tropy-tutorials

Release 0.1

Mar 22, 2020

Data IO

CHAPTER 1

Intro

This is a set of tutorial notebooks which aim to partially describe the functionality of the python package *tropy* which I started to develop at TROPOS since 2011. It contains a rather diverse collection of utility functions for which I did not found an appropriate solution in an existing package. The python landscape has considerably changed - hence some of the tools are somehow outdated, some functions are now covered by established package collections.

All tutorials are available as jupyter notebooks.

CHAPTER 2

Tools for IO

Function have been written to ease input of certain obervational or model-simulated fields available at TROPOS. These function included data from

- **observations from Meteosat SEVIRI** *with the 'MSevi' class* and *make RGBs with the 'MSeviRGB' class*
- **observations from the DWD radolan Radar Composite** :doc: *with the Radolan class TBD*

CHAPTER 3

Tools for Plotting

Some helper functions to make typical task in plotting easier.

- **plotting shades with a non-linear (discrete) colormap** *with the 'shaded' module*
- **using some special colormaps** *with the 'colormaps' module*
- **adding meta data to PNG Files** *with the 'meta2png' module*

CHAPTER 4

Tools for Analysis

Some helper functions to make special tasks in data analysis easier.

- **rank transformations and histogram matching** *with the 'statistics' module*

4.1 How to read MSG SEVIRI data from the TROPOS archive?

This tutorial shows how to read MSG SEVIRI data at TROPOS using the `MSevi` data container class which is part of the python library collection `tropy`. The `MSevi` data container loads satellite data from hdf file if the hdf is available and the region is inside the predefined 'eu'-domain. If not, then the `MSevi` class can also load the original HRIT files, but in this case no HRV input is implemented.

4.1.1 Import Libraries

[1]:
```
%matplotlib inline

# standard libs
import numpy as np
import datetime

# plotting and mapping
import pylab as plt
import cartopy.crs as ccrs
import cartopy.feature as cfeature
import seaborn as sns
sns.set_context('talk')

# the own tropy lib
from tropy.l15_msevi.msevi import MSevi
```

SEVIRI data are loaded into the `MSevi` data container.

4.1.2 Configuration of Rapid Scan

For configuration, we have to set region, scan_type and time (as object).

[2]:
```
time = datetime.datetime( 2013, 6, 8, 12, 0)
region = 'eu'
scan_type = 'rss'
```

Initialize the Data Container.

[3]:
```
s = MSevi(time = time, region = region, scan_type = scan_type)
```

4.1.3 Load Infrared Channel

Start with the infrared channel at 10.8 um.

[4]:
```
s.load('IR_108')
```

```
Region suggests use of hdf file
```

```
/vols/fs1/store/senf/.conda/python37/lib/python3.7/site-packages/h5py/_hl/dataset.py:
↪313: H5pyDeprecationWarning: dataset.value has been deprecated. Use dataset[()]␣
↪instead.
  "Use dataset[()] instead.", H5pyDeprecationWarning)
```

Now, the sat data are available as radiance (i.e. already converted from the counts using slope and offset). A dictionary s.rad is used to store the radiance information.

[5]:
```
print (s.rad)
```

```
{'IR_108': array([[ 53.30927581,  51.46395473,  48.59345526, ...,  77.29844993,
         78.32362831,  80.9890921 ],
       [ 55.56466825,  53.10424014,  47.15820553, ...,  79.14377102,
         80.78405643,  81.19412778],
       [ 52.89920446,  49.20856229,  41.41720659, ...,  72.58262938,
         76.6833429 ,  79.75887804],
       ...,
       [134.70843927, 136.55376035, 137.16886738, ...,  99.23726728,
         99.44230296,  99.44230296],
       [135.11851062, 135.11851062, 136.34872468, ...,  99.44230296,
         99.23726728,  99.44230296],
       [135.3235463 , 135.11851062, 135.73361765, ...,  99.44230296,
         99.23726728,  99.44230296]])}
```

What you want for infrared channel is perhaps the brightness temperature (BT). To get it, the is the method `rad2bt` that creates a BT dictionary under `s.bt`.

[6]:
```
s.rad2bt('IR_108')
```

Plotting

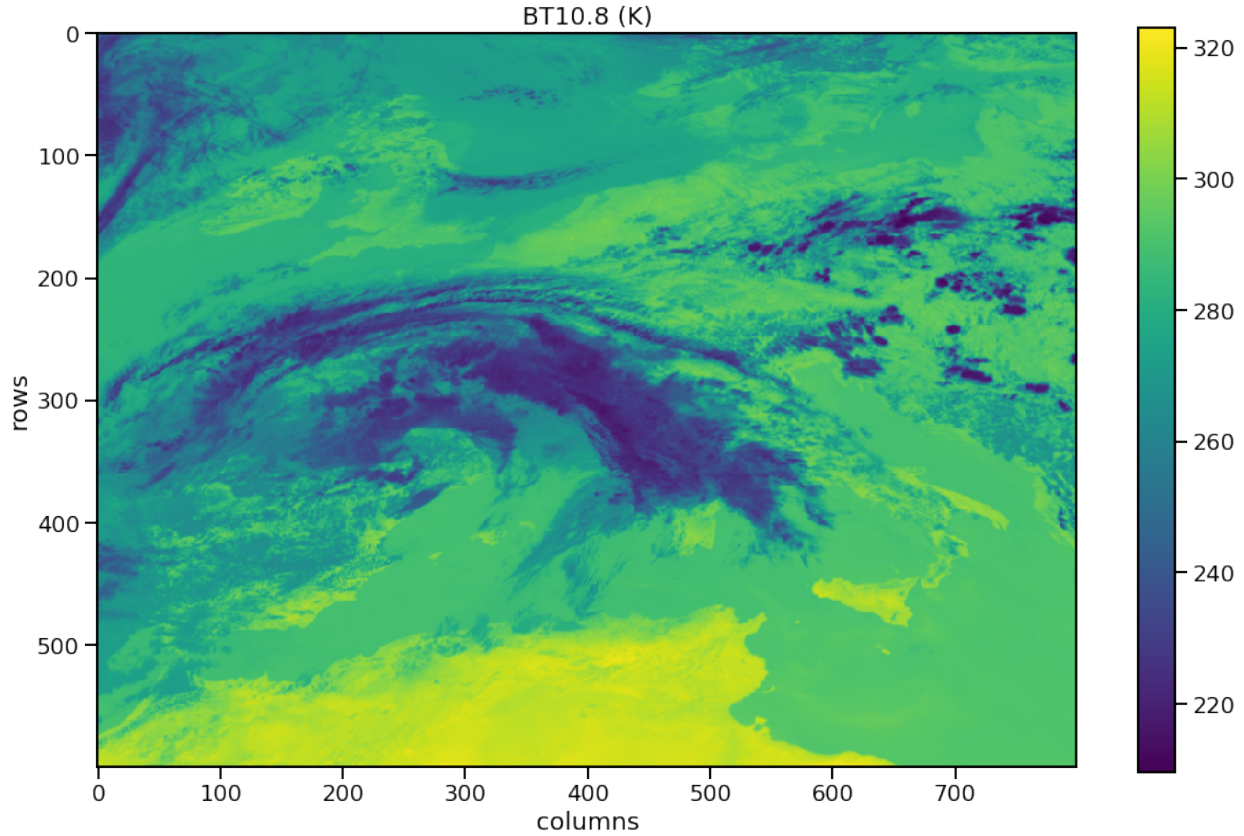
[7]:
```
plt.figure( figsize = (16,10))
plt.imshow(s.bt['IR_108'])
plt.xlabel('columns')
plt.ylabel('rows')
```

(continues on next page)

(continued from previous page)

```
plt.title('BT10.8 (K)')
plt.colorbar()
```

[7]:
```
<matplotlib.colorbar.Colorbar at 0x7f0f5f17b550>
```



4.1.4 Read the other Scan Type: Operational Scan

The SEVIRI operational scan is called `pzs` in our case. No hdf files are available, therefore the native HRIT format needs to be read.

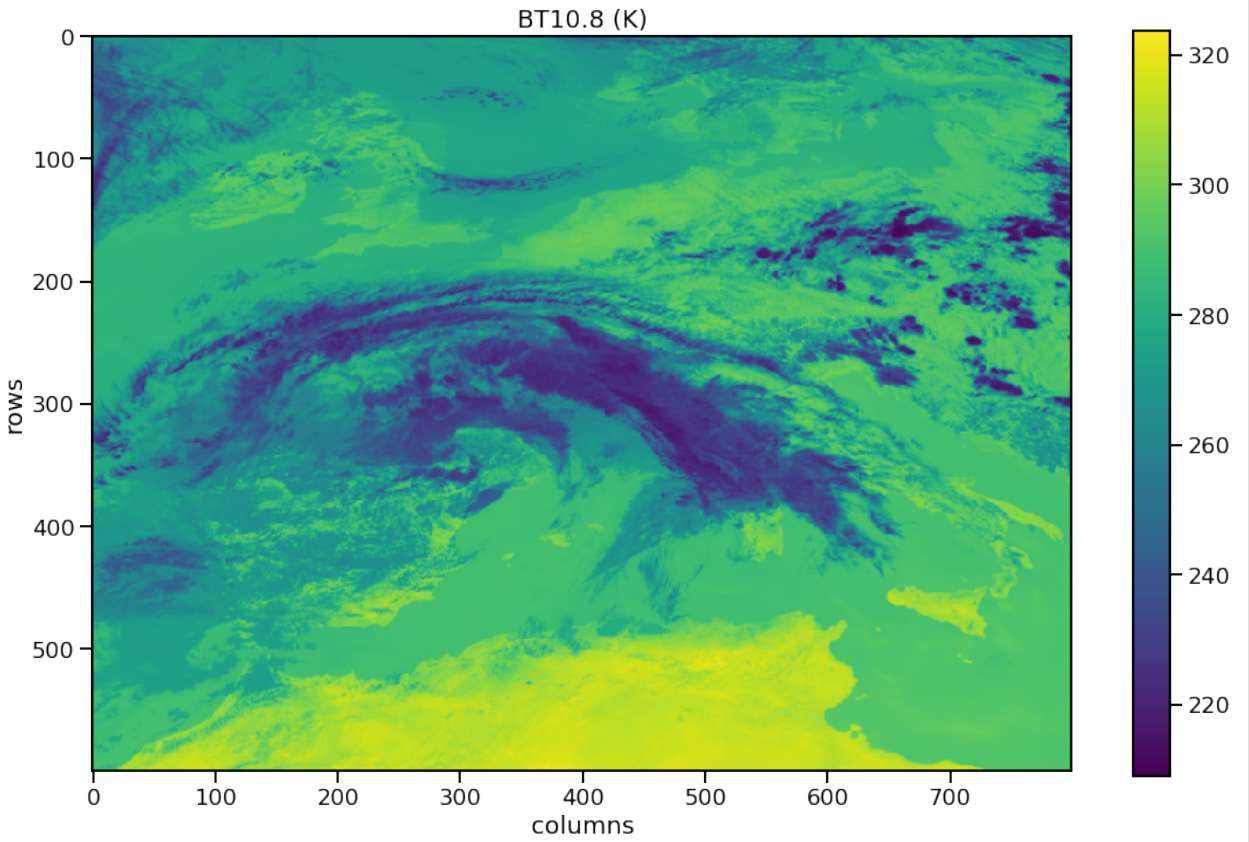
[8]:
```
scan_type = 'pzs'
s = MSevi(time = time, region = 'eu', scan_type = scan_type)
s.load('IR_108')
s.rad2bt()
```

```
Region suggests use of hdf file
ERROR:  /vols/fs1/satellit/datasets/eumcst/msevi_pzs/l15_hdf/eu/2013/06/08/msg?-sevi-
↪20130608t1200z-l15hdf-pzs-eu.c2.h5  does not exist!
... reading  /tmp/hrit4900095955/H-000-MSG3__-MSG3________-IR_108___-000007___-
↪201306081200-__
... reading  /tmp/hrit4900095955/H-000-MSG3__-MSG3________-IR_108___-000008___-
↪201306081200-__

Combine segments

Do calibration
```

Plotting

[9]:
```
plt.figure( figsize = (16,10))
plt.imshow(s.bt['IR_108'])
plt.xlabel('columns')
plt.ylabel('rows')
plt.title('BT10.8 (K)')
plt.colorbar()
```

[9]:
```
<matplotlib.colorbar.Colorbar at 0x7f0f5f086128>
```



It is slightly shifted! Do you see that?

4.1.5 Back to the Rapid Scan: Adjusting Region Configuration

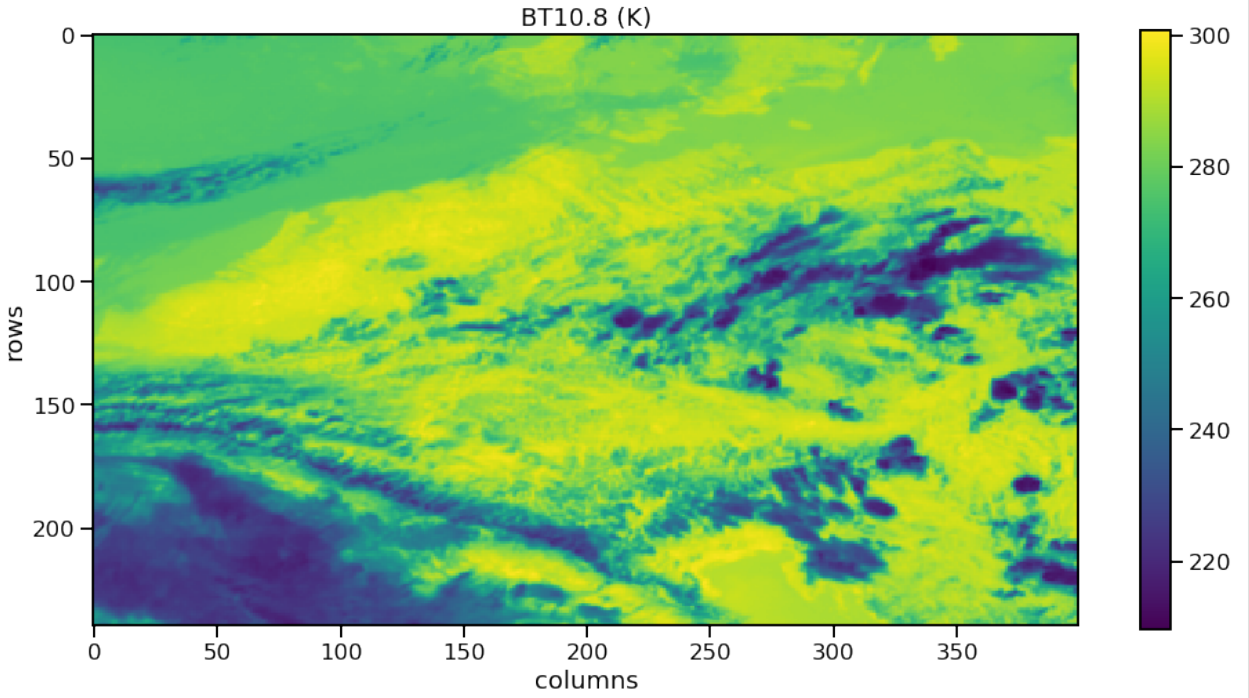
[10]:
```
region = ((216, 456), (1676, 2076))
scan_type = 'rss'

s = MSevi(time = time, region = region, scan_type = scan_type)
```

[11]:
```
s.load('IR_108')
s.rad2bt('IR_108')
```

```
Region suggests use of hdf file
```

[12]:
```
plt.figure( figsize = (16,8))
plt.imshow(s.bt['IR_108'])
plt.xlabel('columns')
plt.ylabel('rows')
plt.title('BT10.8 (K)')
plt.colorbar()
```

[12]:
```
<matplotlib.colorbar.Colorbar at 0x7f0f5efbf630>
```



4.1.6 Getting geo-reference

The `MSevi` class also provides the method to read geo-reference, i.e. longitude and latitude (in deg), for the selected region cutout. The method is called `s.lonlat()` and the result is stored in the attributes `s.lon` and `s.lat`.

[13]:
```
s.lonlat()

print( s.lon, s.lat )
```

```
[[-0.3794155  -0.3231449  -0.26698685 ... 21.472942   21.529903
  21.58696   ]
 [-0.3580389  -0.30195236 -0.24592876 ... 21.446722   21.50356
  21.560478  ]
 [-0.33676338 -0.28081322 -0.22490978 ... 21.42049    21.47727
  21.53405   ]
 ...
 [ 2.335071    2.3752937   2.415495   ... 18.15725    18.197723
  18.238157  ]
 [ 2.3413725   2.3815722   2.421739   ... 18.149572   18.190002
  18.230433  ]
 [ 2.347673    2.3878417   2.427947   ... 18.141941   18.182312
  18.22275   ]] [[57.7341   57.731903 57.729946 ... 57.810207 57.81226  57.814465]
 [57.662613 57.660572 57.658646 ... 57.7384   57.740463 57.742657]
```

(continues on next page)

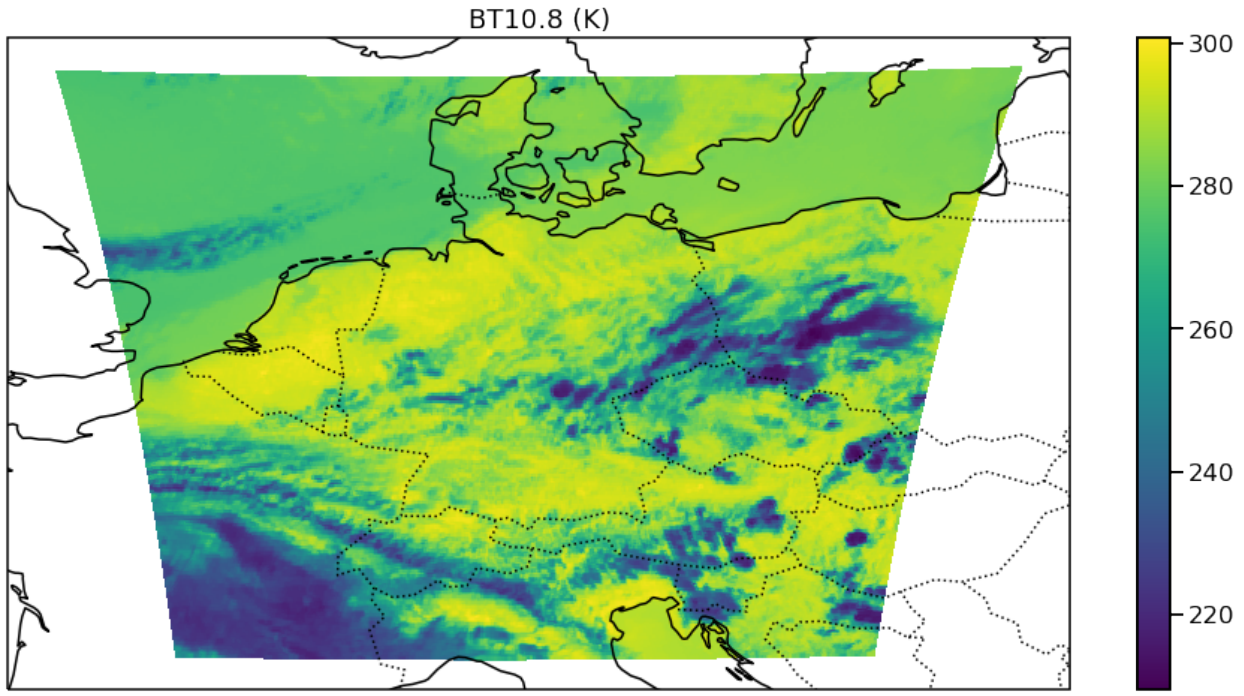
(continued from previous page)

```
 [57.591175 57.58916  57.58723  ... 57.66635  57.668564 57.67074 ]
 ...
 [44.519314 44.518463 44.51767  ... 44.55012  44.551014 44.551758]
 [44.473633 44.47273  44.47193  ... 44.504284 44.505165 44.506012]
 [44.427914 44.42699  44.426296 ... 44.45857  44.459362 44.460358]]
```

Now, we plot again, but geo-referenced.

[14]:
```
plt.figure( figsize = (16,8))

ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines( resolution='50m' )
ax.add_feature(cfeature.BORDERS, linestyle=':')

plt.pcolormesh(s.lon, s.lat, s.bt['IR_108'])
plt.title('BT10.8 (K)')
plt.colorbar()
```

[14]:
```
<matplotlib.colorbar.Colorbar at 0x7f0f5eedcd68>
```



Okay, this set of plotting methods might be needed. Let's make a function:

4.1.7 Loading several channels

Now, we make a channel list and read all channels. The easied way is the get the default channel list from the msevi config.

[15]:
```
from tropy.l15_msevi.msevi_config import _narrow_channels
print( _narrow_channels )
```

(continues on next page)

(continued from previous page)

```
full_channel_list = _narrow_channels + ['HRV',]
```

```
['VIS006', 'VIS008', 'IR_016', 'IR_039', 'WV_062', 'WV_073', 'IR_087', 'IR_097', 'IR_
↪108', 'IR_120', 'IR_134']
```

[16]:
```
s.load(full_channel_list)
print( s.rad.keys())
```

```
Region suggests use of hdf file
dict_keys(['IR_108', 'VIS006', 'VIS008', 'IR_016', 'IR_039', 'WV_062', 'WV_073', 'IR_
↪087', 'IR_097', 'IR_120', 'IR_134', 'HRV'])
```

Juchu! All channels are loaded!

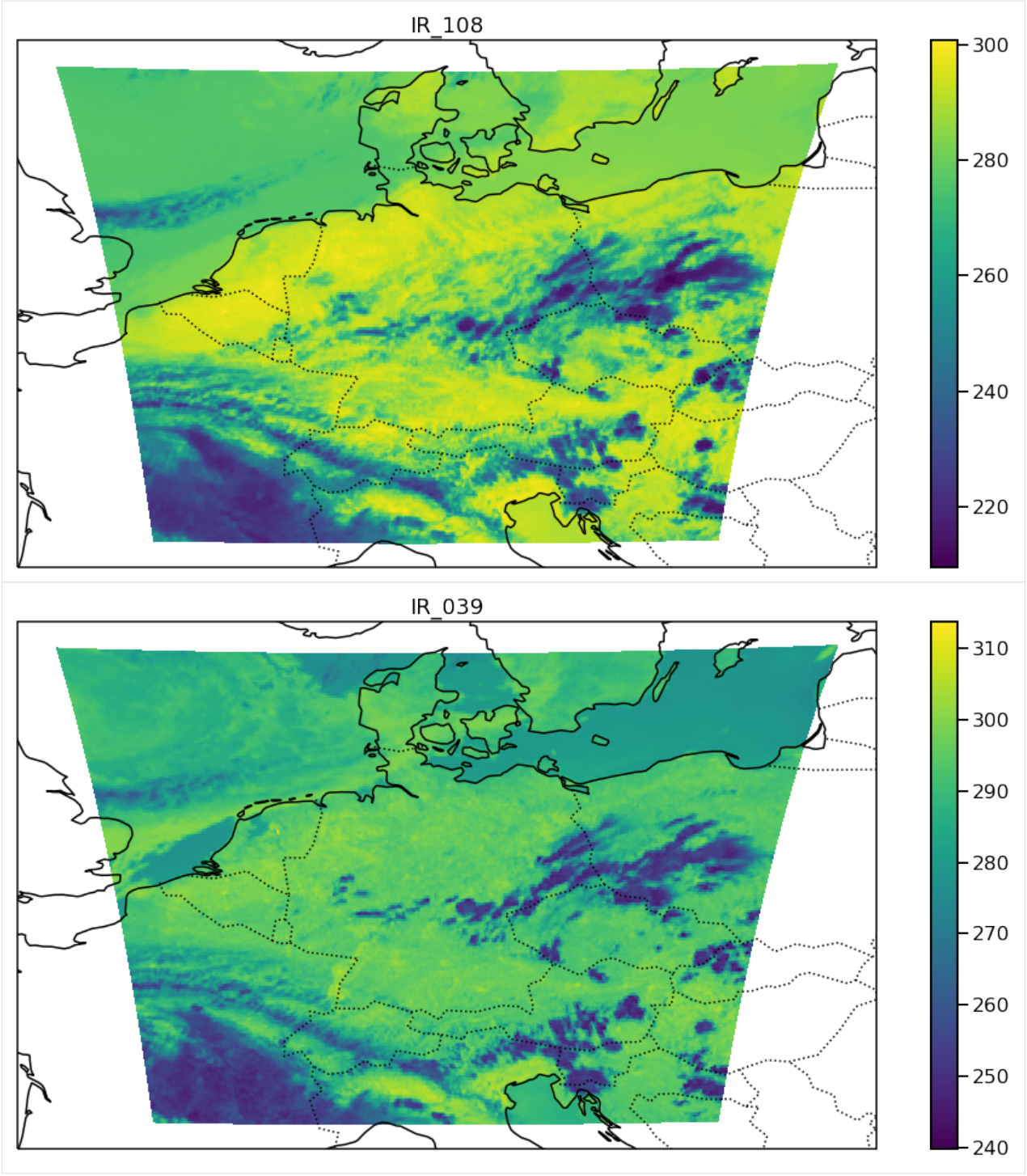
Now, we convert all IR channel radiances to BTs. No arg means all...

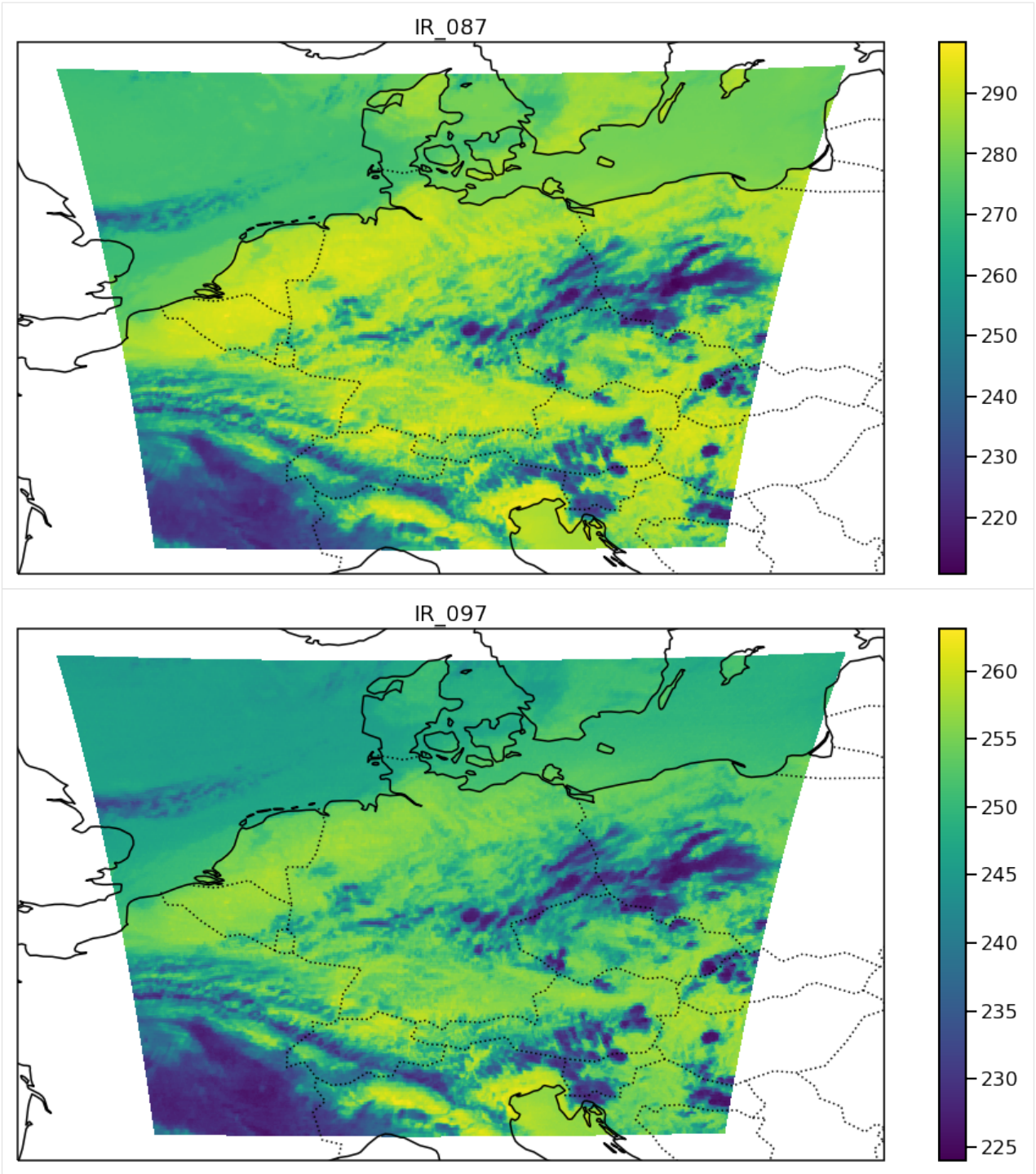
[17]:
```
s.rad2bt()
print (s.bt.keys())
```

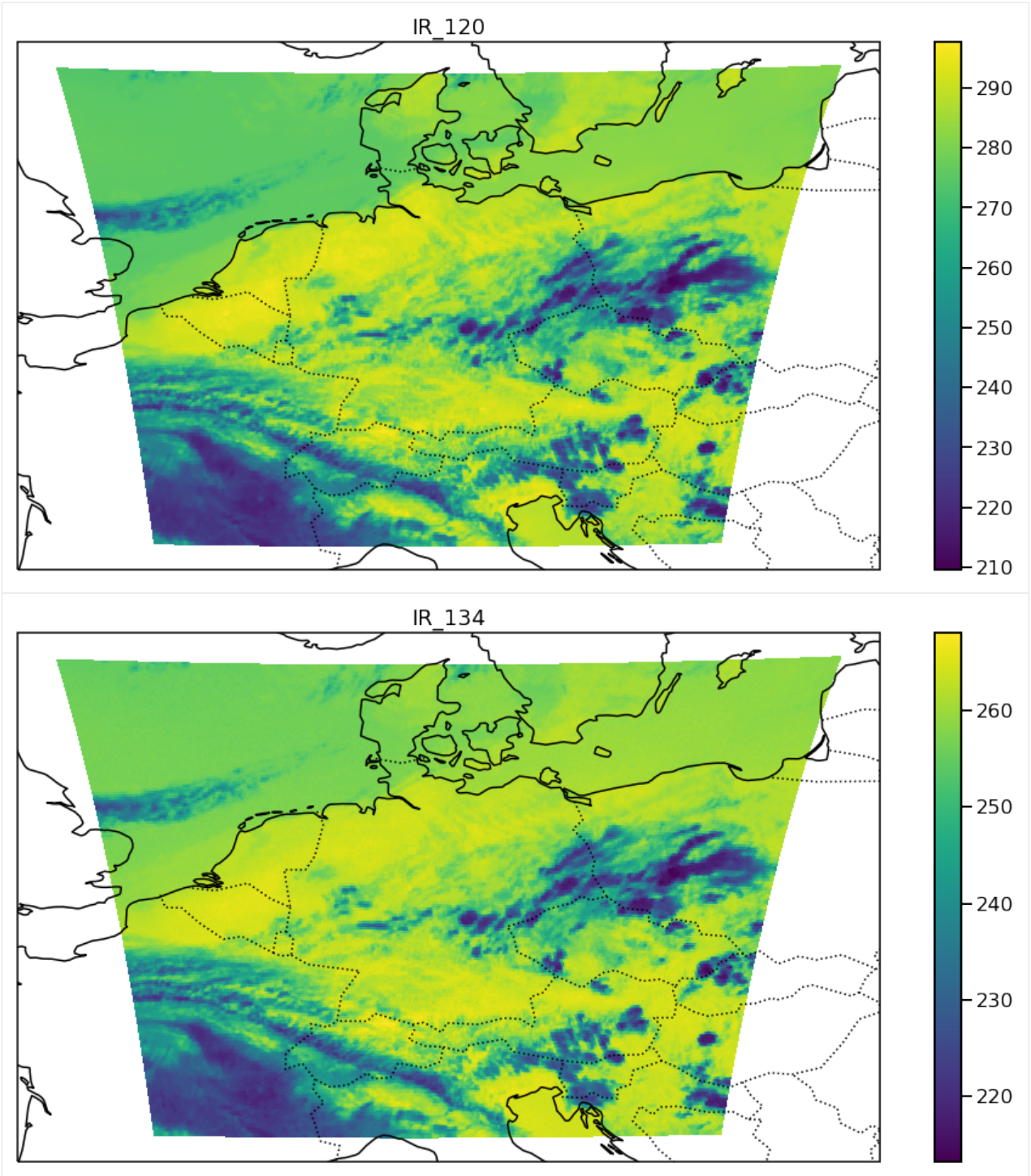
```
dict_keys(['IR_108', 'IR_039', 'IR_087', 'IR_097', 'IR_120', 'IR_134', 'WV_062', 'WV_
↪073'])
```

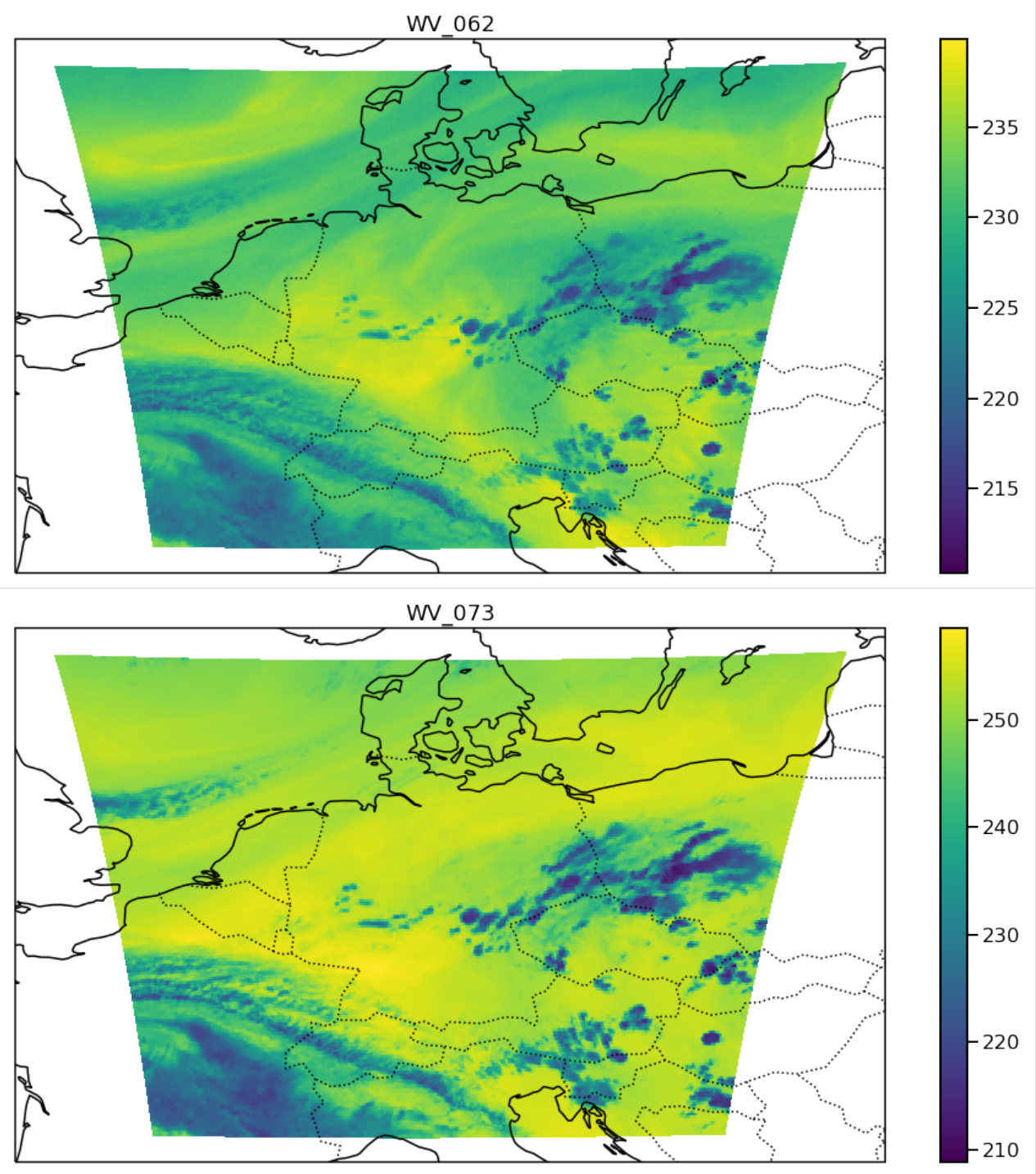
Time to explore the content. We loop over all IR channels...

[18]:
```
for chan_name in s.bt.keys():
    plt.figure( figsize = (16,8))

    ax = plt.axes(projection=ccrs.PlateCarree())
    ax.coastlines( resolution='50m' )
    ax.add_feature(cfeature.BORDERS, linestyle=':')

    plt.pcolormesh(s.lon, s.lat, s.bt[chan_name])
    plt.colorbar()
    plt.title(chan_name)
```


IR_108
300
280
260
240
220
IR_039
310
300
290
280
270
260
250
240


IR_087
290
280
270
260
250
240
230
220
IR_097
260
255
250
245
240
235
230
225


IR_120
290
280
270
260
250
240
230
220
210
IR_134
260
250
240
230
220



Wow, what a welth of information ;-)

But, visible is still missing, right? For this, there is the second conversion method called s.rad2refl that calculated reflectances. Note, no cos of solar zenith angle correction is applied here!

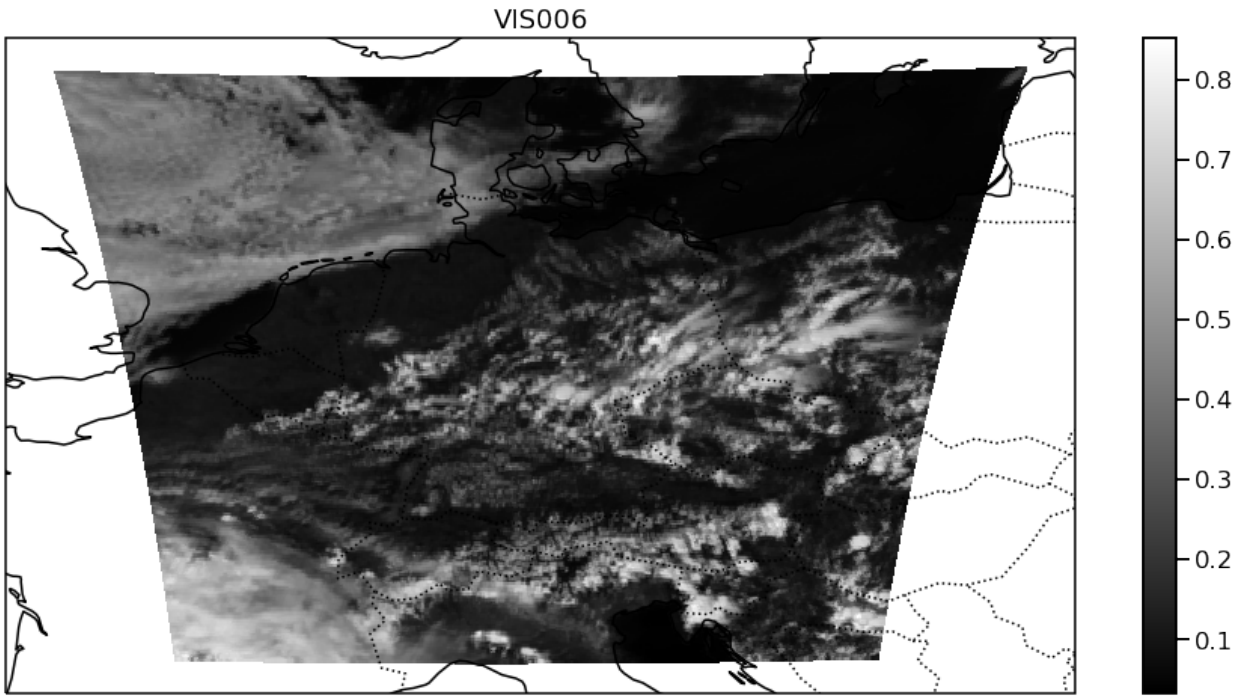
[19]:
```
s.rad2refl()
print (s.ref.keys())
```

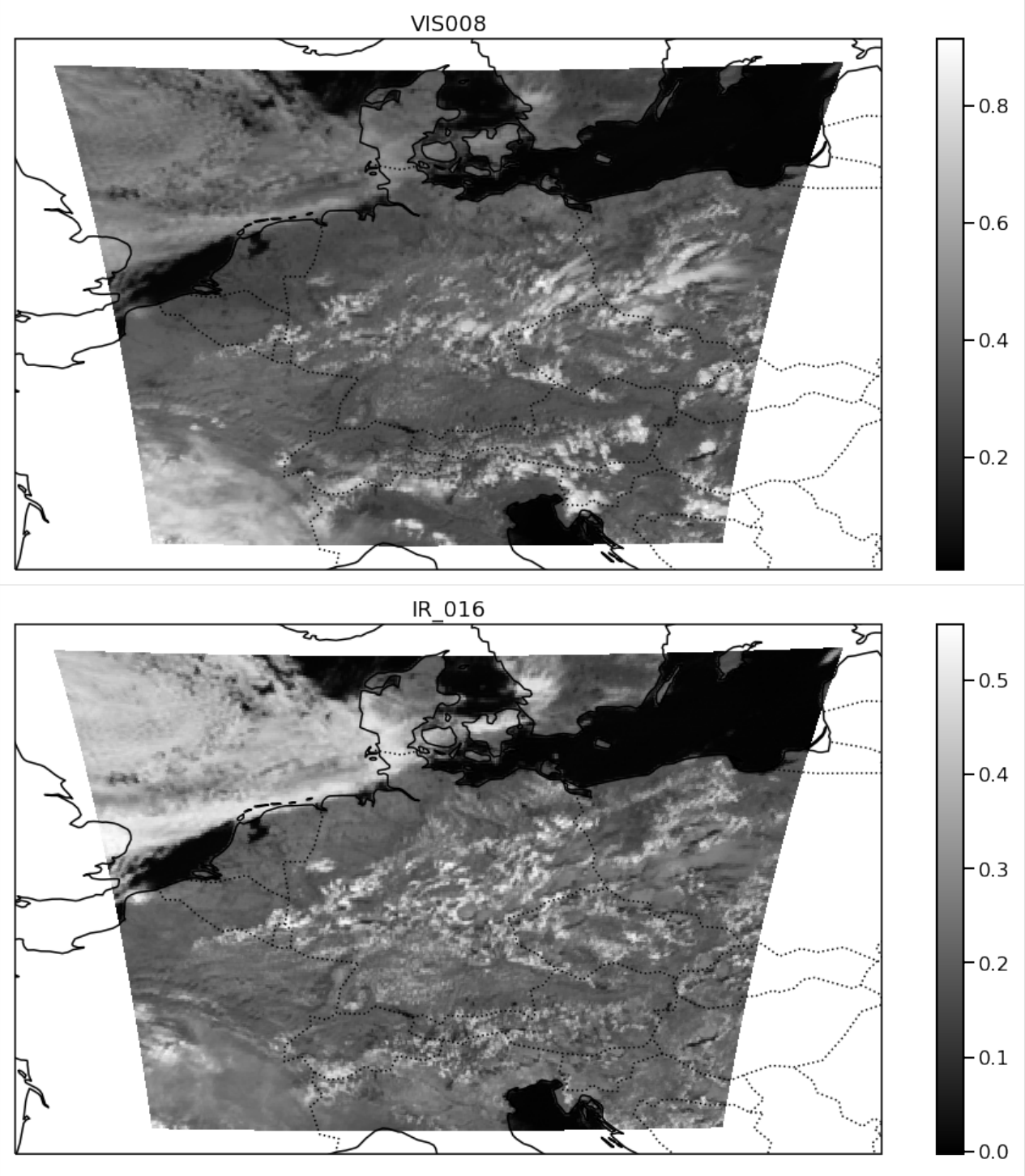
```
dict_keys(['HRV', 'IR_016', 'VIS006', 'VIS008'])
```

Now, we plot the visible stuff.

First, the narrow-band channels.

[20]:
```
for chan_name in ['VIS006', 'VIS008', 'IR_016']:
    plt.figure( figsize = (16,8))

    ax = plt.axes(projection=ccrs.PlateCarree())
    ax.coastlines( resolution='50m' )
    ax.add_feature(cfeature.BORDERS, linestyle=':')

    plt.pcolormesh(s.lon, s.lat, s.ref[chan_name], cmap = plt.cm.gray)
    plt.colorbar()
    plt.title(chan_name)
```





Sat-Images are really beautiful!

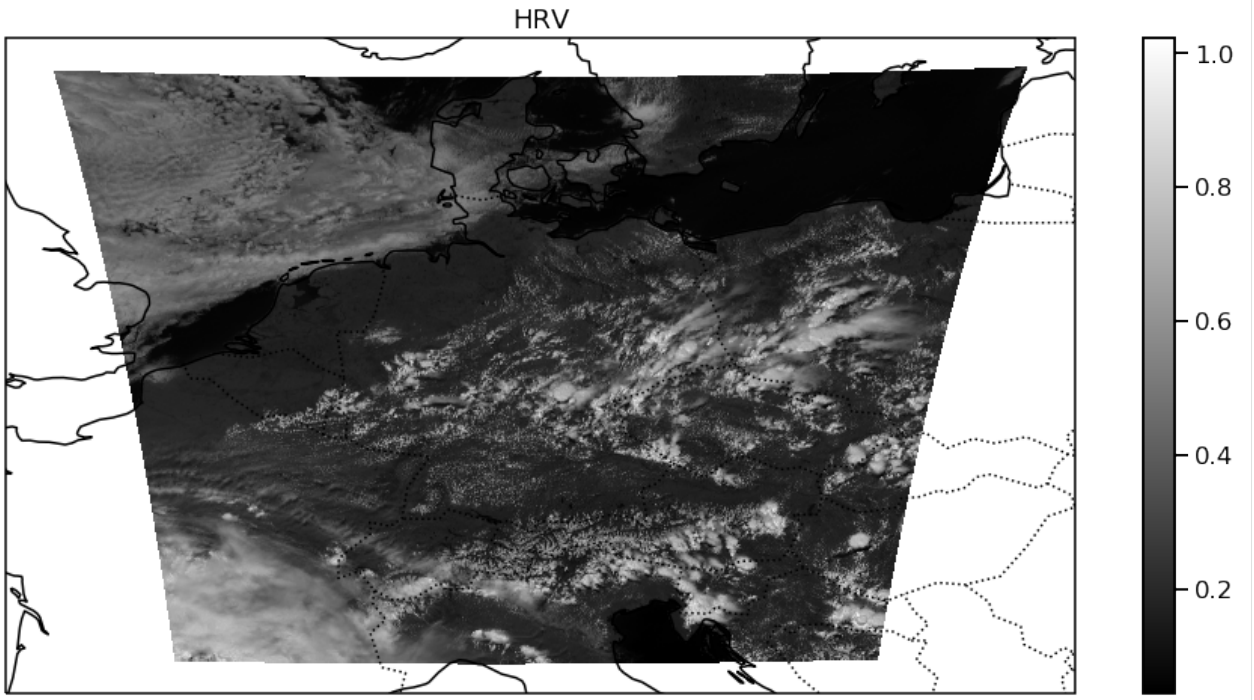
4.1.8 HRV channel

We already loaded did all the input for HRV. We * loaded the radiances of the HRV channel * converted them to reflectances * and loaded georeference

With the narrow-band geo-ref, also a high-res geo-ref was calculated (just by linear interpolation). The high-res. lon and lat values are stored in the attributes s.hlon and s.hlat and can be used to plot and analyze HRV data

[21]:
```
plt.figure( figsize = (16,8))

ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines( resolution='50m' )
ax.add_feature(cfeature.BORDERS, linestyle=':')

plt.pcolormesh(s.hlon, s.hlat, s.ref['HRV'], cmap = plt.cm.gray)
plt.colorbar()
plt.title('HRV')
```

[21]:
```
Text(0.5, 1.0, 'HRV')
```



Great detail in there!

4.2 How to make MSG SEVIRI RGB with the `tropy` interface?

This tutorial shows how to make RGBs with MSG SEVIRI data on our TROPOS servers. It is possible to read data from hdf or HRIT (only low-res).

4.2.1 Import Libraries

[1]:
```
%matplotlib inline

import pylab as plt
import numpy as np
import datetime
```

(continues on next page)

(continued from previous page)

```
from tropy.l15_msevi.msevi_rgb import MSeviRGB
```

[3]:
```
plt.rcParams['figure.figsize'] = (12.0, 8.0)
plt.rcParams['font.size'] = 18.0
plt.rcParams['lines.linewidth'] = 3
```

SEVIRI data are loaded into the `MSeviRGB` data container.

4.2.2 Configuration

For configuration, we have to set `region`, `scan_type` and `time` (as object).

[5]:
```
time = datetime.datetime( 2013, 6, 8, 12, 0)
region = 'eu'
scan_type = 'rss'
```

Initialize the Data Container.

[6]:
```
s = MSeviRGB(time = time, region = region, scan_type = scan_type)
```

```
Region suggests use of hdf file
```

```
/vols/fs1/store/senf/.conda/python27mod/lib/python2.7/site-packages/h5py/_hl/dataset.
↪py:313: H5pyDeprecationWarning: dataset.value has been deprecated. Use dataset[()]␣
↪instead.
  "Use dataset[()] instead.", H5pyDeprecationWarning)
```

4.2.3 Load RGBs

Start with the natural color RGB.

[7]:
```
s.create_rgb('pytroll_nc')
```

```
['IR_016', 'VIS008', 'VIS006'] is already loaded!
```

The images are stored in the s.images dictionary.

[8]:
```
print (s.images)
```

```
{'pytroll_nc': <PIL.Image.Image image mode=RGB size=800x600 at 0x7FF8F833A390>}
```

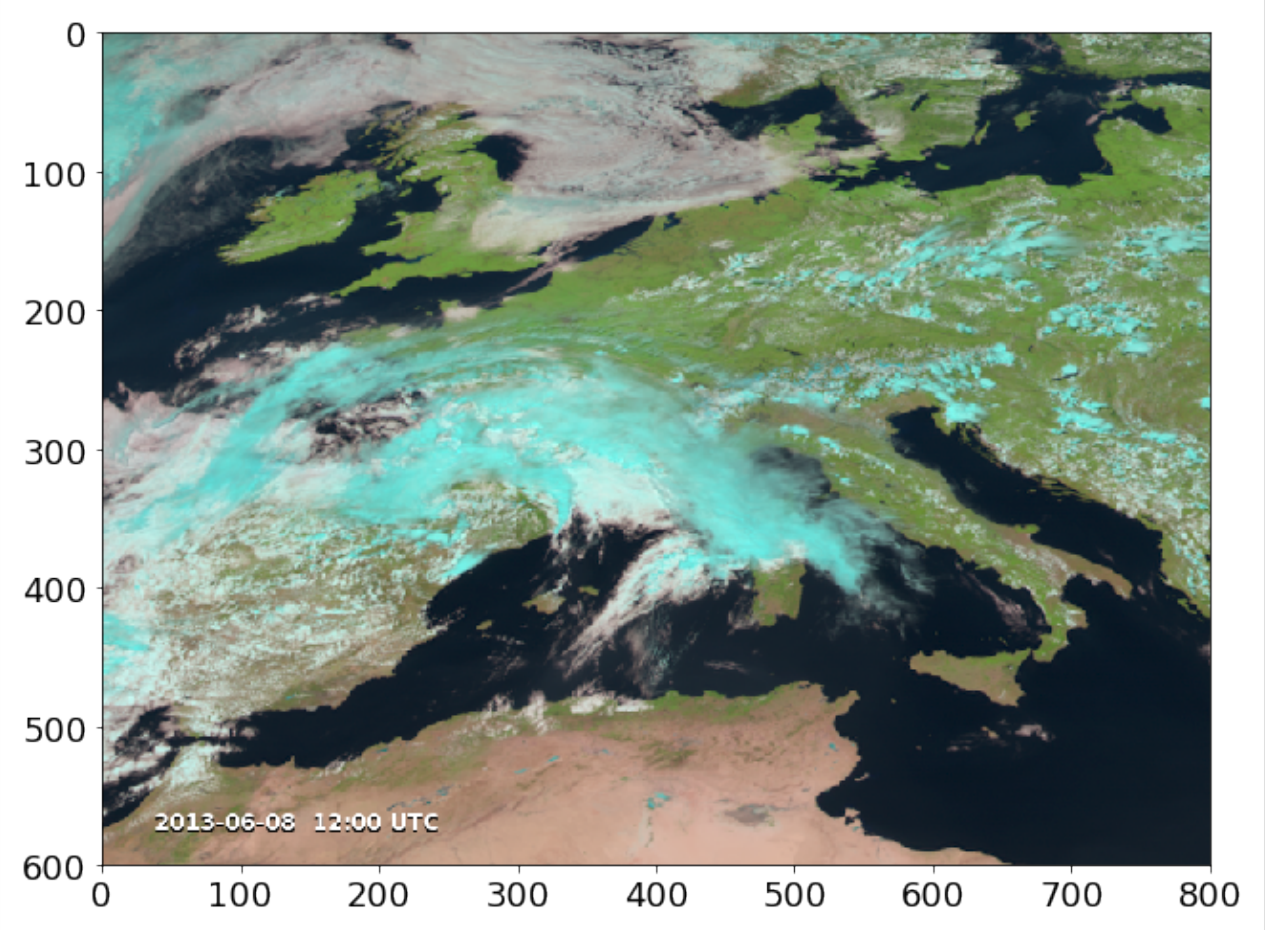
`s.show` is a method which opens an external viewer to see the image.

[9]:
```
#s.show('pytroll_nc')
```

This is how the image look like...

[11]:
```
rgb = np.array( s.images['pytroll_nc'] )
plt.imshow(rgb)
```

[11]:
```
<matplotlib.image.AxesImage at 0x7ff8db59b990>
```



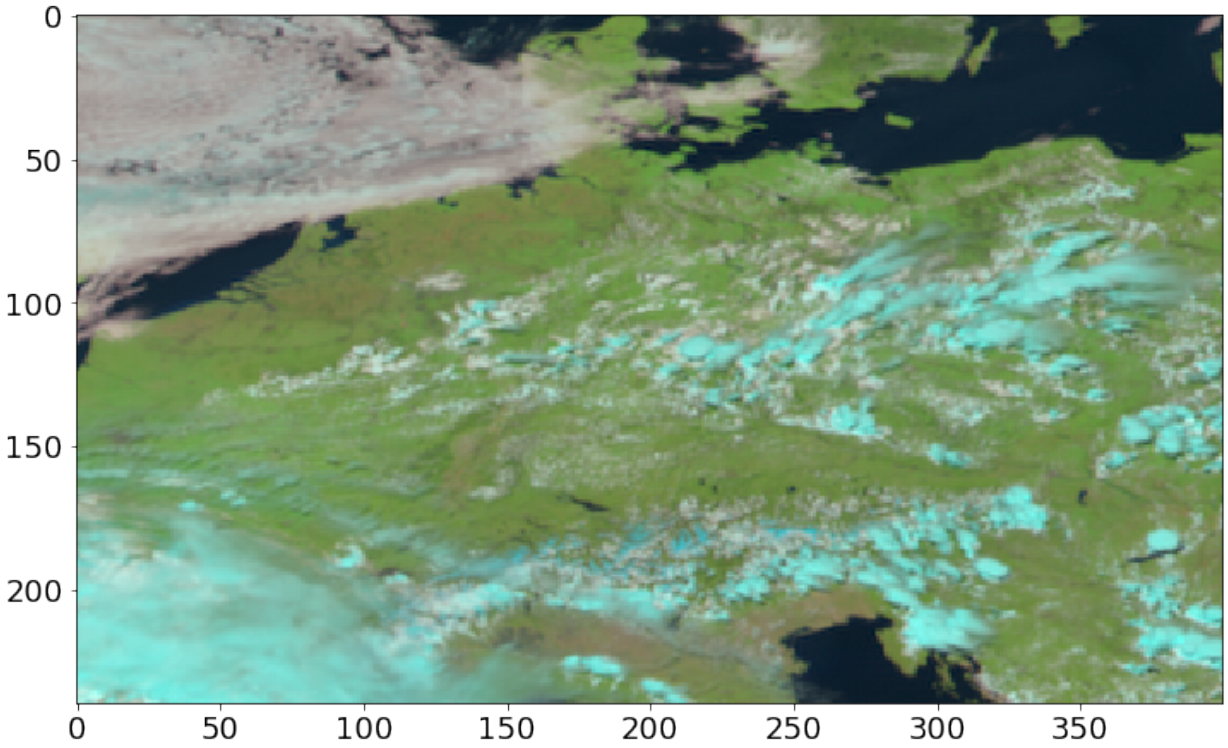
4.2.4 Adjusting Region Configuration

[12]:
```
region = ((216, 456), (1676, 2076))
s = MSeviRGB(time = time, region = region, scan_type = scan_type, tstamp = False)
```

```
Region suggests use of hdf file
```

[13]:
```
rgb = np.array( s.images['pytroll_nc'] )
plt.imshow(rgb)
```

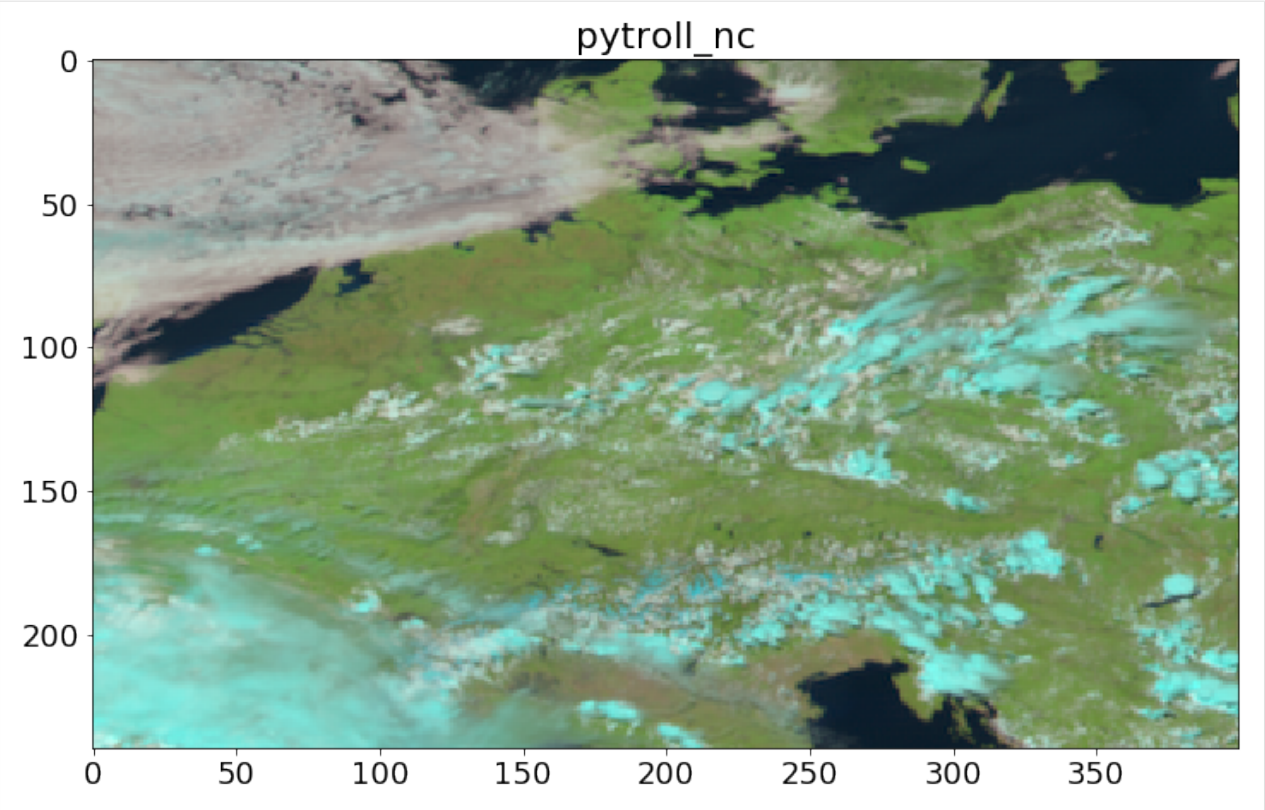
[13]:
```
<matplotlib.image.AxesImage at 0x7ff8db46f810>
```

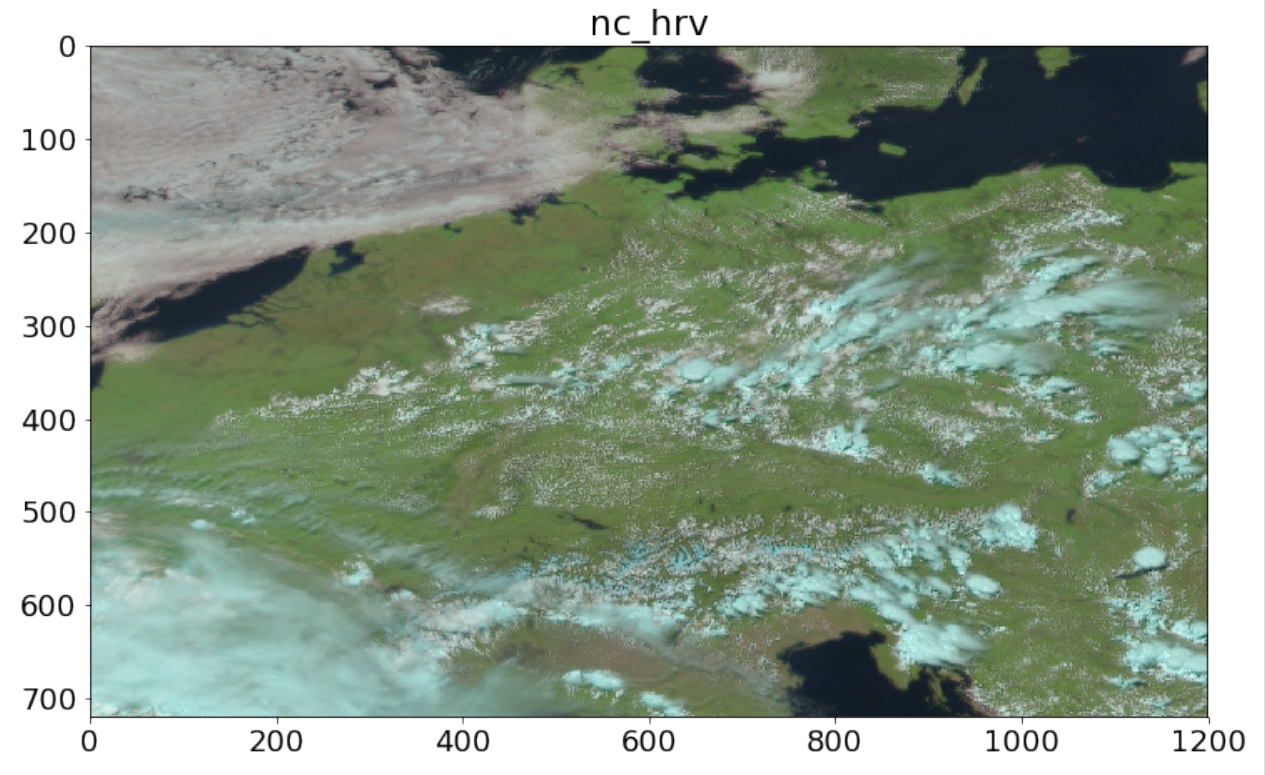


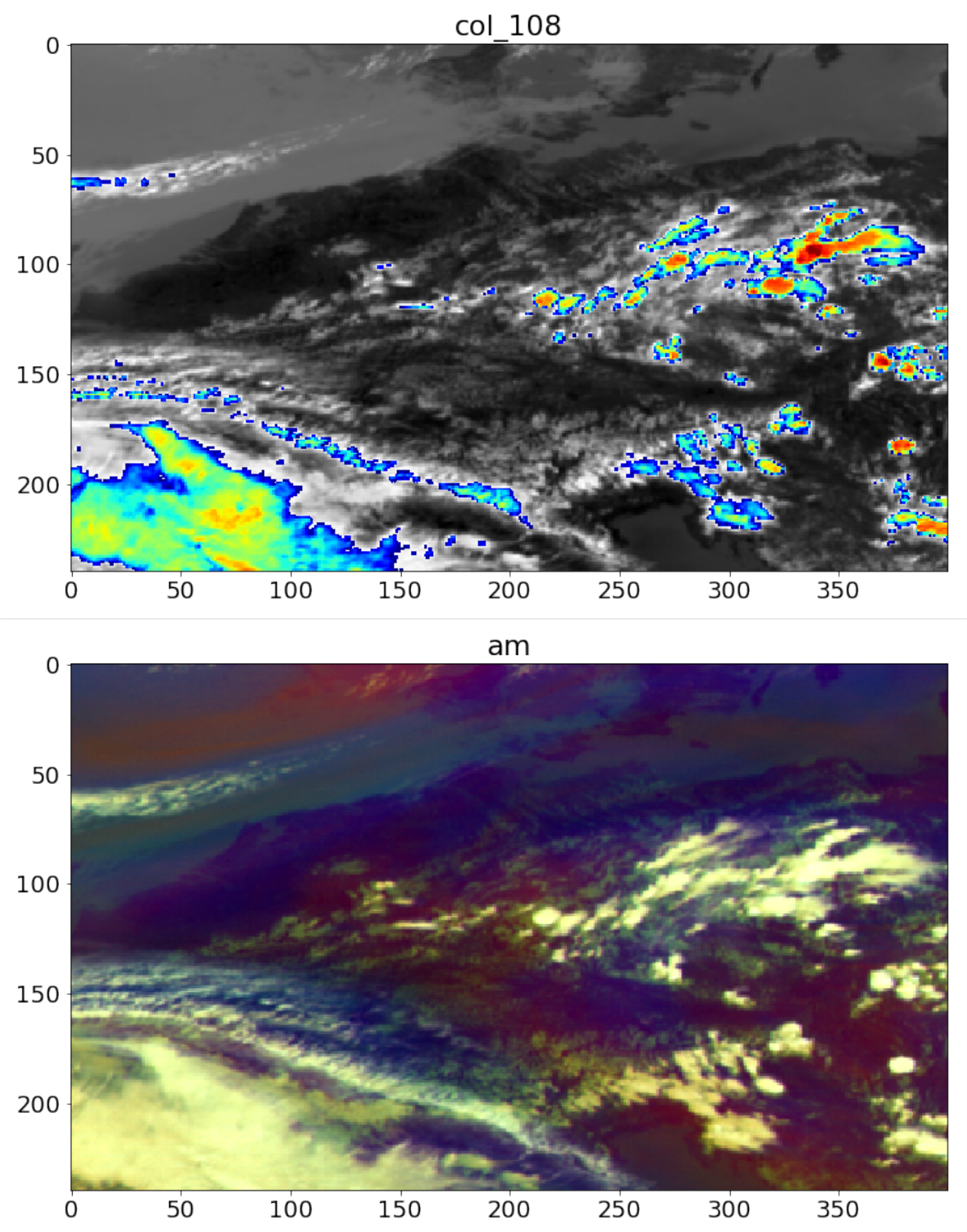
4.2.5 Loop over several RGBs

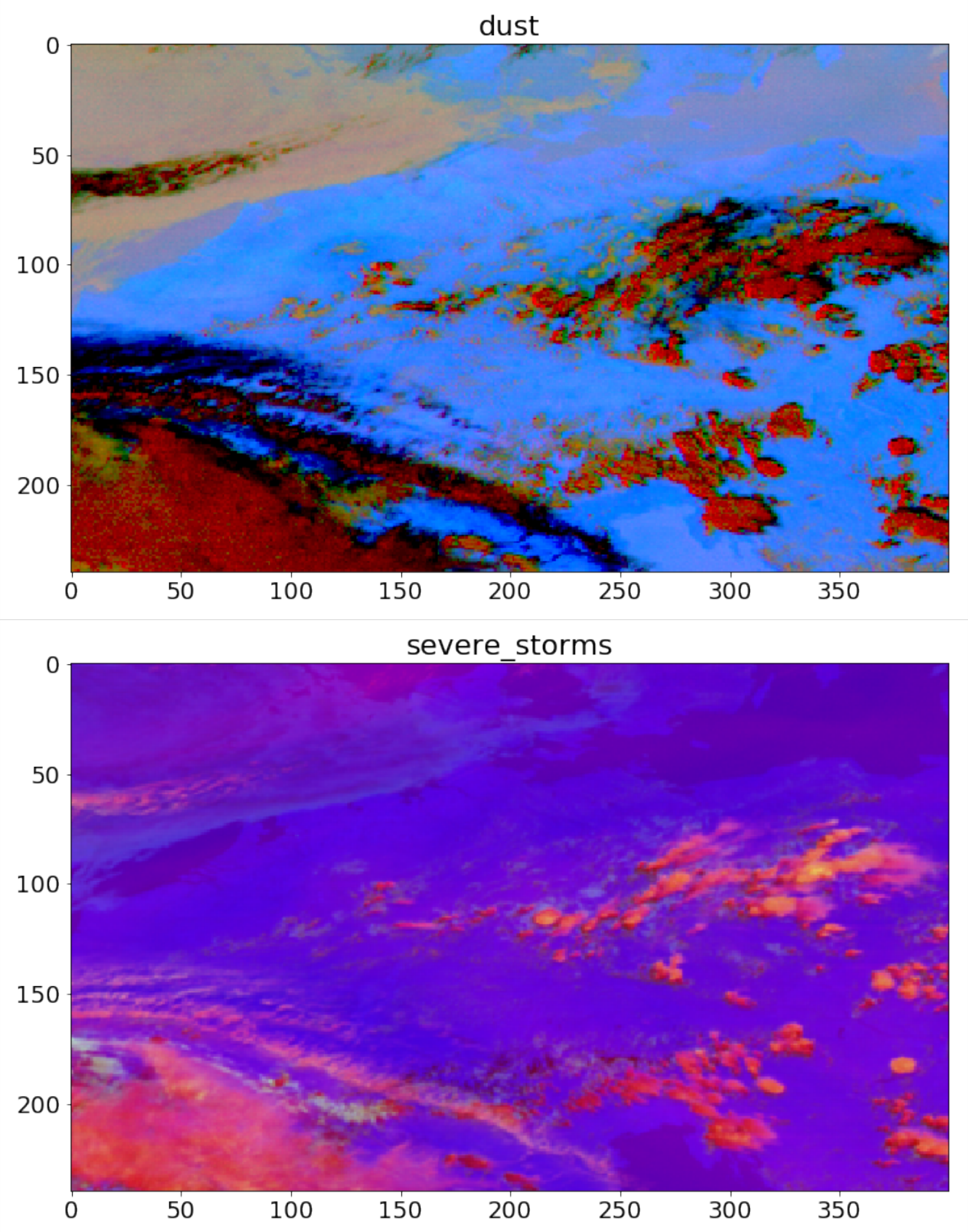
[16]:
```
rgb_list = ['pytroll_nc',  'nc_hrv', 'col_108', 'am', 'dust', 'severe_storms', 'hrv_
↪clouds']

for rgbname in rgb_list:
    plt.figure()
    s.create_rgb(rgbname, tstamp=False)
    rgb = np.array( s.images[rgbname] )
    plt.imshow(rgb)
    plt.title(rgbname)
```

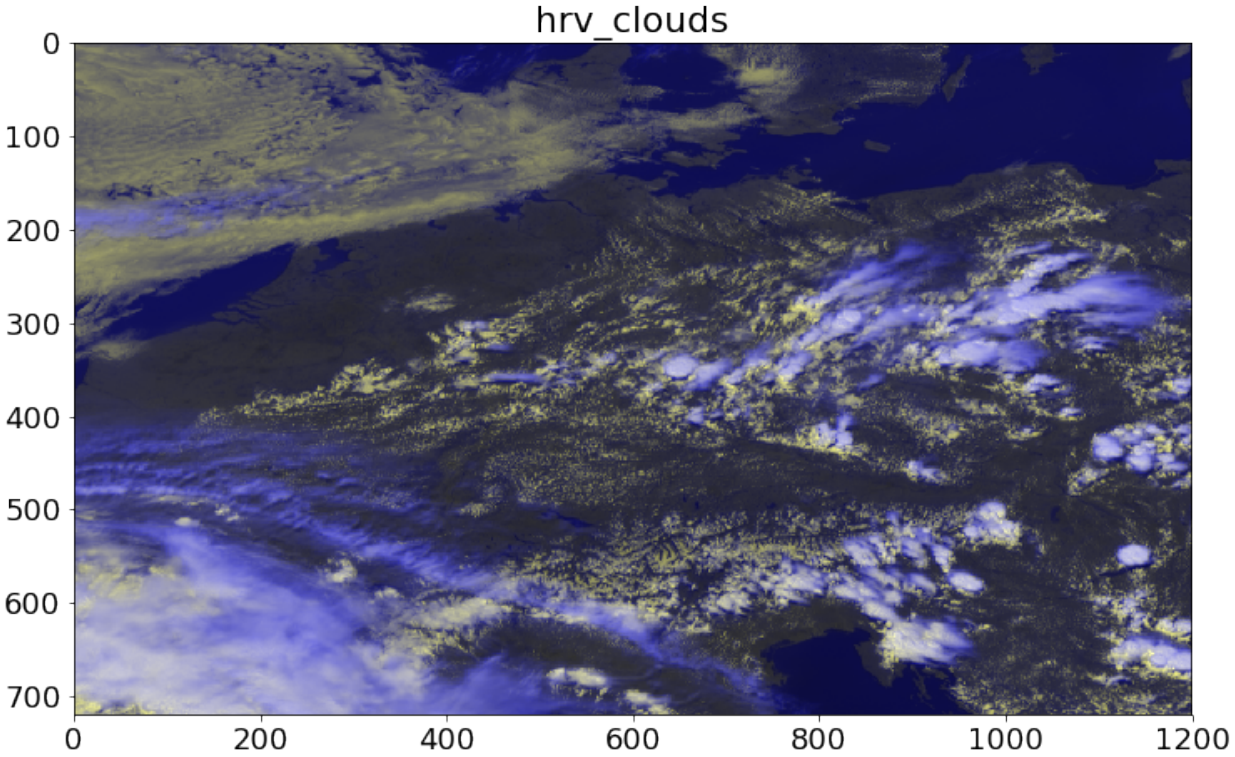
```
['IR_016', 'VIS008', 'VIS006'] is already loaded!
['IR_016', 'VIS008', 'VIS006', 'HRV'] is already loaded!
['IR_108'] is already loaded!
['WV_062', 'WV_073', 'IR_097', 'IR_108'] is already loaded!
['IR_087', 'IR_108', 'IR_120'] is already loaded!
['WV_062', 'WV_073', 'IR_039', 'IR_016', 'IR_108', 'VIS006'] is already loaded!
['IR_108', 'HRV'] is already loaded!
```


pytroll_nc
0
50
100
150
200
0
50
100
150
200
250
300
350


nc_hrv
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


col_108
0
50
100
150
200
0
50
100
150
200
250
300
350
am
0
50
100
150
200
0
50
100
150
200
250
300
350


dust
0
50
100
150
200
0
50
100
150
200
250
300
350
severe_storms
0
50
100
150
200
0
50
100
150
200
250
300
350



[ ]:

4.3 Making a Shaded Plot with a Non-linear Colormap

This tutorial shows how to use the module `tropy.plotting_tools.shaded` to make shaded plot with a non-linear colormap.

4.3.1 Import Libraries

[1]:
```
%matplotlib inline

# standard libs
import numpy as np
import scipy.ndimage

# plotting and mapping
import pylab as plt
import seaborn as sns
sns.set_context('talk')

# the own tropy lib
import tropy.plotting_tools.shaded
```

4.3.2 Making Example Data

We make some random example data for plotting.

[2]:
```
nrow, ncol = 180, 200
x = np.linspace(0, 1, ncol)
y = np.linspace(0, 1, nrow)

r = 4 * np.random.randn( nrow, ncol )

# smoothing
r_sm = scipy.ndimage.gaussian_filter(r, 2 )


# non-linear transformation
dset = np.exp( r_sm )
```

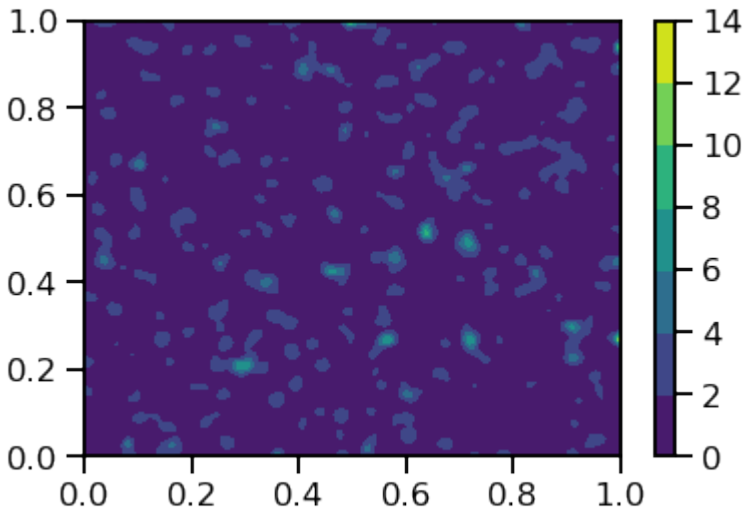
4.3.3 Standard Plotting

Internally Defined Levels

First, we plot the random data with internally defined levels.

[3]:
```
plt.contourf(x, y, dset)
plt.colorbar()
```

[3]:
```
<matplotlib.colorbar.Colorbar at 0x7f808ced1cc0>
```



Nonlinear Levels

I implemented a functionality to generate simple non-linear level in `tropy.plotting_tools.shaded`

[4]:
```
help( tropy.plotting_tools.shaded.set_levs )
```

```
Help on function set_levs in module tropy.plotting_tools.shaded:
```

(continues on next page)

(continued from previous page)

```
set_levs(nmax, depth, largest=5, sym=True, sign=True, zmax='None')
    Makes a non-linear level set based on pre-defined numbers.


    Parameters
    ----------
    nmax : int
        exponent of maximum number

    depth : int
        number of iterations used down to scales of 10**(nmax - depth)

    largest : {5, 8}, optional
        set the largest number in the base array either to 5 or 8

    sym : {True, False}, optional
        switch if levels are symmetric around origin

    sign : {True, False}, optional
        switches sign if negative

    zmax : float, optional, default = 'none'
        limiter for levels, |levs| > zmax are not allowed


    Returns
    --------
    levs : np.array
        set of non-linear levels
```

Let's first play with the function.

[5]:
```
tropy.plotting_tools.shaded.set_levs( 0, 1 )
```

[5]:
```
array([-5., -3., -2., -1.,  0.,  1.,  2.,  3.,  5.])
```

[6]:
```
tropy.plotting_tools.shaded.set_levs( 1, 1 )
```

[6]:
```
array([-50., -30., -20., -10.,   0.,  10.,  20.,  30.,  50.])
```

[7]:
```
tropy.plotting_tools.shaded.set_levs( 0, 2 )
```

[7]:
```
array([-5. , -3. , -2. , -1. , -0.5, -0.3, -0.2, -0.1,  0. ,  0.1,  0.2,
        0.3,  0.5,  1. ,  2. ,  3. ,  5. ])
```

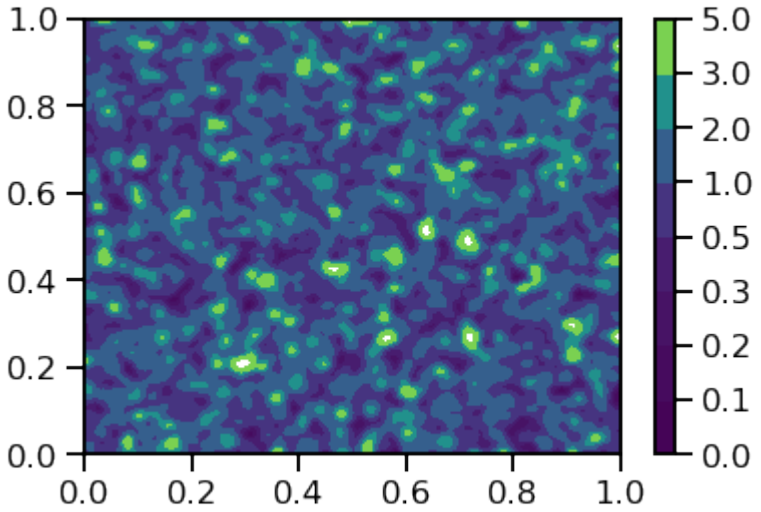
Okay, we see how different levels and depths can be set...

[8]:
```
levels = tropy.plotting_tools.shaded.set_levs( 0, 2, sym = False )
levels
```

[8]:
```
array([0. , 0.1, 0.2, 0.3, 0.5, 1. , 2. , 3. , 5. ])
```

[9]:
```
plt.contourf(x, y, dset, levels, vmax = levels.max() )
plt.colorbar()
```

[9]:
```
<matplotlib.colorbar.Colorbar at 0x7f808cdb1b70>
```



4.3.4 Non-linear Shadings

Let's look at the plotting function.

[10]:
```
help( tropy.plotting_tools.shaded.shaded )
```

```
Help on function shaded in module tropy.plotting_tools.shaded:

shaded(x, y, z, *args, **kwargs)
    The function shaded is a wrapper for the pylab function contourf.


    In addition to contourf, shaded can plot filled contours for which
    a non-linear colormap is used.


    Parameters
    ----------
    x : np.array
        x-values passed to `plt.contourf`

    y : np.array
        y-values passed to `plt.contourf`

    z : np.array
        z-values passed to `plt.contourf` (color value)

    *args : list
        other positional arguments passed to `plt.contourf`

    **kwargs : dict
        other optional arguments passed to `plt.contourf`

        special keywords:

        * 'levels' : numpy array of color / contour levels
```

(continues on next page)

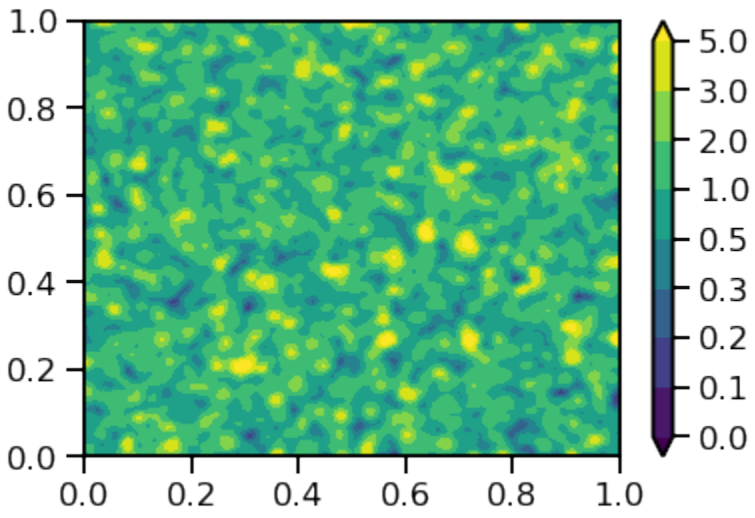
(continued from previous page)

```
        * 'lev_depth' : gives the depth of an automatically generated level set
          i.e. an initial base (e.g. [1,2,3,5] ) that contains
          the maximum value of the field (e.g. 4.3) is downscaled
          by 10**-1, ..., 10**-lev_depth.

          Example:
          Given lev_depth = 2, for a positive field with maximum 4.3
          a level set [0.01, 0.02, 0.03, 0.05, 0.1, 0.2, 0.3, 0.5, 1, 2, 3, 5]
          is generated.

    Returns
    --------
    pl.contourf instance
```

With Pre-Defined Levels

[11]:
```
tropy.plotting_tools.shaded.shaded(x, y, dset, levels = levels, vmax = levels.max(),␣
↪cmap = plt.cm.viridis )
plt.colorbar()
```

[11]:
```
<matplotlib.colorbar.Colorbar at 0x7f8088c204e0>
```



With Internally-Defined Levels

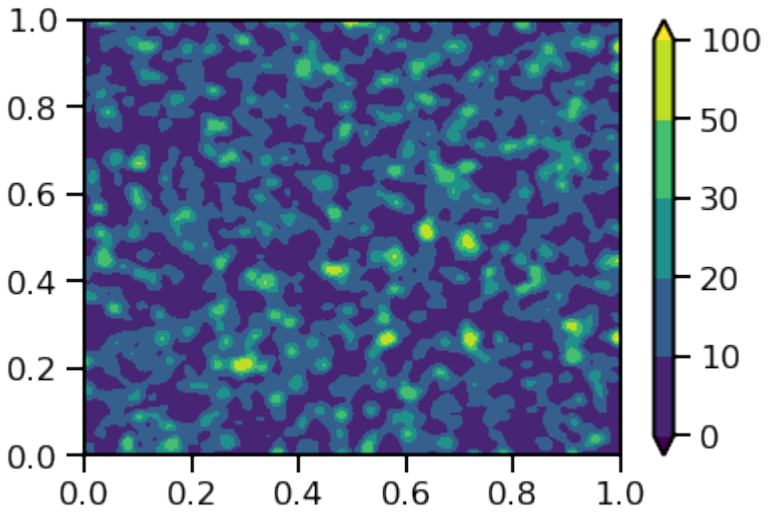
It is also possible to run the plotting with internally generated levels:

- using only one "cascade" (`lev_depth = 1`)

[12]:
```
tropy.plotting_tools.shaded.shaded(x, y, 10 * dset, cmap = plt.cm.viridis, lev_depth␣
↪= 1 )
plt.colorbar()
```

```
/vols/fs1/store/senf/.conda/python37/lib/python3.7/site-packages/tropy/plotting_tools/
↪shaded.py:281: UserWarning: The following kwargs were not used by contour: 'lev_
↪depth'
  return py.contourf(x, y, z, *args, **kwargs)
```

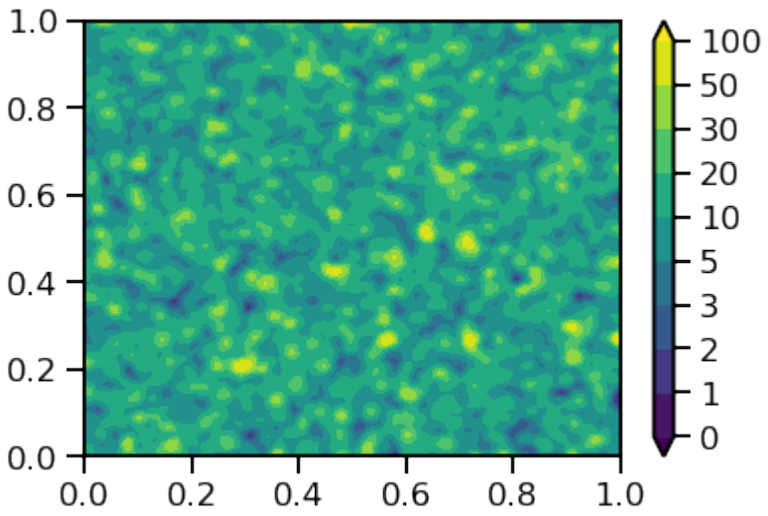
[12]:
```
<matplotlib.colorbar.Colorbar at 0x7f8088b17b70>
```



- or using e.g. two "cascades" (`lev_depth = 2`)

[14]:
```
tropy.plotting_tools.shaded.shaded(x, y, 10 * dset, cmap = plt.cm.viridis, lev_depth␣
↪= 2 )
plt.colorbar()
```

[14]:
```
<matplotlib.colorbar.Colorbar at 0x7f80889247f0>
```



4.3.5 Summary

The module `tropy.plotting_tools.shaded` help you to

- automatically generate a set of non-linear levels based on simple digits, e.g. [1, 2, 3, 5]

- to use non-linear levels in filled contour plots.

4.4 Plots with Self-Defined Colormaps

This tutorial shows how to use the module `tropy.plotting_tools.colormaps` to make plots with the colormaps provided by this module.

4.4.1 Import Libraries

[1]:
```
%matplotlib inline

# standard libs
import numpy as np
import scipy.ndimage

# plotting and mapping
import pylab as plt
import seaborn as sns
sns.set_context('talk')
plt.rcParams['figure.figsize'] = (8, 6)

# the own tropy lib
import tropy.plotting_tools.colormaps
import tropy.plotting_tools.shaded
```

4.4.2 Making Example Data

We make some random example data for plotting.

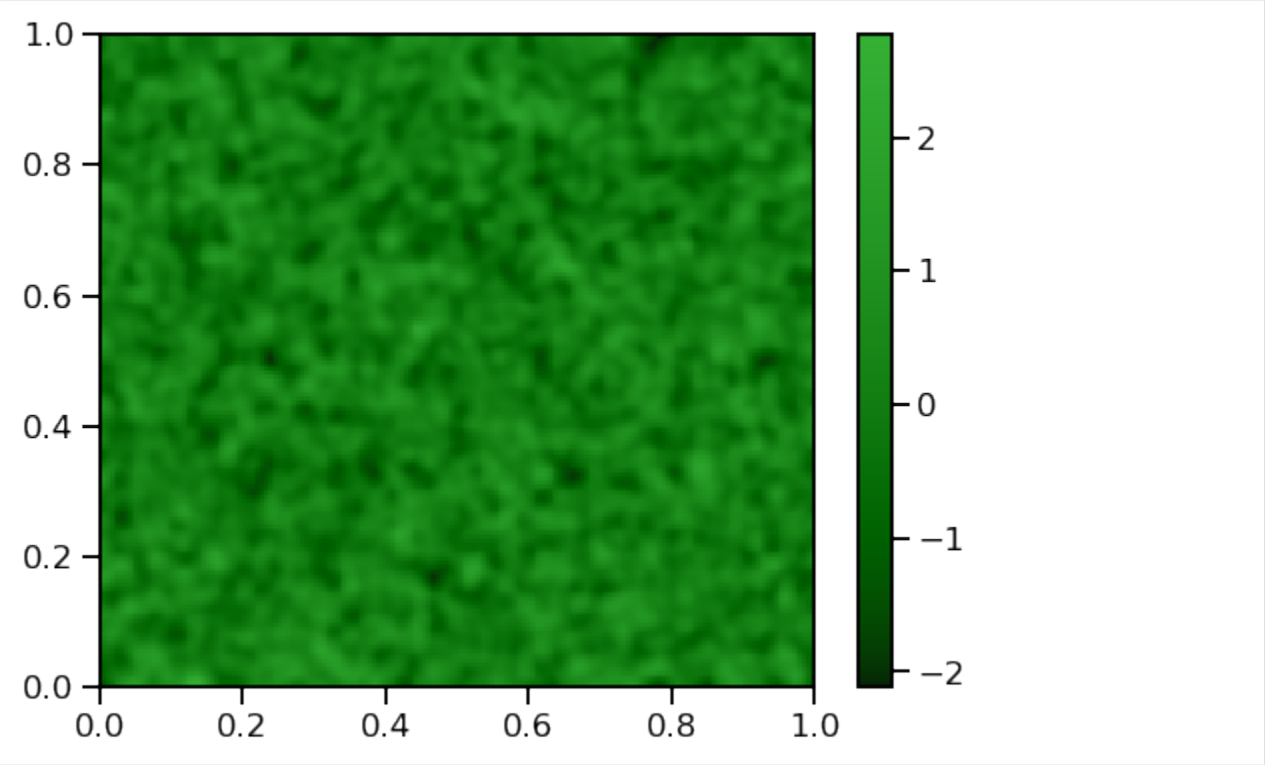
[2]:
```
nrow, ncol = 180, 200
x = np.linspace(0, 1, ncol)
y = np.linspace(0, 1, nrow)

r = 4 * np.random.randn( nrow, ncol )

# smoothing
r_sm = scipy.ndimage.gaussian_filter(r, 2 )
```

4.4.3 Colormap based on Colorname

We take the random data and test different colormap options.

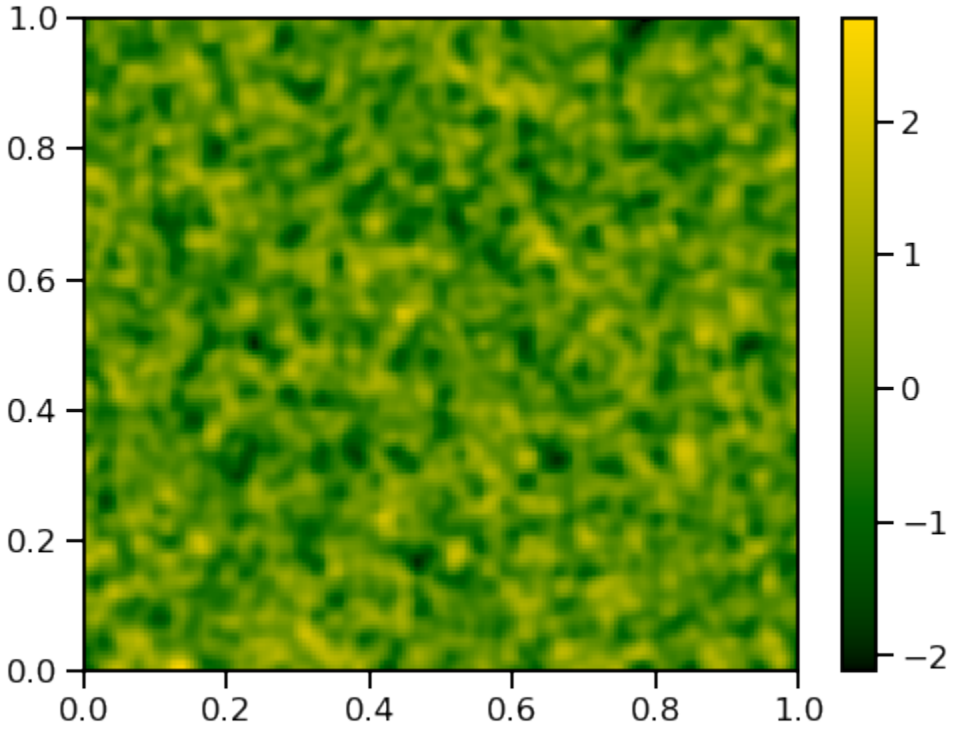
[3]:
```
cmap = tropy.plotting_tools.colormaps.colorname_based_cmap( 'darkgreen')
plt.pcolormesh(x, y, r_sm, cmap = cmap)
plt.colorbar()
```

[3]:
```
<matplotlib.colorbar.Colorbar at 0x7f484e2bdc10>
```



[4]:
```
cmap = tropy.plotting_tools.colormaps.colorname_based_cmap( 'darkgreen',
                                                            start_col = 'black',
                                                            final_col = 'gold')

plt.pcolormesh(x, y, r_sm, cmap = cmap)
plt.colorbar()
```

[4]:
```
<matplotlib.colorbar.Colorbar at 0x7f484e1b8f50>
```



Hmm, don't know if you need this … let's look at the next functions.

4.4.4 "Nice" Colormaps

We loop over the different colormap choices.

[5]:
```
help( tropy.plotting_tools.colormaps.nice_cmaps )
```

```
Help on function nice_cmaps in module tropy.plotting_tools.colormaps:

nice_cmaps(cmap_name)
    Some pre-defined "nice" colormaps.


    Parameters
    ----------
    cmap_name : str
        name of a pre-defined colormap.

        * 'red2blue_disc'           :  a discrete red to blue
          transistion
        * 'red2blue'                :  a contineous red to blue
          transistion
        * 'white-green-orange'      :  a green-based transition
          from white to orange
        * 'white-blue-green-orange' :  a blue-based transition
          from white to orange
        * 'white-purple-orange'     :  a purple-based transistion
          from white to orange
        * 'ocean-clouds'            :  for clouds over ocean
```

(continues on next page)

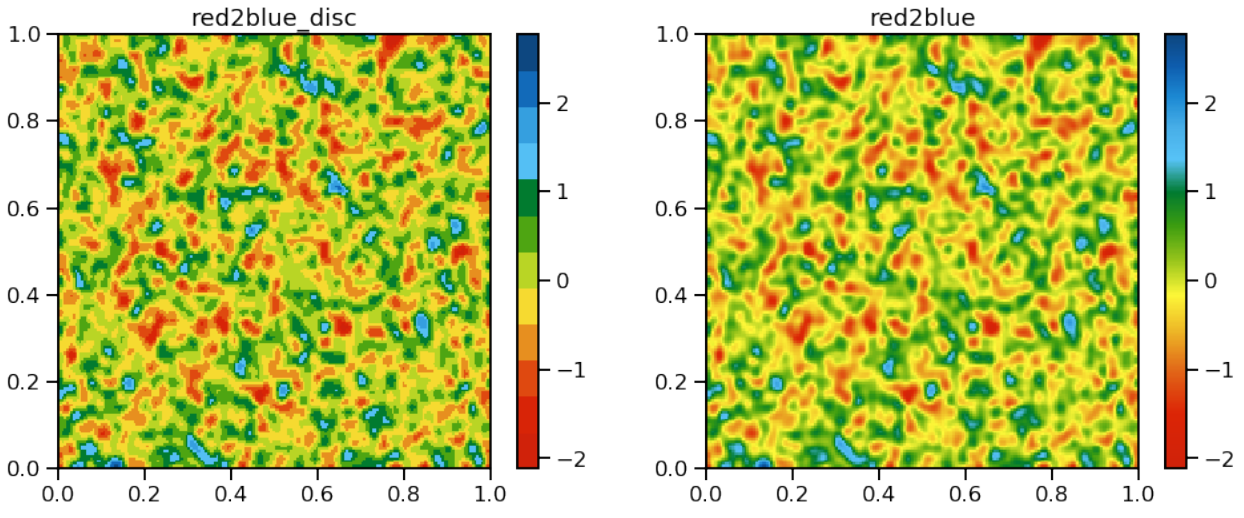
(continued from previous page)

```
        * 'land-clouds'              :  for clouds over land


    Returns
    -------
    cmap : matplotlib colormap object
        resulting colormap
```

[6]:
```
colormap_names = [ 'red2blue_disc',           #   a discrete red to blue transistion
                   'red2blue' ,               #   a continuous red to blue transistion
                   'white-green-orange',      #   a green-based transition from white␣
↪to orange
                   'white-blue-green-orange', #   a blue-based transition from white␣
↪to orange
                   'white-purple-orange',     #   a purple-based transistion from␣
↪white to orange
                   'ocean-clouds',            #   for clouds over ocean
                   'land-clouds'              #   for clouds over land
                 ]
```

Another Rainbow

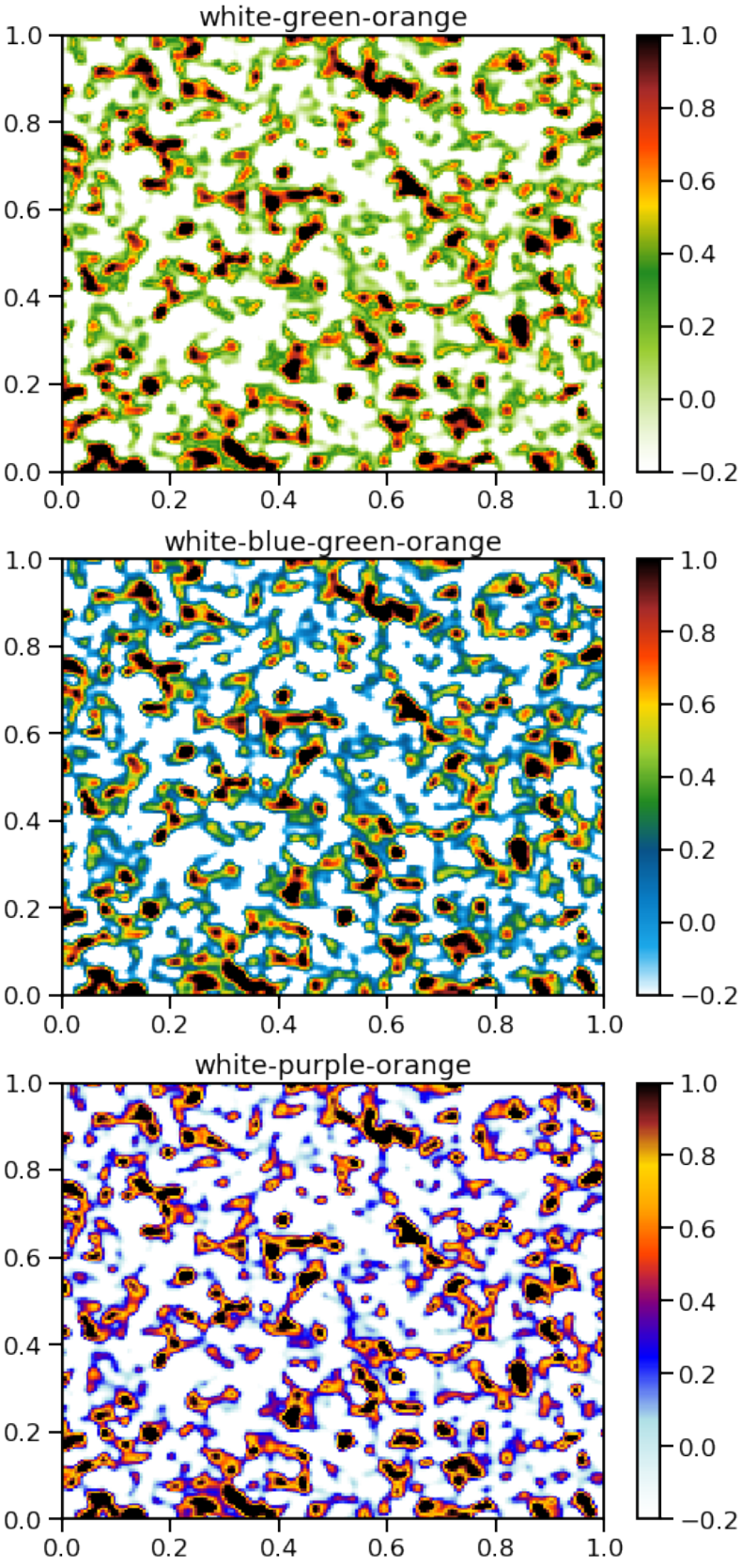
[7]:
```
fig, axs = plt.subplots(nrows = 1, ncols = 2, figsize = (16,6))
axs = axs.flatten()

for i, colmap in enumerate( colormap_names[0:2] ):

    cmap = tropy.plotting_tools.colormaps.nice_cmaps( colmap )

    plt.sca( axs[i] )
    plt.title( colmap )
    plt.pcolormesh(x, y, r_sm, cmap = cmap)
    plt.colorbar()
```



Starting from "nowhere"

The colormaps start with white and are useful for positive-definite fields, like PDFs.

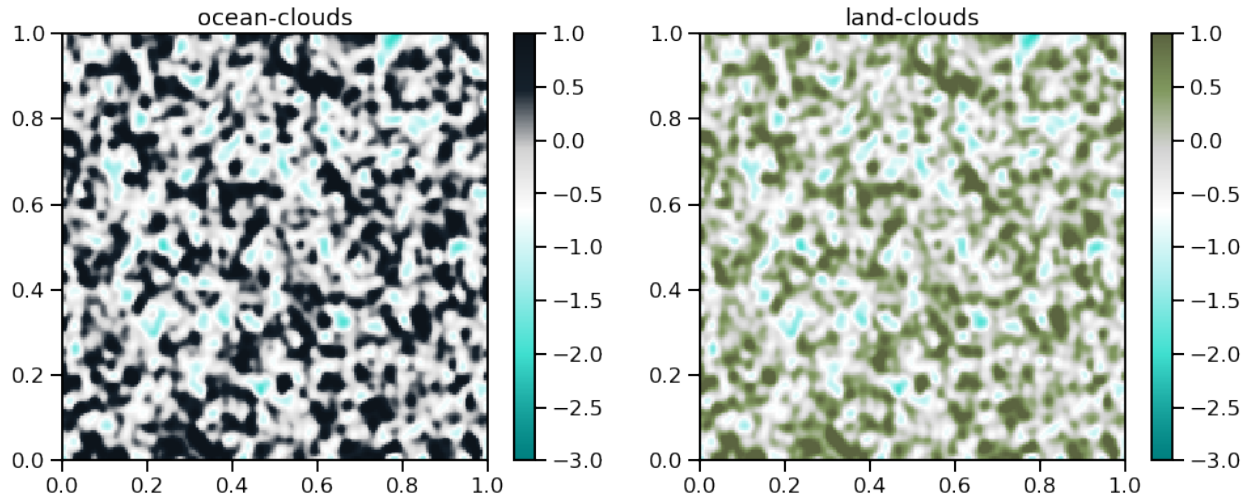
[8]:
```
fig, axs = plt.subplots(nrows = 3, ncols = 1, figsize = (8,18))
axs = axs.flatten()
plt.subplots_adjust( wspace = 0.4 )

for i, colmap in enumerate( colormap_names[2: 5] ):

    cmap = tropy.plotting_tools.colormaps.nice_cmaps( colmap )

    plt.sca( axs[i] )
    plt.title( colmap )
    plt.pcolormesh(x, y, r_sm, cmap = cmap, vmin = -0.2, vmax = 1.)
    plt.colorbar()
```


white-green-orange
white-blue-green-orange
white-purple-orange
0.0
0.2
0.4
0.6
0.8
1.0
−0.2

Clouds

The colormaps are for clouds over different undergrounds - to mimick the visible satellite composites.

[9]:
```
fig, axs = plt.subplots(nrows = 1, ncols = 2, figsize = (16,6))
axs = axs.flatten()

for i, colmap in enumerate( colormap_names[-2:] ):

    cmap = tropy.plotting_tools.colormaps.nice_cmaps( colmap )

    plt.sca( axs[i] )
    plt.title( colmap )
    plt.pcolormesh(x, y, r_sm, cmap = cmap, vmin = -3., vmax = 1.)
    plt.colorbar()
```



4.4.5 Colormaps for Satellite Data

The function `enhanced_colormap` is for 10.8 um brightness temperatures from satellite (e.g. for Meteosat SEVIRI).

We scale our random data in the typical range (200 K, 300 K)

[10]:
```
help( tropy.plotting_tools.colormaps.enhanced_colormap)
```

```
Help on function enhanced_colormap in module tropy.plotting_tools.colormaps:

enhanced_colormap(vmin=200.0, vmed=240.0, vmax=300.0)
    Standard color enhancement used for infrared satellite brightness
    temperatures (BTs) in the atmospheric window around 10.8 um.


    Parameters
    ----------
    vmin : float, optional
        minimum BT

    vmed : float, optional
        BT where transition between gray-scale and rainbow
```

(continues on next page)

(continued from previous page)

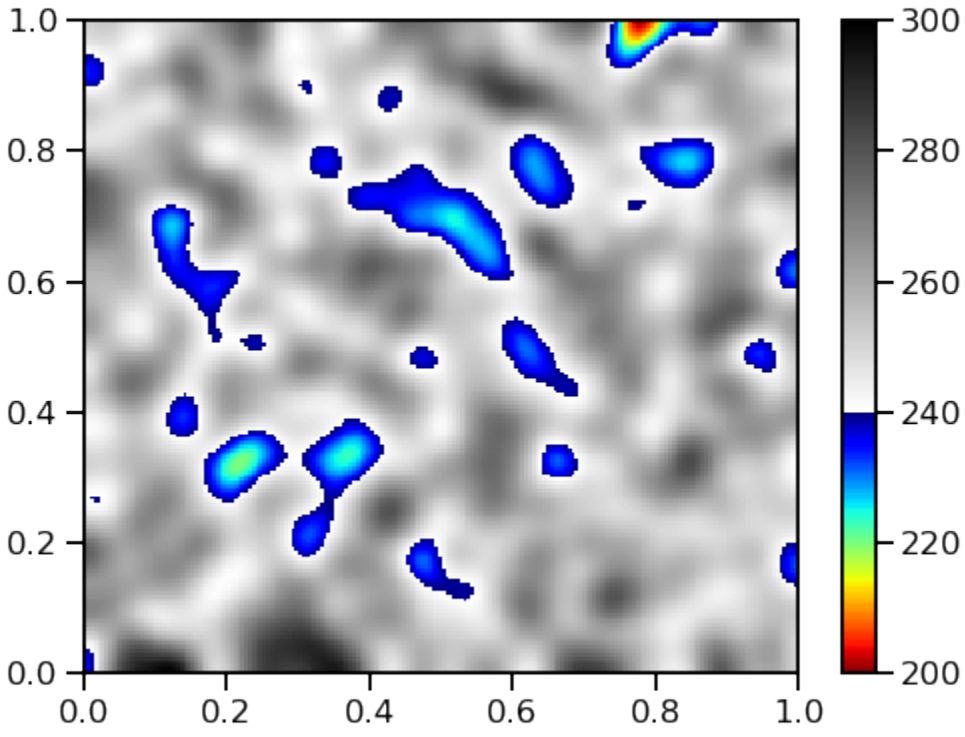
```
        colors is done

    vmax : float, optional
        maximum BT


    Returns
    -------
    mymap : matplotlib colormap object
        resulting colormap
```

[11]:
```
b = scipy.ndimage.gaussian_filter( r_sm, 5)
bn =  (b - b.min()) / (b.max() - b.min())
```

[12]:
```
bt108 = 100 * bn + 200
```

[13]:
```
cmap = tropy.plotting_tools.colormaps.enhanced_colormap( )
plt.pcolormesh(x, y, bt108, cmap = cmap, vmin = 200, vmax = 300)
plt.colorbar()
```

[13]:
```
<matplotlib.colorbar.Colorbar at 0x7f484c60f0d0>
```



The function `enhanced_wv62_cmap` is for the brightness temperature in the water vapor channel (e.g. for Meteosat SEVIRI).

We scale our random data in the typical range (200 K, 260 K)

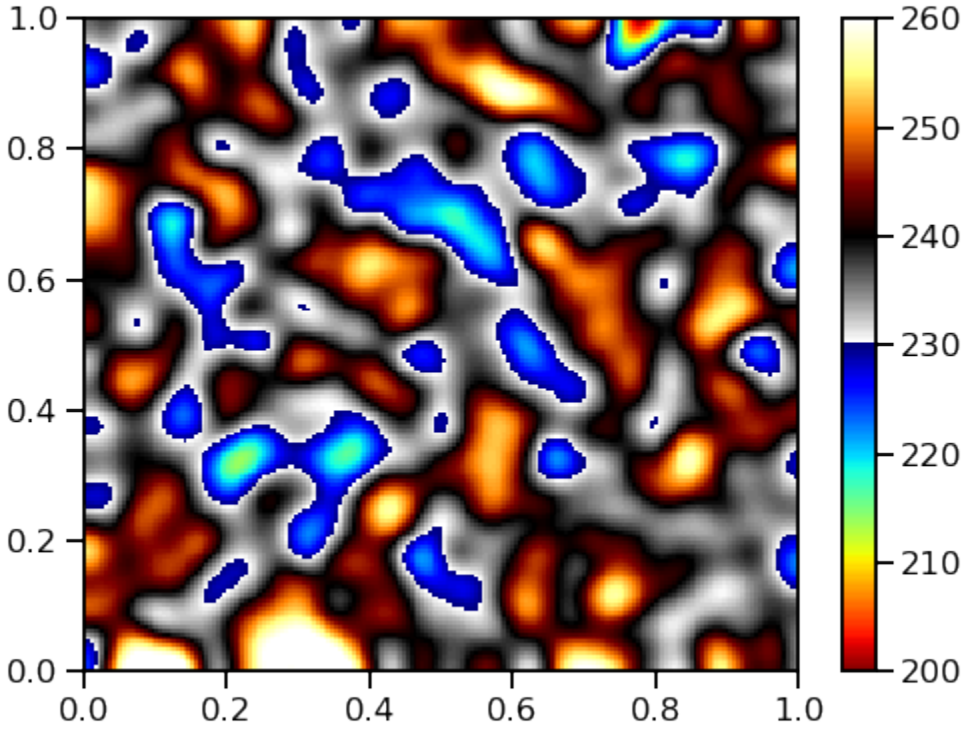
[14]:
```
help(tropy.plotting_tools.colormaps.enhanced_wv62_cmap)
```

```
Help on function enhanced_wv62_cmap in module tropy.plotting_tools.colormaps:

enhanced_wv62_cmap(vmin=200.0, vmed1=230.0, vmed2=240.0, vmax=260.0)
    Color enhancement (non-standard - more artistic) used for infrared satellite
    brightness temperatures (BTs) for water vapor channels.


    Parameters
    ----------
    vmin : float, optional
        minimum BT

    vmed1 : float, optional
        BT where first transition between brownish colors
        and gray-scale is done

    vmed2 : float, optional
        BT where second transition between gray-scale
        and rainbow colors is done

    vmax : float, optional
        maximum BT


    Returns
    -------
    mymap : matplotlib colormap object
        resulting colormap
```

[15]:
```
bt062 = 70 * bn + 200
```

[16]:
```
cmap = tropy.plotting_tools.colormaps.enhanced_wv62_cmap( )
plt.pcolormesh(x, y, bt062, cmap = cmap, vmin = 200, vmax = 260)
plt.colorbar()
```

[16]:
```
<matplotlib.colorbar.Colorbar at 0x7f484c92e210>
```



4.4.6 Colormap for Radar Data

This colormap is taken from the DWD website to plot rain accumulations.

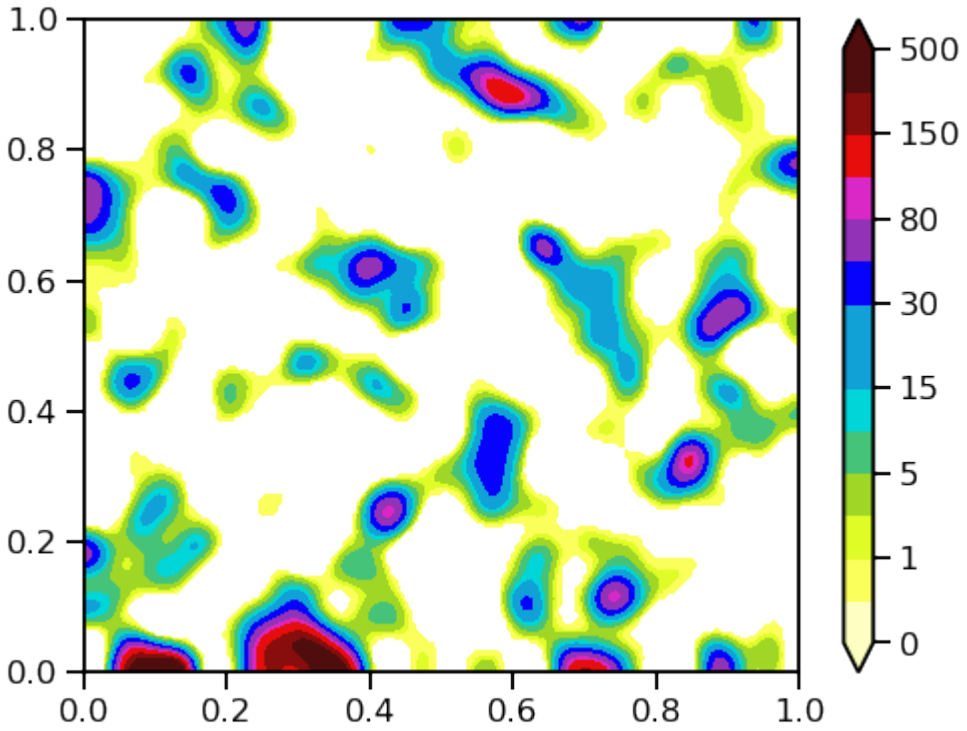
[17]:
```
help( tropy.plotting_tools.colormaps.dwd_sfmap )
```

```
Help on function dwd_sfmap in module tropy.plotting_tools.colormaps:

dwd_sfmap()
    Colormap for DWD Radolan SF product for observed daily rain
    accumulations.


    Returns
    -------
    rrlevs : list
        rain rate levels (in mm)

    cmap :  matplotlib colormap object
        SF colormap
```

[18]:
```
rr = 600 * b**3
rr = np.ma.masked_less(rr, 0.1)
```

[19]:
```
rrlevs, cmap = tropy.plotting_tools.colormaps.dwd_sfmap( )
tropy.plotting_tools.shaded.shaded(x, y, rr, cmap = cmap, levels = rrlevs)
plt.colorbar()
```

[19]: <matplotlib.colorbar.Colorbar at 0x7f484c6bf590>



4.4.7 Summary

The module `tropy.plotting_tools.colormaps` provides you some self-defined colormaps, esp.

- for satellite data
- positive-definite fields (like pdfs) which start from zero (white)

4.5 Adding Meta Data to a PNG File

This tutorial shows how to use the module `tropy.plotting_tools.meta2png`.

4.5.1 Introduction

Problem Statement

We often have the situation that an image is generated with a python script. If you find that image (years) later, you like to have an easy way to find the script that made the image.

Suggested Solution

A possible solution is to write source filename and author name into the image meta data. Then, you can search for this info in the meta data. This info also remains attached to the file once you distribute your image.

A linux-based command to screen the meta data is

```
identify -verbose  ${IMAGE_NAME} | grep Author
identify -verbose  ${IMAGE_NAME} | grep Source
```

4.5.2 Import Libraries

[1]:
```
%matplotlib inline

# standard libs
import numpy as np

# plotting and mapping
import pylab as plt

# the own tropy lib
from tropy.plotting_tools.meta2png import pngsave
```

4.5.3 Making Example Data

We make some random example data for plotting.

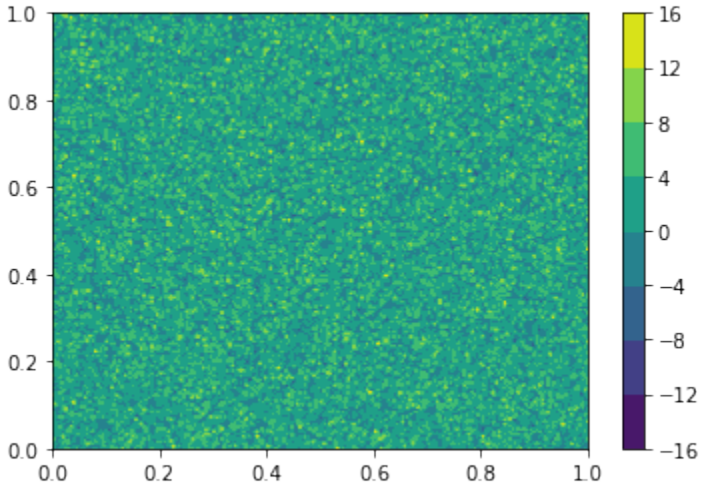
[2]:
```
nrow, ncol = 180, 200
x = np.linspace(0, 1, ncol)
y = np.linspace(0, 1, nrow)

r = 4 * np.random.randn( nrow, ncol )
```

4.5.4 Make a Plot

First, we plot the random data with internally defined levels.

[3]:
```
plt.contourf(x, y, r)
plt.colorbar()
```

[3]:
```
<matplotlib.colorbar.Colorbar at 0x7fc24c2b4b38>
```



4.5.5 Saving the Plot as PNG

Standard Way

[4]:
```
plt.savefig('test.png')
```

```
<Figure size 432x288 with 0 Axes>
```

I implemented a functionality for adding metadata to `tropy.plotting_tools.meta2png`. We use the class `pngsave` similarily to `plt.savefig`

[5]:
```
help( pngsave )
```

```
Help on class pngsave in module tropy.plotting_tools.meta2png:

class pngsave(builtins.object)
 |  pngsave(*args, **kwargs)
 |
 |  That class is designed to save pylab figures in png and add meta data.
 |
 |
 |  Parameters
 |  ----------
 |  *args : list
 |      other positional arguments passed to `plt.savefig`
 |
 |  **kwargs : dict
 |      other optional arguments passed to `plt.savefig``
 |
 |      'author' : Place yeur name into author keyword
 |
 |      'source' : Specify your source filename if needed,
 |                 if not set, it tries to automatically find the filename
 |
 |      'notebook' : {False, True} if you are working in a notebook
 |
 |  Methods defined here:
```

(continues on next page)

(continued from previous page)

```
 |
 |  __call__(self, fname, **kwargs)
 |      Call self as a function.
 |
 |  __init__(self, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  meta(self)
 |      Set the meta data, esp. the Author name and Source file information.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

[6]:
```
pngsave('test_withmeta.png', author = 'Hans Wurst');
```

```
... save image to  test_withmeta.png
```

```
<Figure size 432x288 with 0 Axes>
```

[7]:
```
!identify -verbose test_withmeta.png | grep Author
```

```
    Author: Hans Wurst
```

[8]:
```
!identify -verbose test_withmeta.png | grep Source
```

```
    Source: /vols/fs1/store/senf/proj/2018-05_pypackage_devel/tropy-tutorials/docs/
↪source//vols/fs1/store/senf/.conda/python37/lib/python3.7/site-packages/ipykernel_
↪launcher.py
```

This is the place were the source filename appears.

Bug with Jupyter

- It is well working with python scripts,
- but not with jupyter notebooks. Sorry for this . . . there might be a clever solution in future. . .

4.5.6 Solving the Notebook Issue

Here, I give a solution to the notebook issue. However, it is not a nice and convenient way.

1. You need to run the function below in a notebook (unfortunately it is not working within a python module)
2. And then explicitly provide the name to `pngsave`

Getting the Notebook Name

[33]:
```
from IPython.core.display import Javascript
from IPython.display import display
import os

def get_notebook_name():
    """Returns the name of the current notebook as a string

    From From https://mail.scipy.org/pipermail/ipython-dev/2014-June/014096.html
    """
    display(Javascript('IPython.notebook.kernel.execute("theNotebook = " + \
    "\'"+IPython.notebook.notebook_name+"\'");'))

    #
    nb_full_path = os.path.join(os.getcwd(), theNotebook)

    return os.path.join(os.getcwd(), theNotebook)


def pngkws(name):

    kws = dict()
    kws['author'] = name
    kws['source'] = get_notebook_name()

    return kws
```

[34]:
```
source = get_notebook_name()
print( source )
```

```
<IPython.core.display.Javascript object>
```

```
/vols/fs1/store/senf/proj/2018-05_pypackage_devel/tropy-tutorials/docs/source/Adding_
↪Meta-Data_to_PNG_File.ipynb
```

[36]:
```
print( pngkws('Hans Wurst') )
```

```
<IPython.core.display.Javascript object>
```

```
{'author': 'Hans Wurst', 'source': '/vols/fs1/store/senf/proj/2018-05_pypackage_devel/
↪tropy-tutorials/docs/source/Adding_Meta-Data_to_PNG_File.ipynb'}
```

[37]:
```
pngsave('test_withmeta.png', **pngkws('Hans Wurst'));
```

```
<IPython.core.display.Javascript object>
```

```
... save image to  test_withmeta.png
```

```
<Figure size 432x288 with 0 Axes>
```

[38]:
```
!identify -verbose test_withmeta.png | grep Source
```

```
    Source: /vols/fs1/store/senf/proj/2018-05_pypackage_devel/tropy-tutorials/docs/
↪source/Adding_Meta-Data_to_PNG_File.ipynb
```

Now it works! Yeah!

4.5.7 Summary

The module `tropy.plotting_tools.meta2png` help you to

- place meta data into PNG files
- when `savepng` is used it automatically places name and source info into png.

4.6 Rank Transformation and Histogram Matching

This tutorial shows an application of the module `tropy.analysis_tools.statistics`. It shows

- How to transform a (2-dim) input field into a field of ranks.
- How to perform histogram matching.

4.6.1 Import Libraries

[1]:
```
%matplotlib inline

# standard libs
import numpy as np
import scipy.ndimage

# plotting and mapping
import pylab as plt
import seaborn as sns
sns.set_context('talk')
plt.rcParams['figure.figsize'] = (8, 6)

# the own tropy lib
import tropy.analysis_tools.statistics
```

```
SimpleITK not available.
```

4.6.2 Making Example Data

We make some random example data for plotting.

[2]:
```
nrow, ncol = 180, 200

r = 4 * np.random.randn( nrow, ncol )
rpoi = 4 * np.random.poisson( size = (nrow, ncol) )

# smoothing
r_sm = scipy.ndimage.gaussian_filter(r, 2 )


# non-linear transformation
dset = np.exp( r_sm )
```

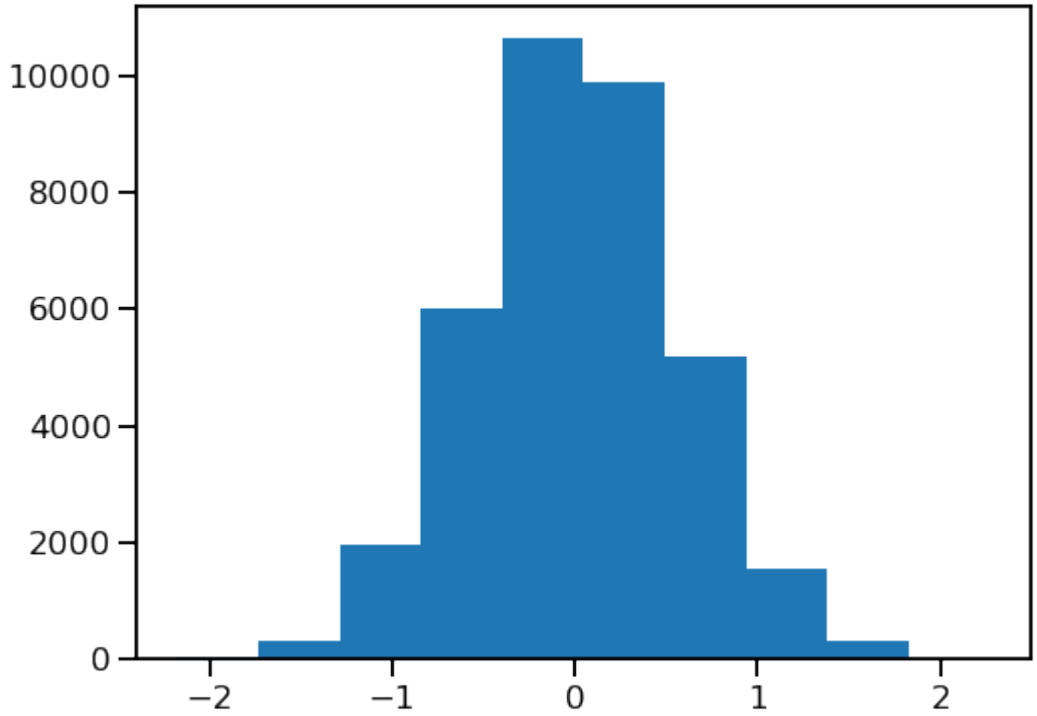
4.6.3 Apply Rank Transformation

The rank transformation sorts the data in ascending order. We return rank data normalized between zero and one, i.e. the minimum gets the zero and the maximum gets the one.

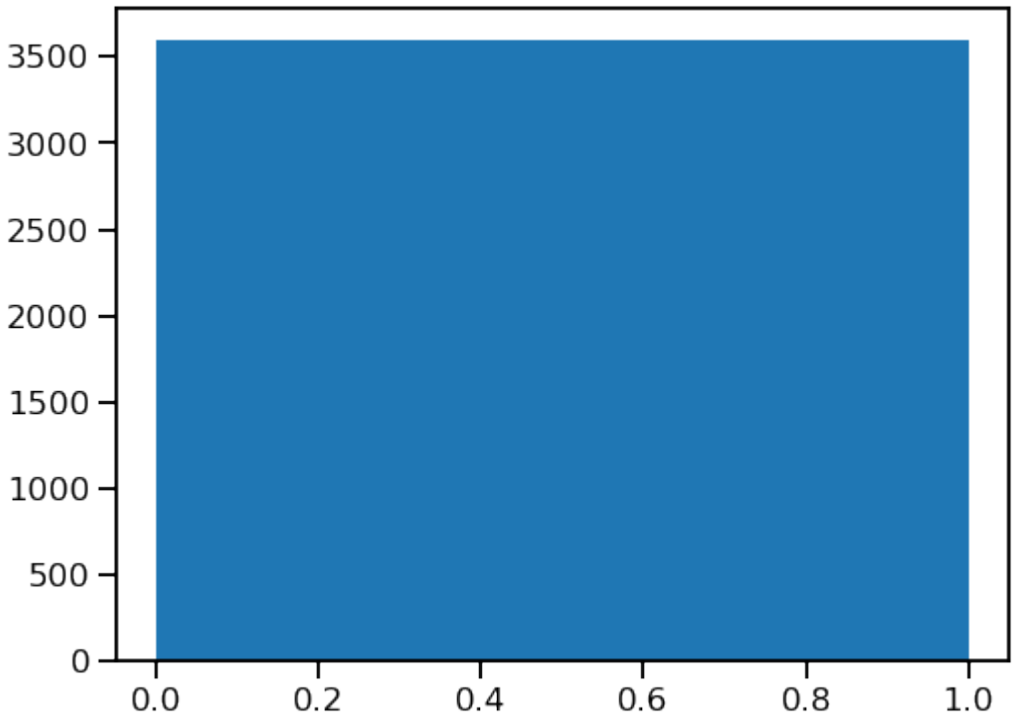
[3]:
```
frank = tropy.analysis_tools.statistics.rank_transformation( r_sm )
```

Check Histograms

[4]:
```
plt.hist( r_sm.flatten() );
```



[5]:
```
plt.hist( frank.flatten() );
```

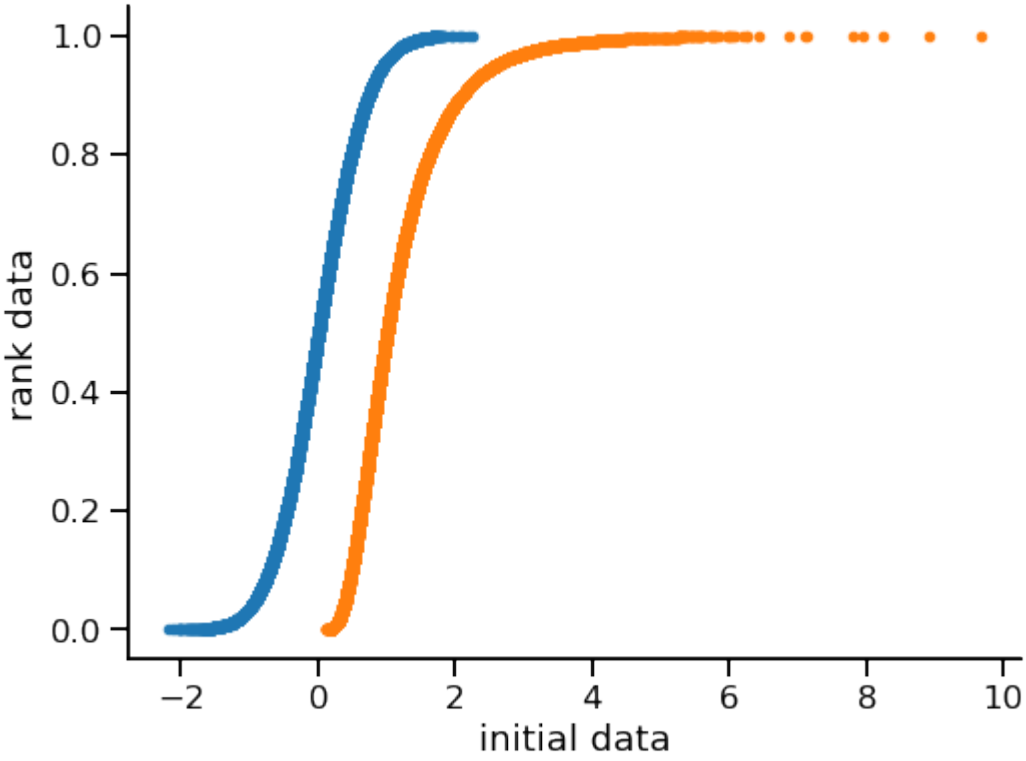


You see that rank data are uniformly distributed that has some advantages for histogram matching. Rank transformation only depends on data order and is same for different non-linear transformations of the intial data.

CDF from Rank Transformation

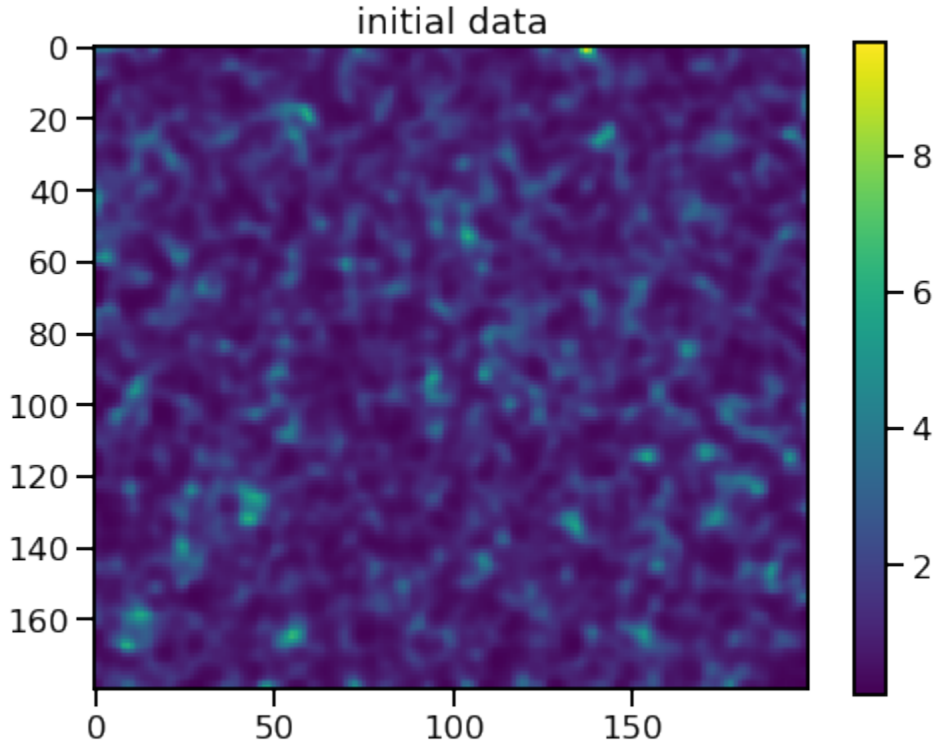
Okay, this is really easy - we just plot the one against the another.

[6]:
```
plt.plot( r_sm.flatten(), frank.flatten(), '.')
plt.plot( dset.flatten(), frank.flatten(), '.')

plt.xlabel( 'initial data' )
plt.ylabel( 'rank data')
sns.despine()
```

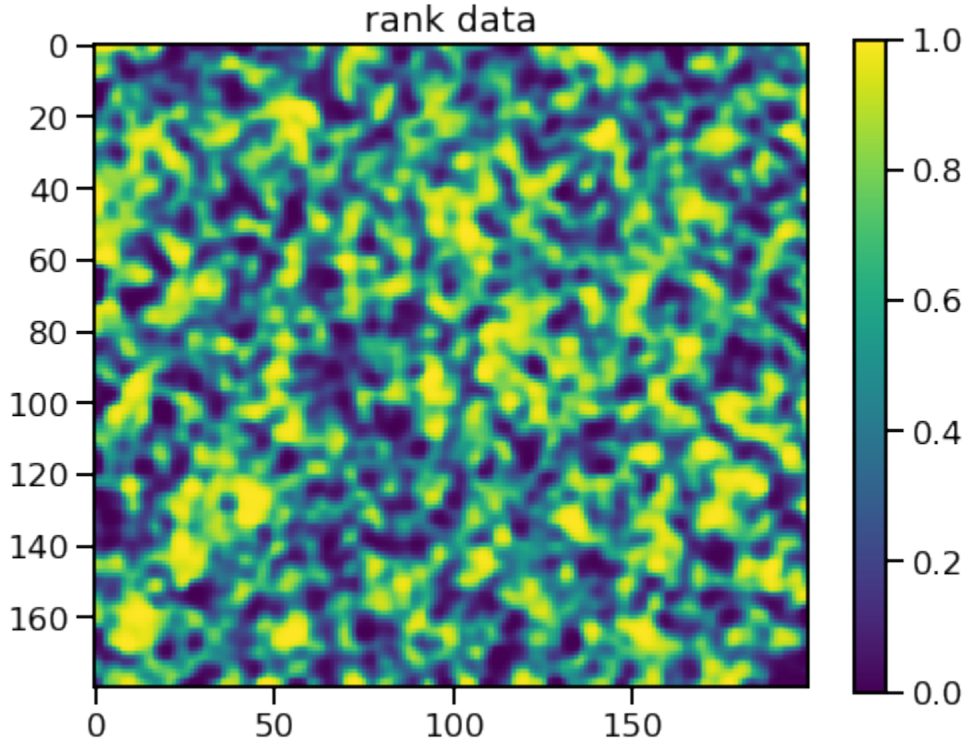


4.6.4 Rank Transformation to Stretch Color Range

[7]:
```
plt.imshow( dset )
plt.title( 'initial data')
plt.colorbar();
```



[8]:
```
plt.imshow( frank )
plt.title( 'rank data' )
plt.colorbar();
```



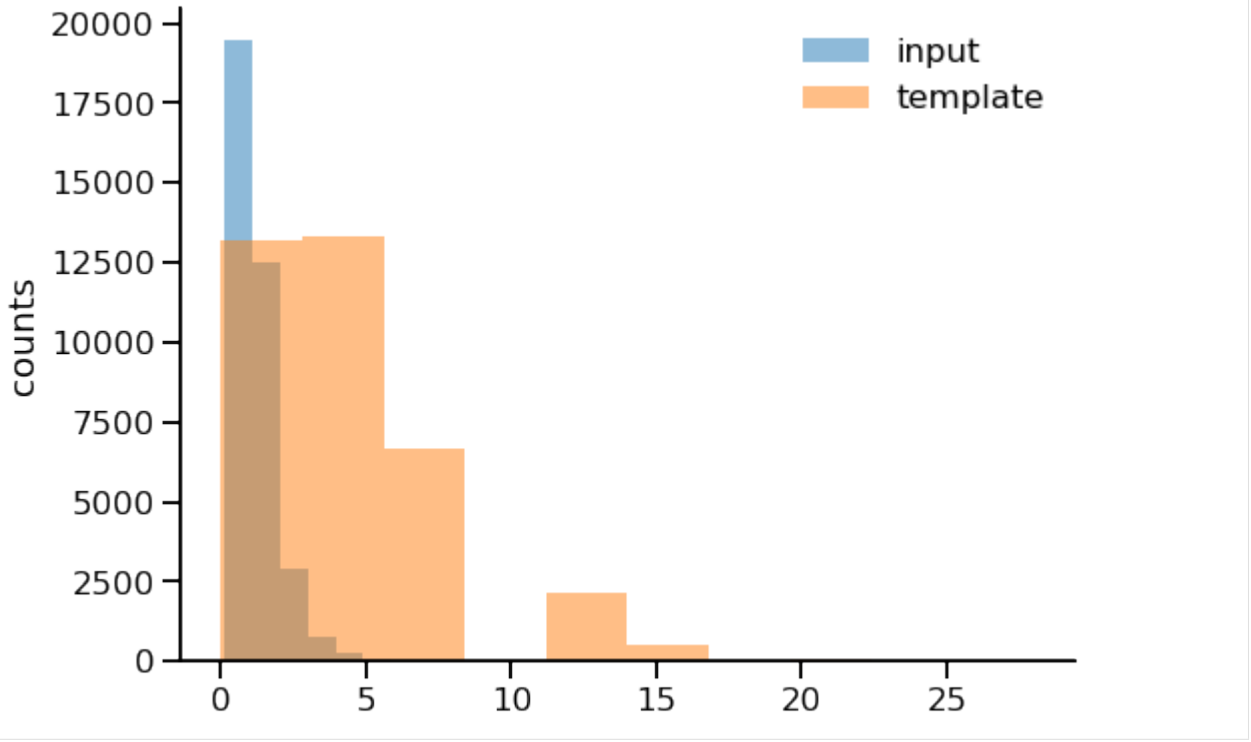
4.6.5 Histogram Matching

We apply histogram matching to transform input data such that

- order of the data remains
- final distribution matches the distribution of template data

Data with Different Distributions

[9]:
```
plt.hist( dset.flatten(), label = 'input', alpha = 0.5);
plt.hist( rpoi.flatten(), label = 'template', alpha = 0.5);

plt.ylabel( 'counts')
plt.legend(frameon = False)
sns.despine()
```

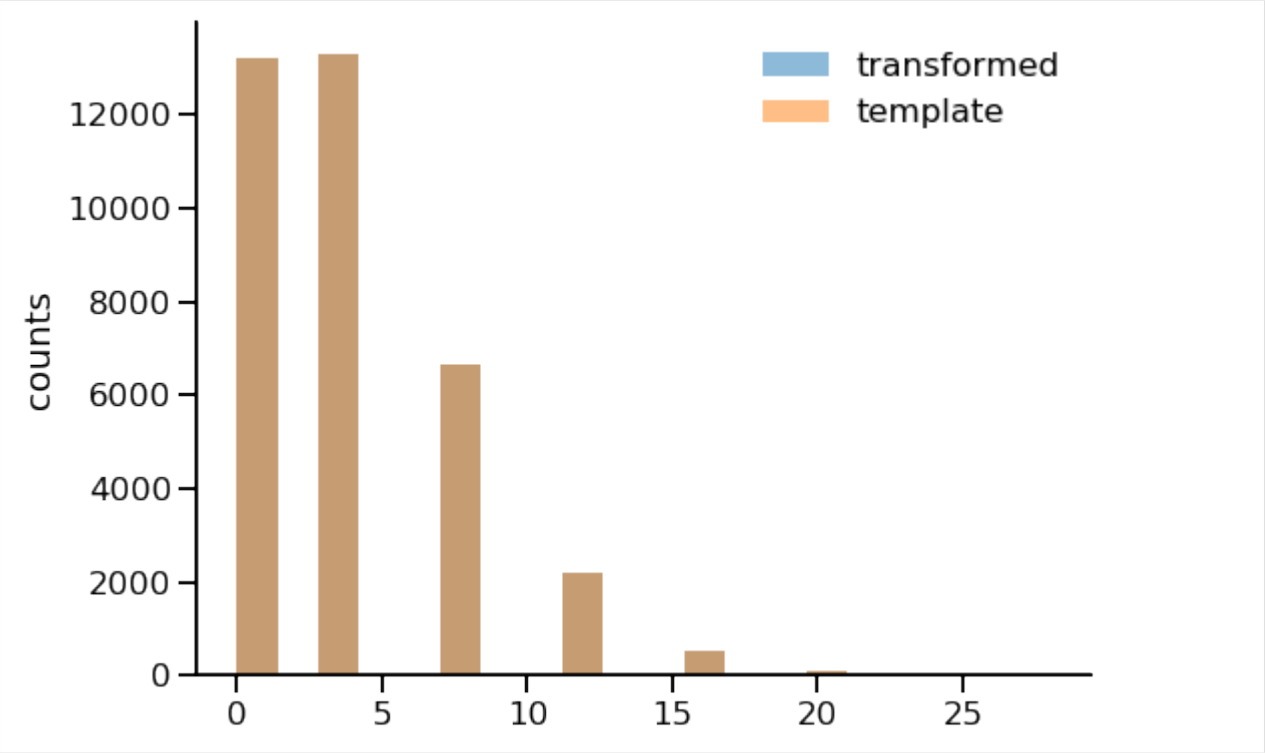


Now, we apply a non-inear transformation to the input dataset `dset`. As result, the transformed data `dset_t` have the same distribution as `rpoi`.

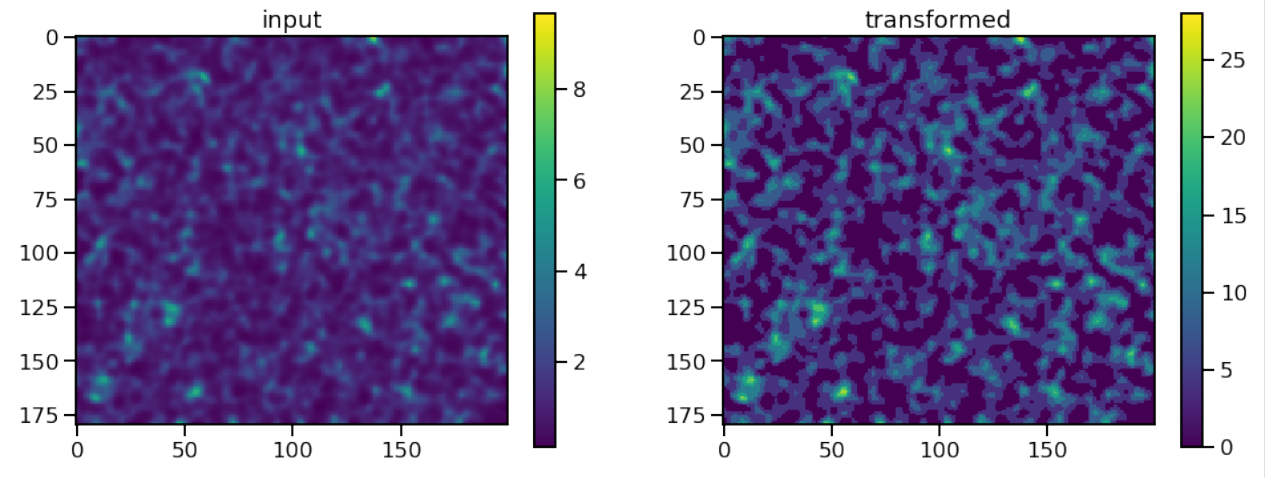
Applying the Matching

[10]:
```
dset_t = tropy.analysis_tools.statistics.fdistrib_mapping( dset, rpoi )
```

[11]:
```
plt.hist( dset_t.flatten(), 20, label = 'transformed', alpha = 0.5);
plt.hist( rpoi.flatten(),  20, label = 'template', alpha = 0.5);

plt.ylabel( 'counts')
plt.legend(frameon = False)
sns.despine()
```



[12]:
```
fig, ax = plt.subplots( ncols = 2, figsize = (16, 6) )

plt.sca (ax[0])
plt.imshow( dset )
plt.title( 'input' )
plt.colorbar()

plt.sca (ax[1])
plt.imshow( dset_t )
plt.title( 'transformed' )
plt.colorbar();
```



4.6.6 Summary

The module `tropy.analysis_tools.statistics` provides functions to

- apply rank transformations
- histogram matching / probability matching