

epr.**E_SMID_LOG**
: Logarithmic pixel scaling.

CHAPTER 4

Change history

4.1 PyEPR 0.9.5 (23/08/2018)

- Fix compatibility with numpy >= 1.14: `np.fromstring()` is deprecated.
- Update the pypi sidebar in the documentation
- Use *.rst* extension for doc source files
- Fix setup script to not use system libs if epr-api sources are available
- Do not access fields of bands after that the product has been closed (fix a segmentation fault on windows)
- unittest2 is now required for Python < 3.4

4.2 PyEPR 0.9.4 (29/04/2018)

- Fix compatibility with cython >= 0.28
- PyEPR has been successfully tested with PyPy

4.3 PyEPR 0.9.3 (02/05/2015)

- Fix PyEprExtension class in setup.py (closes gh-11)
- Updated internal EPR API version

4.4 PyEPR 0.9.2 (08/03/2015)

- Improved string representation of fields in case of `E_TID_STRING` data type. Now bytes are decoded and represented as Python strings.
- New tutorial *GDAL export example*
- Improved "Installation" and "Testing" sections of the user manual

4.5 PyEPR 0.9.1 (27/02/2015)

- Fix source distribution (missing EPR API C sources)

4.6 PyEPR 0.9 (27/02/2015)

- basic support for update mode: products can now be opened in update mode ('rb+') and it is possible to call *epr.Field.set_elem()* and *epr.Field.set_elems()* methods to set *epr.Field* elements changing the contents of the *epr.Product* on disk. This feature is not available in the EPR C API.
- new functions/methods and properties:
  - *epr.Record.index* property: returns the index of the *epr.Record* within the *epr.Dataset*
  - *epr.Band.dataset* property: returns the source *epr.Dataset* object containing the raw data used to create the *epr.Band*'s pixel values
  - `epr.Band._field_index` and `epr.Band._elem_index` properties: return the *epr.Field* index (within the *epr.Record*) and the element index (within the *epr.Field*) containing the raw data used to create the *epr.Band*'s pixel values
  - *epr.Record.dataset_name* property: returns the name of the *epr.Dataset* from which the `Record` has bee read
  - *epr.Record.tot_size* and *epr.Field.tot_size* properties: return the total size in bytes of the *epr.Record* and *epr.Field* respectively
  - *epr.get_numpy_dtype()* function: retrieves the numpy data type corresponding to the specified EPR type ID
  - added support for some low level feature: the *_magic* private attribute stores the identifier of EPR C stricture, the *epr.Record.get_offset()* returns the offset in bytes of the *epr.Record* within the file, and the *epr.Field.get_offset()* method returns the *epr.Field* offset within the *epr.Record*
- improved functions/methods:

  - *epr.Field.get_elems()* now also handles *epr.E_TID_STRING* and *epr.E_TID_TIME* data types
  - improved *epr.get_data_type_size()*, *epr.data_type_id_to_str()*, *epr.get_scaling_method_name()* and *epr.get_sample_model_name()* functions that are now defined using the cython *cpdef* directive
  - the *epr.Field.get_elems()* method has been re-written to remove loops and unnecessary data copy
  - now generator expressions are used to implement *__iter__* special methods
- the *index* parameter of the *epr.Dataset.read_record()* method is now optional (defaults to zero)
- the deprecated *__revision__* variable has been removed
- declarations of the EPR C API have been moved to the new `epr.pyd`
- the *const_char* and *const_void* definitions have been dropped, no longer necessary with cython >= 0.19
- minimum required version for cython is now 0.19
- the `setup.py` script has been completely rewritten to be more "pip friendly". The new script uses setuptools if available and functions that use numpy are evaluated lazily so to give a chance to pip and setuptools to install dependencies, numpy, before they are actually used. This should make PyEPR "pip-installable" even on system there numpy is not already installed.
- the `test` directory has been renamed into `tests`
- the test suite now has a `setUpModule()` function that automatically downloads the ENVISAT test data required for test execution. The download only happens if the test dataset is not already available.
- tests can now be run using the `setup.py` script:

```
$ python3 setup.py test
```

- enable continuous integration and testing in for Windows using AppVeyor (32bit only)
- status badges for AppVeyor CI and PyPI added to the HTML doc index

4.7 PyEPR 0.8.2 (03/08/2014)

- fixed segfault caused by incorrect access to *epr.Dataset.description* string in case of closed products
- fixed a memory leak in *epr.Raster* (closes gh-10)

- the size parameters (*src_width* and *src_height*) in *epr.Band.create_compatible_raster()* are now optional. By default a *epr.Raster* with the same size of the scene is created
- the test suite have been improved
- improved the *NDVI computation example*
- updates sphinx config
- small clarification in the *Installation* section of the *User Manual*.
- EPR C API (version bundled with the official source tar-ball)
  - in case of error always free resources before setting the error code. This avoids error shadowing in some cases.
  - fixed a bug that caused reading of the incorrect portion of data in case of mirrored annotation datasets (closes gh-9)
  - fixed a bug that caused incorrect data sub-sampling in case of mirrored datasets

4.8 PyEPR 0.8.1 (07/09/2013)

- fixed an important bug in the error checking code introduced in previous release (closes gh-8)
- fixed the NDVI example
- no more display link URL in footnotes of the PDF User Manual

4.9 PyEPR 0.8 (07/09/2013)

- now the *epr.Product* objects have a *epr.Product.close()* method that can be used to explicitly close products without relying on the garbage collector behaviour (closes gh-7)
- new *epr.Product.closed* (read-only) attribute that can be used to check if a *epr.Product* has been closed
- the `Product` class now supports context management so they can be used in `with` statements
- added entries for *epr.__version__* and `epr.__revision__` in the reference manual
- the `epr.__revision__` module attribute is now deprecated
- some *cythonization* warnings have been fixed
- several small improvements to the documentation

4.10 PyEPR 0.7.1 (19/08/2013)

- fixed potential issues with conversion from python strings to `char*`
- new snapshot of the EPR C API sources (2.3dev):
  - the size of the record tables has been fixed
  - the EPR_NUM_PRODUCT_TABLES has been fixed
  - fixed a missing prototype
  - several GCC warnings has been silenced
  - additional checks on return codes
  - now and error is raised when an invalid flag name is used
- better factorization of Python 3 specific code
- use the *CLOUD* flag instead of *BRIGHT* in unit tests
- added function/method signature to all doc-strings for better interactive help
- several improvements to the documentation:
  - updated the `README.txt` file to mention EPR C API sourced inclusion in the PyEPR 0.7 (and lates) source tar-ball
  - small fix in the installation instructions: the pip tool does not have a "–prefix" parameter
  - always use the python3 syntax for the *print* function in all examples in the documentation
  - links to older (and dev) versions of the documentation have been added in the man page of the HTML doc
  - removed *date* form the doc meta-data. The documentation build date is reported in the front page of the LaTeX (PDF) doc and, starting from this release, in the footer of the HTML doc.
  - the Ohloh widget has been added in the sidebar of the HTML doc
  - improved the regexp for detecting the SW version in the :file'setup.py' script
  - formatting

4.11 PyEPR 0.7 (04/08/2013)

- more detailed error messages in case of open failures
- new sphinx theme for the HTML documentation
- Travis-CI has been set-up for the project

- now the source tar-ball also includes a copy of the EPR C API sources so that no external C library is required to build PyEPR.

  This features also makes it easier to install PyEPR using pip.

  The user can still guild PyEPR against a system version of the ERP-API library simply using the *–epr-api-src* option of the `setup.py` script with "None"" as value.

  The ERP C API included in the source tar-ball is version *2.3dev-pyepr062*, a development and patched version that allows the following enhancements.

  - support for ERS products in ENVISAT format
  - support for ASAR products generated with the new ASAR SW version 6.02 (ref. doc. PO-RS-MDA-GS-2009_4/C
  - fix incorrect reading of "incident_angle" bands (closes gh-6). The issue is in the EPR C API.

4.12 PyEPR 0.6.1 (26/04/2012)

- fix compatibility with cython 0.16
- added a new option to the setup script (*–epr-api-src*) to build PyEPR using the EPR-API C sources

4.13 PyEPR 0.6 (12/08/2011)

- full support for Python 3
- improved code highligh in the documentation
- depend from cython >= 0.13 instead of cython >= 0.14.1. Cythonizing `epr.pyx` with Python 3 requires cython >= 0.15

4.14 PyEPR 0.5 (25/04/2011)

- stop using `PyFile_AsFile()` that is no more available in Python 3
- now documentation uses intersphinx capabilities
- code examples added to documentation
- tutorials added to documentation
- the LICENSE.txt file is now included in the source distribution
- the cython construct `with nogil` is now used instead of calling `Py_BEGIN_ALLOW_THREADS()` and `Py_END_ALLOW_THREADS()` directly
- dropped old versions of cython; now cython 0.14.1 or newer is required

- suppressed several constness related warnings

4.15 PyEPR 0.4 (10/04/2011)

- fixed a bug in the `epr.Product.__str__()`, `Dataset.__str__()` and `erp.Band.__repr__()` methods (bad formatting)
- fixed *epr.Field.get_elems()* method for char and uchar data types
- implemented *epr.Product.read_bitmask_raster()*, now the *epr.Product* API is complete
- fixed segfault in *epr.Field.get_unit()* method when the field has no unit
- a smaller dataset is now used for unit tests
- a new tutorial section has been added to the user documentation

4.16 PyEPR 0.3 (01/04/2011)

- version string of the EPR C API is now exposed as module attribute *epr.EPR_C_API_VERSION*
- implemented `__repr__`, `__str__`, `__eq__`, `__ne__` and `__iter__` special methods
- added utility methods (not included in the C API) like:
  - *epr.Record.get_field_names()*
  - *epr.Record.fields()*
  - *epr.Dataset.records()*
  - *epr.Product.get_dataset_names()*
  - *epr.Product.get_band_names()*
  - *epr.Product.datasets()*
  - *epr.Product.bands()*
- fixed a logic error that caused empty messages in custom EPR exceptions

4.17 PyEPR 0.2 (20/03/2011)

- sphinx documentation added
- added docstrings to all method and classes
- renamed some method and parameter in order to avoid redundancies and have a more *pythonic* API

- in case of null pointers a *epr.EPRValueError* is raised
- improved C library shutdown management
- introduced some utility methods to *epr.Product* and *epr.Record* classes

4.18 PyEPR 0.1 (09/03/2011)

Initial release

CHAPTER 5

License

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P