
PyEPR Documentation

Release 0.8.2

Antonio Valentino

August 03, 2014

1	User Manual	3
1.1	Quick start	3
1.2	Requirements	3
1.3	Download	4
1.4	Installation	4
1.5	Testing	5
1.6	Python vs C API	6
1.7	Memory management	6
1.8	Arrays	7
1.9	Enumerators	7
1.10	Error handling and logging	8
1.11	Library initialization	8
1.12	High level API	8
1.13	Special methods	9
2	Tutorials	11
2.1	Interactive use of PyEPR	11
2.2	Exporting band data	19
2.3	Exporting bitmasks	22
2.4	NDVI computation	25
3	API Reference	31
3.1	Classes	31
3.2	Functions	46
3.3	Exceptions	48
3.4	Data	48
4	Change history	51
4.1	PyEPR 0.8.2 (03/08/2014)	51
4.2	PyEPR 0.8.1 (07/09/2013)	51
4.3	PyEPR 0.8 (07/09/2013)	52
4.4	PyEPR 0.7.1 (19/08/2013)	52
4.5	PyEPR 0.7 (04/08/2013)	53
4.6	PyEPR 0.6.1 (26/04/2012)	53

4.7	PyEPR 0.6 (12/08/2011)	53
4.8	PyEPR 0.5 (25/04/2011)	54
4.9	PyEPR 0.4 (10/04/2011)	54
4.10	PyEPR 0.3 (01/04/2011)	54
4.11	PyEPR 0.2 (20/03/2011)	55
4.12	PyEPR 0.1 (09/03/2011)	55
5	License	57

List of Figures

2.1	Image data read from the “proc_data” band	17
-----	---	----

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

Author Antonio Valentino

Contact antonio.valentino@tiscali.it

Copyright 2011-2014, Antonio Valentino

Version 0.8.2

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()
```

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
- numpy >= 1.5.0

- [EPR API](#) ≥ 2.2 (optional, since PyEPR 0.7 the source tar-ball comes with a copy of the PER C API sources)
- a reasonably updated C compiler ¹ (build only)
- [Cython](#) ≥ 0.13 ² (optional and build only)
- [unittest2](#) (only required for Python < 2.7)

Note: in order to build [PyEPR](#) for [Python3](#) it is required [Cython](#) ≥ 0.15

1.3 Download

Official source tar-balls can be downloaded from [PyPi](#):

<https://pypi.python.org/pypi/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](#) or [easy_install](#):

```
$ pip install numpy pyepr
```

Note: the `setup.py` script does not use [easy_install](#) specific functions so it is unable to handle dependencies automatically.

Also the `setup.py` script uses [numpy](#) to retrieve the path of headers and libraries. For this reason [numpy](#) must be already installed when `setup.py` is executed.

In the above example the required [numpy](#) package is explicitly included in the list of packages to be installed.

See also:

[Requirements](#)

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](#)).

PyEPR uses the standard [Python distutils](#) so it can be installed from sources using the following command:

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

For a user specific installation use:

```
$ python setup.py install --user
```

To install [PyEPR](#) in a non-standard path:

```
$ python setup.py install --prefix=<TARGET_PATH>
```

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

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` of PyEPR 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 but in order to run it the `ENVISAT` sample product used for testing `MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_0001.N1` have to be downloaded from the [ESA](#) website, saved in the `test` directory and decompressed.

On GNU Linux platforms the following shell commands can be used:

```
$ cd pyepr-0.X/test
$ wget http://earth.esa.int/services/sample_products/meris/LRC/L2/\
MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_0001.N1.gz
$ gunzip MER_LRC_2PTGMV20000620_104318_00000104X000_00000_00000_0001.N1.gz
```

After installation the test suite can be run using the following command in the `test` directory:

```
$ python test_all.py
```

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 users have never to worry about memory allocation/deallocation. 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 deallocated 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 *dataset* 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	Python
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 used 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 field1 == field2:
    print('field 1 and field2 are equal')
    print(field1)
else:
    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.

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 [IPython](#) interactive shell an started with the `ipython -pylab` option to enable interactive plotting 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 ipython shell with the following command:

```
$ ipython -pylab
```

one can import the [epr](#) module and start taking confidence with available classes and functions:

```
Python 2.6.6 (r266:84292, Sep 15 2010, 16:22:56)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?                -> Introduction and overview of IPython's features.
%quickref        -> Quick reference.
help             -> Python's own help system.
object?         -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.
```

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

```
In [3]: epr.__version__, epr.EPR_C_API_VERSION
Out[3]: ('0.8.2', '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 `? IPython` shortcut as showed above.

Also `IPython` 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
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
```


<code>epr.E_TID_INT</code>	<code>epr.open</code>
<code>epr.E_TID_SHORT</code>	<code>epr.so</code>
<code>epr.E_TID_SPARE</code>	<code>epr.sys</code>

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
```

```
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',  
 '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
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
```

<code>band.description</code>	<code>band.scaling_method</code>
<code>band.get_name</code>	<code>band.scaling_offset</code>
<code>band.lines_mirrored</code>	<code>band.spectr_band_index</code>
<code>band.product</code>	<code>band.unit</code>
<code>band.read_as_array</code>	

```
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)
```

```
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>
```

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
```

```
-----
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) ()
```

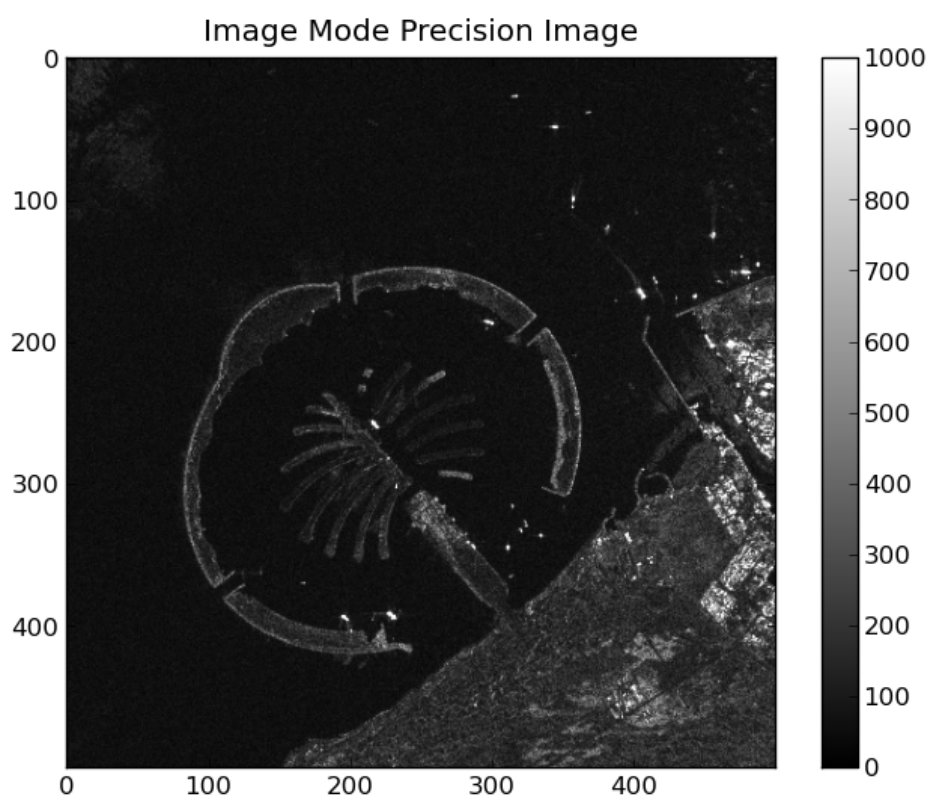


Figure 2.1: Image data read from the “proc_data” band

```
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 underlying C library is opportunely initialized.

```
#!/usr/bin/env python  
  
# 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 is 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::  
  
        $ 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 in-

stantiate 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:

```
1  #!/usr/bin/env python
2
3  # This program is a direct translation of the sample program
4  # "write_bitmask.c" bundled with the EPR-API distribution.
5  #
6  # Source code of the C program is available at:
7  # https://github.com/bcdev/epr-api/blob/master/src/examples/write_bitmask.c
8
9  '''Generates bit mask from ENVISAT flags information as "raw" image
10 for (e.g.) Photoshop
11
12 Call::
13
14     $ python write_bitmask.py <envisat-product> <bitmask-expression>
15     <output-file>
16
17 Example to call the main function::
18
19     $ python write_bitmask.py MER_RR__2P_TEST.N1 \
20     'l2_flags.LAND and !l2_flags.BRIGHT' my_flags.raw
21
22 '''
23
24 from __future__ import print_function
25
26 import sys
27 import epr
28
29
30 def main(*argv):
31     if not argv:
32         argv = sys.argv
33
34     if len(argv) != 4:
35         print('Usage: write_bitmask <envisat-product> <bitmask-expression> '
36             '<output-file>')
37         print(' where envisat-product is the input filename')
38         print(' and bitmask-expression is a string containing the bitmask '
39             'logic')
40         print(' and output-file is the output filename.')
41         print('Example:')
42         print(" MER_RR__2P_TEST.N1 'l2_flags.LAND and not l2_flags.BRIGHT' "
43             "my_flags.raw")
44         print
45         sys.exit(1)
46
47     product_file_path = argv[1]
```

```
48     bm_expr = argv[2]
49     image_file_path = argv[3]
50
51     # Open the product; an argument is a path to product data file
52     with epr.open(product_file_path) as product:
53         offset_x = 0
54         offset_y = 0
55         source_width = product.get_scene_width()
56         source_height = product.get_scene_height()
57         source_step_x = 1
58         source_step_y = 1
59
60         bm_raster = epr.create_bitmask_raster(source_width, source_height,
61                                              source_step_x, source_step_y)
62
63         product.read_bitmask_raster(bm_expr, offset_x, offset_y, bm_raster)
64
65         with open(image_file_path, 'wb') as out_stream:
66             bm_raster.data.tofile(out_stream)
67
68     print('Raw image data successfully written to "%s".' % image_file_path)
69     print('Data type is "byte", size is %d x %d pixels.' % (source_width,
70                                                             source_height))
71
72
73 if __name__ == '__main__':
74     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 from 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 DDDDB) 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.

```
1  #!/usr/bin/env python
2
3  # This program is a direct translation of the sample program
4  # "write_ndvi.c" bundled with the EPR-API distribution.
5  #
6  # Source code of the C program is available at:
7  # https://github.com/bcdev/epr-api/blob/master/src/examples/write_ndvi.c
8
9
10 '''Example for using the epr-api
11
12 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 print_function
```

```
import sys
```

```
import struct
```

```
import logging
```

```
import epr
```

```
def main(*argv):
```

```
    if not argv:
```

```
        argv = sys.argv
```

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

```
        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
```

```

70         offset_y = 0
71
72         logging.info('read "%s" data' % band1_name)
73         band1.read_raster(offset_x, offset_y, raster1)
74
75         logging.info('read "%s" data' % band2_name)
76         band2.read_raster(offset_x, offset_y, raster2)
77
78         # Open the output file
79         logging.info('write ndvi to "%s"' % argv[2])
80         with open(argv[2], 'wb') as out_stream:
81
82             # Loop over all pixel and calculate the NDVI.
83             #
84             # @NOTE: looping over data matrices is not the best solution.
85             # It is done here just for demonstrative purposes
86             for j in range(height):
87                 for i in range(width):
88                     rad1 = raster1.get_pixel(i, j)
89                     rad2 = raster2.get_pixel(i, j)
90                     if (rad1 + rad2) != 0.0:
91                         ndvi = (rad2 - rad1) / (rad2 + rad1)
92                     else:
93                         ndvi = -1.0
94                     out_stream.write(struct.pack('f', ndvi))
95             logging.info('ndvi was written success')
96
97
98 if __name__ == '__main__':
99     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)
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 solution.
#       It is done here just for demonstrative 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* in 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)
```

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

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 bands contained in a product

get_num_datasets ()

Gets the number of all datasets contained in a product

get_num_dsds()

Gets the number of all [DSDs](#) (dataset descriptors) contained in the product

get_scene_height()

Gets the product's scene height in pixels

get_scene_width()

Gets the product's 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.

High level interface methods

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

closed

True if the `Product` is closed.

get_dataset_names()

Return the list of names of the datasets in the product

get_band_names()

Return the list of names of the bands in the product

datasets()

Return the list of dataset in the product

bands()

Return the list of bands 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's 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 records 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
- **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

High level interface methods

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

records()

Return the list of records 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`

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 fields 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

High level interface methods

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

get_field_names ()

Return the list of names of the fields in the product

fields ()

Return the list of fields 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`

`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

`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

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-information the product file

`ds_size`

The size of dataset-information in dataset 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__`

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()`)

- `__ne__`

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's 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's 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:
- $$y = \text{offset} + \text{scale} * x$$
- Log_Scale -> logarithmic scaling applied:

$$y = \log_{10}(\text{offset} + \text{scale} * x)$$

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 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's pixel values

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 correspondent 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’s 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 attributes 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's buffer

get_height ()

Gets the raster's height in pixels

get_width ()

Gets the raster's 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)`

Opens the ENVISAT product

Opens the ENVISAT product file with the given file path, reads MPH, SPH and all DSDs, organized the table with parameter of line length and tie points number.

Parameters `product_file_path` – the path to the ENVISAT product file

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()`

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

`epr.get_data_type_size()`

Gets the size in bytes for an element of the given data type

`epr.get_sample_model_name()`

Return the name of the specified sample model

`epr.get_scaling_method_name()`

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

Param 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.__revision__`

PyEPR revision (currently it is the same as `__version__`)

Deprecated since version 7.2.

`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

Change history

4.1 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

4.2 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

4.3 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

4.4 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 an 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 later) 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* from 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

4.5 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-pyep062*, 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.

4.6 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

4.7 PyEPR 0.6 (12/08/2011)

- full support for [Python 3](#)

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

4.8 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

4.9 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

4.10 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

4.11 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

4.12 PyEPR 0.1 (09/03/2011)

Initial release

License

Copyright (C) 2011-2014 Antonio Valentino <antonio.valentino@tiscali.it>

PyEPR is free software: you can redistribute it and/or modify it under the terms of the [GNU General Public License](#) as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

PyEPR is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with PyEPR. If not, see <<http://www.gnu.org/licenses/>>.