rasterpy Documentation

Release 04.04.2018

Ismail Baris

Apr 04, 2018

Contents

1 Introduction **1**
1.1 Indices and tables . . . 1

2 Installation **3**
2.1 Using pip . . . 3
2.2 Standard Python . . . 3
2.3 Test installation success . . . 3
2.4 Indices and tables . . . 3

3 Examples **5**
3.1 Raster Calculation . . . 5
3.2 Raster Conversion . . . 6
3.3 Raster Stack . . . 8
3.4 Indices and tables . . . 9

4 Technical documentation **11**
4.1 Raster . . . 11
4.2 Indices and tables . . . 14

5 Indices and tables **15**

Python Module Index **17**

CHAPTER 1

Introduction

Rasterpy contains basic routines to read and write raster files with python. With Rasterpy one can read the following raster formats:

- **'tif'**: All *.tiff* formats like GEOtiff etc.
- **'.hdr'**: All ENVI formats like *.bin* and *.hdr* formats. This is usefull if you want to read data from third-party software like Polsarpro.

You can find some examples here: *Examples*

1.1 Indices and tables

- *Examples*
- genindex
- modindex
- search

CHAPTER 2

Installation

There are currently different methods to install *rasterpy*.

2.1 Using pip

The ‘rasterpy ‘ package is provided on pip. You can install it with:

```
pip install rasterpy
```

2.2 Standard Python

You can also download the source code package from this repository or from pip. Unpack the file you obtained into some directory (it can be a temporary directory) and then run:

```
python setup.py install
```

2.3 Test installation success

Independent how you installed ‘ prism ‘, you should test that it was sucessfull by the following tests:

```
python -c "from rasterpy import Raster"
```

If you don’t get an error message, the module import was sucessfull.

2.4 Indices and tables

- *Examples*

- genindex
- modindex
- search

CHAPTER 3

Examples

Contents:

3.1 Raster Calculation

Here is an example of some basic features that rasterpy provides. Three bands are read from an image and averaged to produce something like a panchromatic band. This new band is then written to a new single band TIFF.

At first import rasterpy: ..

```
import rasterpy as rpy
import numpy as np
```

After that we define a path where our test tif file is located: ..

```
path = "C:\Users\ibari\Dropbox\GitHub\\rasterpy\\tests\data"
```

Then we open the grid and read it to an multidimensional array: ..

```
grid = rpy.Raster('RGB.byte.tif', path)
grid.to_array()
```

To access the array one can use *grid.array*.By default the loaded grid is flatten. The reason is as following: With a flatten 2 dimensional array the calculations based on the array are much easier: ..

```
>>> print(grid.array)
[[0. 0. 0. ... 0. 0. 0.]     # Band 1
 [0. 0. 0. ... 0. 0. 0.]     # Band 2
 [0. 0. 0. ... 0. 0. 0.]]    # Band 3
```

We can reshape the array to their original shapes with: ..

```
>>> grid.reshape()
>>> print(grid.array)
[[[0. 0. 0. ... 0. 0. 0.]    # Band 1
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]
 [[0. 0. 0. ... 0. 0. 0.]    # Band 2
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]
 [[0. 0. 0. ... 0. 0. 0.]    # Band 3
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]]
```

And we can also flatten it again with: ..

```
>>> grid.flatten()
>>> print(grid.array)
[[0. 0. 0. ... 0. 0. 0.]     # Band 1
 [0. 0. 0. ... 0. 0. 0.]     # Band 2
 [0. 0. 0. ... 0. 0. 0.]]    # Band 3
```

Now average each pixels of the RGB bands: ..

```
pan = np.mean(grid.array, axis=0)
```

After that we can write the reshaped array as a tiff: ..

```
grid.write(data=pan.reshape(grid.rows, grid.cols), filename='RGB_total.tif',␣
↪path=path)
```

Here is the result

3.2 Raster Conversion

Three bands are read from an image. Then we do some raster conversion on it. This new bands are then written to a new multi band TIFF.

At first import rasterpy. ..

```
import rasterpy as rpy
import numpy as np
```

After that we define a path where our test tif file is located. ..

```
path = "C:\Users\ibari\Dropbox\GitHub\\rasterpy\\tests\data"
```

Then we open the grid and read it to an multidimensional array: ..

```
grid = rpy.Raster('RGB.BRF.tif', path)
```

The quantification factor is a factor which scales reflectance value between 0 and 1. For sentinel 2 the factor is 10000 ..

```
grid.to_array(flatten=False, quantification_factor=10000)
```

By default the loaded grid is flatten. The reason is as following: With a flatten 2 dimensional array the calculations based on the array are much easier. But in our case this is not necessary: ..

```
>>> print(grid.array)
[[[0.1049 0.0979 0.1047 ... 0.0469 0.0551 0.0553]
  [0.1008 0.0974 0.0984 ... 0.0449 0.0497 0.0574]
  [0.1039 0.0957 0.1002 ... 0.0477 0.0546 0.0653]
  ...
  [0.1226 0.1151 0.1363 ... 0.0641 0.0599 0.0643]
  [0.1136 0.1299 0.144  ... 0.0654 0.0625 0.0651]
  [0.1069 0.1034 0.1356 ... 0.0616 0.0546 0.0629]]
 [[0.1025 0.0968 0.1027 ... 0.0435 0.0567 0.0585]
  [0.1008 0.0943 0.0996 ... 0.0466 0.0531 0.0625]
  [0.0969 0.0929 0.0987 ... 0.0523 0.0604 0.0692]
  ...
  [0.1223 0.112  0.1367 ... 0.0683 0.0587 0.0621]
  [0.112  0.1282 0.1409 ... 0.0658 0.0591 0.0635]
  [0.106  0.1018 0.1394 ... 0.0616 0.0551 0.0588]]
 [[0.1146 0.1088 0.114  ... 0.0484 0.0644 0.0703]
  [0.1112 0.1076 0.1148 ... 0.048  0.0599 0.0773]
  [0.1125 0.1064 0.1119 ... 0.0564 0.0747 0.0886]
  ...
  [0.1381 0.1259 0.1473 ... 0.0873 0.0746 0.0746]
  [0.1268 0.1404 0.1624 ... 0.0782 0.0702 0.0758]
  [0.1207 0.1194 0.1543 ... 0.0752 0.0617 0.0678]]]
```

Now we convert the whole file from a Bidirectional Reflectance Factor (BRF) into a Bidirectional Reflectance Distribution Function (BRDF): ..

```
grid.convert(system='BRF', to='BRDF', output_unit='dB')
```

After that we can write the converted data as a multi band tiff: ..

```
grid.write(data=grid.array, filename='RGB.BRDF.tif', path=path)
```

We can also convert the arrays from a BRF or BRDF into Backscatter coefficients (BSC). For this we need the inclination (iza) and viewing (vza) angles. These information is in the first two bands of the dataset. Thus, we specify it with parameter *band=(1, 2)*: ..

```
angle = rpy.Raster('RGB.BRF.Angle.tif')
angle.to_array(band=(1, 2), flatten=False)
```

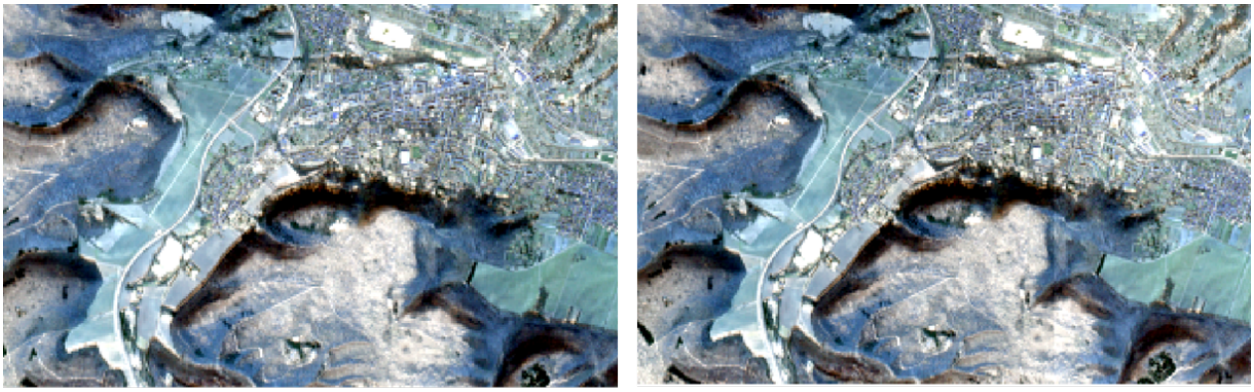
Note, that the arrays were converted from a BRF into a BRDF (dB) in the previous step: ..

```
grid.convert(system='BRDF', to='BSC', system_unit='dB', output_unit='dB', iza=angle.
↪array[0], vza=angle.array[1])
```

After the conversion we can wrtie the array to a raster file: ..

```
grid.write(data=grid.array, filename='RGB.BSC.tif', path=path)
```

The result is



BRDF BSC

3.3 Raster Stack

Here is an example of some basic stack features that rasterpy provides. Three bands of angle information are read from an image. The first band is the inclination(iza), the second band the viewing (vza) and the last band is the relative azimuth (raa) angle. Then we stack these bands together in an new array.

At first import rasterpy. ..

```
import rasterpy as rpy
import numpy as np
```

After that we define a path where our test tif file is located. ..

```
path = "C:\Users\ibari\Dropbox\GitHub\\rasterpy\\tests\data"
```

Then we open the grid and read it to an multidimensional array ..

```
angle = rpy.Raster('RGB.BRF.Angle.tif')
angle.to_array()
```

Now we can store the angle information in a 2D array ..

```
angle.dstack(unfold=True)
```

With *angle.stack* we can access to the stacked arrays ..

```
print(angle.stack)
    # IZA        # VZA        # RAA
[[  8.7763834   59.99580002 113.65518188]
 [  8.7763834   59.99580002 113.65518188]
 [  8.7763834   59.99580002 113.65518188]
 ...
 [  9.26036167  59.9394989  114.21447754]
```

```
 [  9.26036167  59.9394989  114.21447754]
 [  9.26036167  59.9394989  114.21447754]]
```

The advantage of this stack is we can access to the angle information for each pixel wit *angle.stack[pixel]*: .. code:

```
print(angle.stack[0])
[  8.7763834   59.99580002 113.65518188]
```

As one can see the sensing geometry of the first pixel is .. code:

```
print("The inclination zenith angle is: {0} [DEG]".format(angle.stack[0][0]))
print("The viewing zenith angle is: {0} [DEG]".format(angle.stack[0][1]))
print("The relative azimuth angle is: {0} [DEG]".format(angle.stack[0][2]))

The inclination zenith angle is: 8.77638339996 [DEG]
The viewing zenith angle is: 59.9958000183 [DEG]
The relative azimuth angle is: 113.655181885 [DEG]
```

3.4 Indices and tables

- genindex
- modindex
- search

CHAPTER 4

Technical documentation

4.1 Raster

class `rasterpy.`**`Raster`**(*filename=None*, *path=None*, *extension=None*, *check_dim=True*)

Import a binary file of ENVI or PolSARpro or a tif to a raster object.

Parameters

- **filename** (*str or tuple, optional*) – Filename as a string or a tuple with filenames of the raster data.
- **path** (*str, optional*) – Path to raster data. If path is None the path will set to current directory with os.getwd().
- **extension** (*str, optional*) – If you want to import all files in a directory with a specific extension, set filename to None and define an extension like '.tiff' or '.bin'.
- **check_dim** (*bool*) – If True the imported files must have the same dimensions.

raster

osgeo.gdal.Dataset – Contains the gdal data set for the raster files.

cols, rows

int or tuple – Columns and row size of the imported raster files.

band

int or tuple – Number of bands in each raster file.

dim

list or tuple – Information about the dimension. It contains [rows, cols, bands].

dtype

str or tuple – Gdal data types.

projection

str or tuple – Information about the projection of each raster file.

xmin, ymin
int or tuple – Origen of x and y pixel.

xres, yres
int or tuple – Resulution information in x and y axis.

nodata
No data values.

info
dict – All information in a dictionary.

static BRDF(*BSC*, *iza*, *vza*, *angle_unit='RAD'*)
Convert a Radar Backscatter Coefficient (BSC) into a BRDF.

Parameters

- **BSC** (*int, float or array_like*) – Radar Backscatter Coefficient (sigma 0).
- **iza** (*int, float or array_like*) – Sun or incidence zenith angle.
- **vza** (*int, float or array_like*) – View or scattering zenith angle.
- **angle_unit** (*{'DEG', 'RAD'} (default = 'RAD'), optional*) –
  - 'DEG': All input angles (iza, vza, raa) are in [DEG].
  - 'RAD': All input angles (iza, vza, raa) are in [RAD].

Returns **BRDF value**

Return type int, float or array_like

static BRF(*BRDF*)
Convert a BRDF into a BRF.

Parameters **BRDF** (*int, float or array_like*) – BRDF value.

Returns **BRF value**

Return type int, float or array_like

static BSC(*BRDF*, *iza*, *vza*, *angle_unit='RAD'*)
Convert a BRDF in to a Radar Backscatter Coefficient (BSC).

Parameters

- **BSC** (*int, float or array_like*) – Radar Backscatter Coefficient (sigma 0).
- **iza** (*int, float or array_like*) – Sun or incidence zenith angle.
- **vza** (*int, float or array_like*) – View or scattering zenith angle.
- **angle_unit** (*{'DEG', 'RAD'} (default = 'RAD'), optional*) –
  - 'DEG': All input angles (iza, vza, raa) are in [DEG].
  - 'RAD': All input angles (iza, vza, raa) are in [RAD].

Returns **BRDF value**

Return type int, float or array_like

convert(*system='BSC'*, *to='BRDF'*, *system_unit='linear'*, *output_unit='linear'*, *iza=None*, *vza=None*, *angle_unit='RAD'*)
Convert the data from BSC, BRDF, BRF to BRDF, BSC or BRF.

Parameters

- **system** (*{'BSC', 'BRDF', 'BRF'}*) – The actual unit of the data. Default is 'BSC'.
- **to** (*{'BSC', 'BRDF', 'BRF'}*) – The desired unit after conversion. Default is 'BRDF'
- **system_unit** (*{'linear', 'dB'}, optional*) – Are the measurements in a linear scale or in decibel? Default is 'linear'.
- **output_unit** (*{'linear', 'dB'}, optional*) – The desired output format. Default is 'linear'.
- **iza** (*int, float, array_like or None, , optional*) – Sun or incidence zenith angle. Default is None.
- **vza** (*int, float, array_like or None, optional*) – View or scattering zenith angle.
- **angle_unit** (*{'DEG', 'RAD'}, optional*) –
  - 'DEG': All input angles (iza, vza, raa) are in [DEG] (default).
  - 'RAD': All input angles (iza, vza, raa) are in [RAD].

Returns

Return type None

copy()

Copy the imported array.

Returns **copy** – A copy of Raster.array attribute.

Return type array_like or tuple

static dB(*x*)

Convert a linear value to dB.

dstack(*unfold=False*)

Stack 1-D arrays as columns into a 2-D array. Take a sequence of 1-D arrays and stack them as columns to make a single 2-D array. 2-D arrays are stacked as-is, just like with numpy.hstack. 1-D arrays are turned into 2-D columns first.

Parameters **unfold** (*bool, optional*) – If the arrays are multi dimensional this option extracts the individual dimension and stack it to an array. Default is False.

stack

array_like – Stacked array.

flatten()

Collapse the loaded arrays into one dimension.

Returns

Return type None

static linear(*x*)

Convert a dB value in linear.

reset()

Delete the attributes Raster.array and Raster.stack

reshape()

Reshape loaded arrays to their original dimension.

set_nodata (*nodata*)
Set and assign a new no data value.

Parameters **nodata** (*int, float or np.nan*) – New no data value. A tuple with in, float or np.nan is also possible if you imported more than one file.

Returns

Return type None

to_array (*band=None*, *flatten=True*, *quantification_factor=1*)
Converts a binary file of ENVI or PolSARpro or a tif to a numpy array.

Parameters

- **band** (*int, tuple or None, optional:*) – Define bands which you want to import. If None (default) import all bands in a multidimensional array. You can also specify bands in a tuple. E.g. band=(1, 3) will load the first and third band of the image.
- **flatten** (*bool*) – if flatten is True the output array is one dimensional. You can convert it to an 2 dimensional array with Raster.reshape
- **quantification_factor** (*int, optional*) – A quantification factor that scales the reflectance values from 0 to 1. It is only required if the imported raster files are reflectance values. For sentinel 2 the factor is 10000. Default is 1, which have no effect.

array
array_like or tuple with array_likes – Raster files as arrays.

write (*data*, *filename*, *path=None*, *reference=0*)
Convert an array into a binary (.bin) file with header (.hdr) or a Tif file.

Parameters

- **data** (*array_like or tuple*) – Arrays you want to export.
- **filename** (*str or tuple*) – File names of the exported arrays. Supported file extension are '.tif' or '.bin'
- **path** (*str, optional*) – Export path. If path is None the path will set to current directory with os.getwd().
- **reference** – If the Raster import contains several grids, you can specify which of these grids you want to use as reference for geo-spatial information (default=0).

Returns

Return type Grid as .tif or bin

4.2 Indices and tables

- *Examples*
- genindex
- modindex
- search

CHAPTER 5

Indices and tables

- *Examples*
- genindex
- modindex
- search

Python Module Index

r

rasterpy, 11

Index

A

array (rasterpy.Raster attribute), 14

B

band (rasterpy.Raster attribute), 11
BRDF() (rasterpy.Raster static method), 12
BRF() (rasterpy.Raster static method), 12
BSC() (rasterpy.Raster static method), 12

C

convert() (rasterpy.Raster method), 12
copy() (rasterpy.Raster method), 13

D

dB() (rasterpy.Raster static method), 13
dim (rasterpy.Raster attribute), 11
dstack() (rasterpy.Raster method), 13
dtype (rasterpy.Raster attribute), 11

F

flatten() (rasterpy.Raster method), 13

I

info (rasterpy.Raster attribute), 12

L

linear() (rasterpy.Raster static method), 13

N

nodata (rasterpy.Raster attribute), 12

P

projection (rasterpy.Raster attribute), 11

R

Raster (class in rasterpy), 11
raster (rasterpy.Raster attribute), 11
rasterpy (module), 11
reset() (rasterpy.Raster method), 13
reshape() (rasterpy.Raster method), 13

S

set_nodata() (rasterpy.Raster method), 13
stack (rasterpy.Raster attribute), 13

T

to_array() (rasterpy.Raster method), 14

W

write() (rasterpy.Raster method), 14