Practical_ESA_Croatia

September 16, 2023

[1]:
```
import os
import numpy as np
import matplotlib.pyplot as plt
import spectral.io.envi as envi
```

[2]:
```
# set your path here for the files
os.chdir(os.path.expanduser('~es40/Documents/teaching/ESA Croatia/Practical2'))
```

[3]:
```
def Open_ENVI(filename):
    """
    This function opens Sentinel (1, 2, 3 etc.) images
    Requires the file in ENVI format
    The header can be read from the ENVI .hdr
    """
    # This opens the header file (which is a text file)
    lib = envi.open(filename + '.hdr')
    # here we read the size of the image
    header =  np.array([lib.ncols, lib.nrows])
    # here we read the type of the pixel values
    datatype = lib.dtype
    # Opening the image
    f = open(filename + '.img', 'rb')
    img = np.fromfile(f, dtype=datatype)
    # The image is initiually read as a 1D entity, the following makes it 2D
    img = img.reshape(header,order='F').astype(datatype)
    # returns the values of the image pixels as a 2D array
    return(img)

# This is another function that we will be using in the following
def generate_image_dict(dictionary, band_list, path_s3):
    '''
    We store each band (e.g. 'Oa01 radiance) as a value in a dictionary.
    Using a dictionary allows to create key, value pairs, where key is the␣
 ↪band_name, e.g. 'Oa01_radiance' and
    the value is the corresponding 2D image entity.
    We read each band in individually and store it in the dictionary.
    '''
```

```
    # looping over the defined band list - either for the subset or the ac␣
 ↪corrected image
    for band in band_list:
        # opening one image entity
        img = Open_ENVI(path_s3 + band)
        #storing the image entity in the dictionary, where [band] is the key␣
 ↪and img the value
        dictionary[band] = img
```

[4]:
```
#%%
# define the two paths for both files
# standard L1B subset
path_s3_subset =␣
 ↪'subset_2_of_S3A_OL_1_EFR____20230617T092432_20230617T092732_20230618T101954_0179_100_093_2
 ↪/'

# AC (Atmosphere corrected) corrected L2 subset
path_s3_ac = 'DOORS_S3_cluster_example//'
```

[23]:
```
# 1. We wish to inspect the true-color TOA radiance imagery of the subset to
# see if we can spot any obvious clouds or other artifacts in this image.
# We previously created a subset of the full tile focused on the Lithuanian
# lagoon next to the Baltic Sea.

# We create a list of the radiance bands in order for them to be available
# when generating the true-color image.
band_names_subset = ['Oa01_radiance', 'Oa02_radiance', 'Oa03_radiance',␣
 ↪'Oa04_radiance', 'Oa05_radiance','Oa06_radiance',
                     'Oa07_radiance', 'Oa08_radiance', 'Oa09_radiance',␣
 ↪'Oa10_radiance', 'Oa11_radiance','Oa12_radiance', 'Oa16_radiance',
                     'Oa17_radiance', 'Oa18_radiance', 'Oa21_radiance']

# We store each band (e.g. 'Oa01 radiance) as a value in a dictionary.
# Using a dictionary allows to create key, value pairs, where key is the
# band_name, e.g. 'Oa01_radiance' and the value is the corresponding 2D img
# entity we get while looping over the function previously defined to read
# Sentinel image data.
# We read each band in individually and store it in the dictionary

# initialisation of an empty dictionary that we fill with the key, value pairs
dict_subset_image={}

# calling the function with the new empty dict, the defined band_names of the␣
 ↪l1b product and the path of the l1b subset
generate_image_dict(dict_subset_image, band_names_subset, path_s3_subset)

# dictionaries are assessed by their key, that will show the value -
```

```
# in this case the first top-of-atmosphere radiance band
dict_subset_image['Oa01_radiance']

# the benefit of this procedure is that the different bands can be assessed
# unambigously. It allows for clear and reproducible code.

# The true-color composite requires RGB channels. Here we define the colors
# red, blue and green as given out from ESA in the standard L1B product.
# Either here or later when displayed we transpose all matrices here in order
# to display them correctly later. Simply call .T on the matrix to tranpose it.

red = (1.0 + 0.01 * dict_subset_image['Oa01_radiance'] + 0.09 *␣
 ↪dict_subset_image['Oa02_radiance']
         + 0.35 * dict_subset_image['Oa03_radiance'] + 0.04 *␣
 ↪dict_subset_image['Oa04_radiance']
         + 0.01 * dict_subset_image['Oa05_radiance'] + 0.59 *␣
 ↪dict_subset_image['Oa06_radiance']
         + 0.85 * dict_subset_image['Oa07_radiance'] + 0.12 *␣
 ↪dict_subset_image['Oa08_radiance']
         + 0.07 * dict_subset_image['Oa09_radiance'] + 0.04 *␣
 ↪dict_subset_image['Oa10_radiance']).T

green = (1.0 + 0.26 * dict_subset_image['Oa03_radiance'] + 0.21 *␣
 ↪dict_subset_image['Oa04_radiance']
       + 0.50 * dict_subset_image['Oa05_radiance'] +␣
 ↪dict_subset_image['Oa06_radiance']
       + 0.38 * dict_subset_image['Oa07_radiance'] + 0.04 *␣
 ↪dict_subset_image['Oa08_radiance']
       + 0.03 * dict_subset_image['Oa09_radiance']
       + 0.02 * dict_subset_image['Oa10_radiance']).T

blue = (1.0 + 0.07 * dict_subset_image['Oa01_radiance'] + 0.28 *␣
 ↪dict_subset_image['Oa02_radiance']
             + 1.77 * dict_subset_image['Oa03_radiance']
             + 0.47 * dict_subset_image['Oa04_radiance']
             + 0.16 * dict_subset_image['Oa05_radiance']).T

# the shape/size of every band in the dictionary, here e.g. Oa01 radiance is
# 417 , 417 - making it a 2D array we work with throughout the practical.
size = np.shape(dict_subset_image['Oa01_radiance'])
print(size)
```

```
(2017, 1697)
```

[24]:
```
# We generate a 3D-array here with zeros that we fill with the RGB values
# to generate the true-color composite -
```

```
# A composite is essentially a 3D stack of three different bands - red, green,␣
 ↪blue.

# Empty 3D-array
rgb_composite = np.zeros([size[0],size[1],3])
# see shape:
rgb_composite.shape

# first dimension is red (indexed with 0)
rgb_composite[:,:,0] = red/(red.mean()*2.2)
# second is green
rgb_composite[:,:,1] = green/(green.mean()*2.2)␣
 ↪##################################
# third blue
rgb_composite[:,:,2] = blue/(blue.mean()*2.2) ##############################

# The stacked RGB bands are now available to plot to generate the true-color␣
 ↪image:

fig, (ax1) = plt.subplots(1, sharex=True, sharey=True,figsize=(11, 9))
ax1.set_title('RGB L1B true-color composite of the selected Sentinel-3 OLCI␣
 ↪scene over the Lithuanian lagoon', fontsize= 14, y=1.02)
ax1.imshow(rgb_composite)
plt.tight_layout()
plt.show()
# note the x and y axis numbers! Speficially note down where 250, 250
# is positioned. We need this coordinate later.
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[24], line 11
      8 rgb_composite.shape
     10 # first dimension is red (indexed with 0)
---> 11 rgb_composite[:,:,0] = red.T/(red.mean.T()*2.2)
     12 # second is green
     13 rgb_composite[:,:,1] = green/(green.mean()*2.2)␣
 ↪##################################

AttributeError: 'builtin_function_or_method' object has no attribute 'T'
```

[9]:
```
# 2. We load the bands necessary here for water quality calculations from the
# same, but atmospherically corrected image - this is a new file and has been
# produced externally
# Remote-sensing reflectances from top-of-atmosphere radiance were calculated
# using the C2RCC Neural Network processor:
# http://step.esa.int/docs/extra/Evolution%20of%20the%20C2RCC_LPS16.pdf
```

```
# Read this document if you want to understand more and also how the
# uncertainty bands are calculated we are using up next.

# Another array with the band names we need - with remote-sensing reflectances␣
 ↪this time from the atmospherically corrected image.
band_names_l2 = ['rrs_1', 'rrs_2', 'rrs_3', 'rrs_4', 'rrs_5','rrs_6',
              'rrs_7', 'rrs_8', 'rrs_9', 'rrs_10', 'rrs_11','rrs_12', 'rrs_16',
             'rrs_17', 'rrs_18', 'rrs_21',
              # we also load the so-called 'out-of-scope (oos)' quality flags
              # to inspect the uncertainty of calculated Rrs - by products of␣
 ↪C2RCC AC
             'oos_rtosa','oos_rrs',
              # and iop_apig, which is the absorption of phytoplankton pigments
              # and it's corresponding uncertainty flag
              'iop_apig', 'unc_apig'
             ]

# the function needs a new dict, as the other one is for the L1 subset only
dict_image={}

# calling the function with the new empty dict, the defined band_names of the
# L2 AC product and the new path of the L2 AC image
generate_image_dict(dict_image, band_names_l2, path_s3_ac)

# inspect if correctly loaded
dict_image['rrs_1']
# first value should be XXXXXXX
```

[9]:
```
array([[0.        , 0.        , 0.        , …, 0.01045474, 0.01045103,
        0.01108371],
       [0.        , 0.        , 0.        , …, 0.01062626, 0.01035504,
        0.01075062],
       [0.        , 0.        , 0.        , …, 0.01062626, 0.01035872,
        0.01075062],
       …,
       [0.        , 0.        , 0.        , …, 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , …, 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , …, 0.        , 0.        ,
        0.        ]], dtype=float32)
```

[ ]:
```
#%%
# 3.
# The image corresponding to the values in the 'oos_rtosa' band - a product
# from the C2RCC atmospheric correction showing the top-of-atmosphere radiances.
# This AC algorithm is based on the L1 radiance we previously used to generate
```

```
# the true-color image. The band 'oos_rtosa' shows if the TOA radiances are
# out of scope (see algorithm document linked previously).
# if Rtosa calculation is already indicating high uncertainty, Rrs values will
# most likely be unrealistic, too, as they are derived from Rtosa values.

#create a figure
plt.figure(2)
# We assess the band from the dictionary directly, and transpose the matrix
# of the 2D-image entity to correctly display it (.T method)
plt.imshow(dict_image['oos_rtosa'].T)
# colorbar
cbar = plt.colorbar(pad=0.01)
cbar.set_label('Uncertainty')

plt.axis("off")
plt.title('Uncertainty for top-of-atmosphere radiances', fontsize= 14, y=1.02)
plt.show()

# Looks very good!

# Let's show the remote-sensing reflectances out-of-scope band from the C2RCC
# product and see if we find any area in the water that has high values.
# high values = the calculation of Rrs values is uncertain. Over land we expect
# high values, but we require low values for water, otherwise the water quality
# parameter calculation will fail.
# If Rrs are highly uncertain, they are most likely wrong as the (or any)
# atmospheric correction algorithm is limited in it's application range.

#create a figure
plt.figure(3)
#plot the image corresponding to the values in the oos_rrs band - a product
# from the C2RCC atmospheric correction algorithm.
plt.imshow(dict_image['oos_rrs'].T)    ␣
 ↪##########################################
# colorbar
cbar = plt.colorbar(pad=0.01)
cbar.set_label('Uncertainty')

plt.axis("off")
plt.title('Uncertainty for remote-sensing reflectance calculation', fontsize=␣
 ↪14, y=1.02)
plt.show()


# The water areas are mostly showing low uncertainties, some areas are less
# dark though and these areas need to be treated with more care for the
# reliability of the results - espcially the center part of the lagoon.
```

```
# Let's inspect the reflectances, to display them.

# Sentinel-3 OLCI wavelengths
# We need them to select the right band numbers from the Sentinel L2 product we␣
 ↪already inspected.


# rrs_1 = 400 nm
# rrs_2 = 412.5 nm
# rrs_3 = 442.5 nm
# rrs_4 = 490 nm
# rrs_5 = 510 nm
# rrs_6 = 560 nm
# rrs_7 = 620 nm
# rrs_8 = 665 nm
# rrs_9 = 673.75 nm
# rrs_10 = 681.25 nm
# rrs_11 = 708.75 nm
# rrs_12 = 753.75 nm
# rrs_16 = 778.75 nm
# rrs_17 = 865 nm
# rrs_21 = 1020 nm

# First a single reflectance
# We select one single pixel for all wavelengths (i.e. one position in the 2D
# 417, 417 array) - corresponding to x and y position 250 we looked at in the
# true-color composite.

rrs_400 = dict_image['rrs_1'][600, 600]
rrs_412 = dict_image['rrs_2'][600, 600]
rrs_442 = dict_image['rrs_3'][600, 600]     ###################################
rrs_490 = dict_image['rrs_4'][600, 600]
rrs_510 = dict_image['rrs_5'][600, 600]
rrs_560 = dict_image['rrs_6'][600, 600]
rrs_620 = dict_image['rrs_7'][600, 600]
rrs_665 = dict_image['rrs_8'][600, 600]
rrs_673 = dict_image['rrs_9'][600, 600]
rrs_681 = dict_image['rrs_10'][600, 600]
rrs_708 = dict_image['rrs_11'][600, 600]
rrs_753 = dict_image['rrs_12'][600, 600]
rrs_778 = dict_image['rrs_16'][600, 600]
rrs_865 = dict_image['rrs_17'][600, 600]
rrs_1010 = dict_image['rrs_21'][600, 600]

# we combine all the values into one list:
```

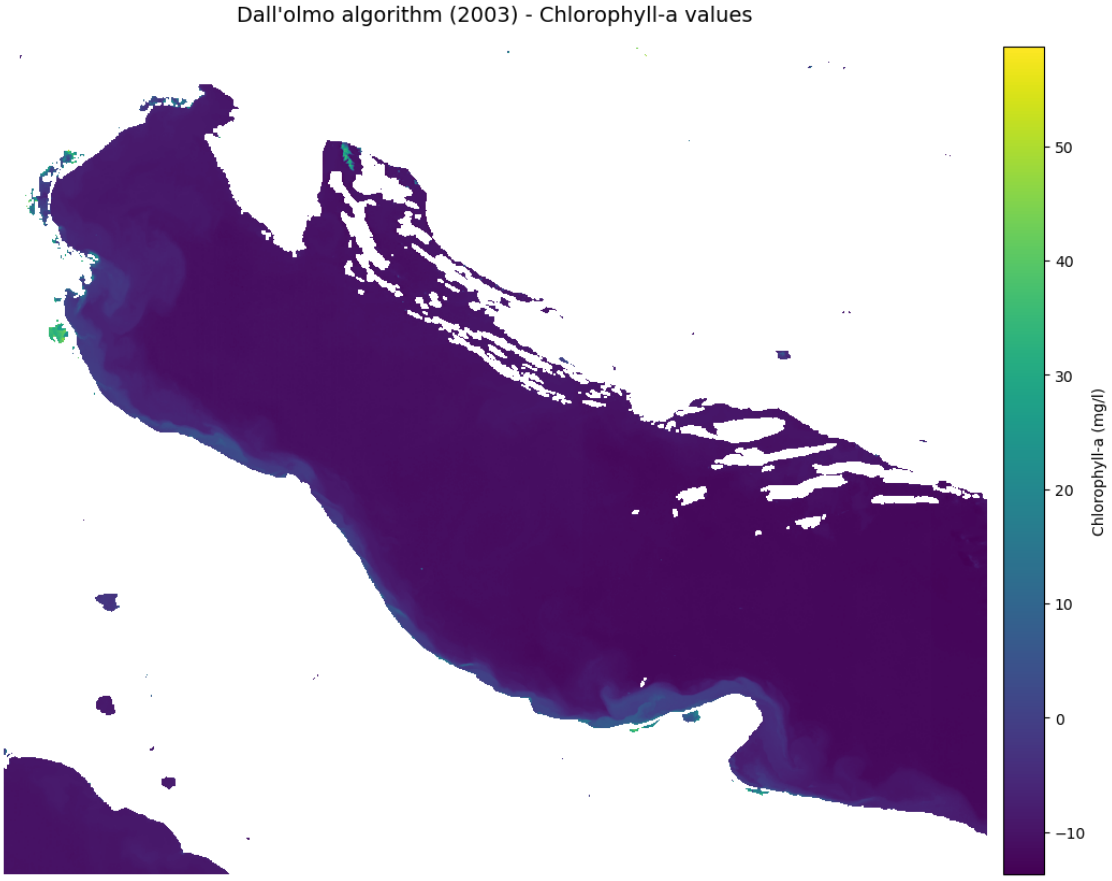
```
rrs = [rrs_400, rrs_412, rrs_442, rrs_490, rrs_510, rrs_560, rrs_620, rrs_665,␣
 ↪rrs_673, rrs_681, rrs_708, rrs_753, rrs_778, rrs_865, rrs_1010]
print(rrs)

# i.e. 15 values for 15 different OLCI wavelengths we are going to define next:

wavelengths_olci = np.array([400, 412, 442, 490, 510, 560, 620, 665, 673, 681,␣
 ↪708, 753, 778, 865, 1010])

# now we can display one Rrs:

plt.figure(4)
plt.plot(wavelengths_olci, rrs)
plt.ylabel(r'Rrs (sr$^{-1}$)', fontsize=14)
plt.xlabel('Wavelength (nm)', fontsize=14)
plt.title('Remote-sensing reflectances of coastal waters', y=1.02, fontsize=14)
plt.xlim(xmin=400, xmax=1010)
plt.ylim(ymin=0)
plt.show()

# What does this Rrs signature tell you?

# Let's do it now for 40 water pixels, selecting a slice of the arrays and
# assessing the first dimension [0]

rrs_400 = dict_image['rrs_1'][600:700, 700:800][0]
rrs_412 = dict_image['rrs_2'][600:700, 700:800][0]
rrs_442 = dict_image['rrs_3'][600:700, 700:800][0]
rrs_490 = dict_image['rrs_4'][600:700, 700:800][0]
rrs_510 = dict_image['rrs_5'][600:700, 700:800][0]
rrs_560 = dict_image['rrs_6'][600:700, 700:800][0]
rrs_620 = dict_image['rrs_7'][600:700, 700:800][0]
rrs_665 = dict_image['rrs_8'][600:700, 700:800][0]
rrs_673 = dict_image['rrs_9'][600:700, 700:800][0]
rrs_681 = dict_image['rrs_10'][600:700, 700:800][0]
rrs_708 = dict_image['rrs_11'][600:700, 700:800][0]
rrs_753 = dict_image['rrs_12'][600:700, 700:800][0]
rrs_778 = dict_image['rrs_16'][600:700, 700:800][0]
rrs_865 = dict_image['rrs_17'][600:700, 700:800][0]
rrs_1010 = dict_image['rrs_21'][600:700, 700:800][0]

# new array with all 40 rrs:
rrs_40 = [rrs_400, rrs_412, rrs_442, rrs_490, rrs_510, rrs_560, rrs_620,␣
 ↪rrs_665, rrs_673, rrs_681, rrs_708, rrs_753, rrs_778, rrs_865, rrs_1010]

plt.figure(5)
```

```
plt.plot(wavelengths_olci, rrs_40)     ␣
 ↪##########################################
plt.ylabel(r'Rrs (sr$^{-1}$)', fontsize=14)
plt.xlabel('Wavelength (nm)', fontsize=14)
plt.title('Remote-sensing reflectances of coastal waters', y=1.02, fontsize=14)
plt.ylim(ymin=0)
plt.xlim(xmin=400, xmax=1010)
plt.show()

# In comparison to a larger area... which signatures do not correspond to
# water reflectances?

rrs_400 = dict_image['rrs_1'][400:600][0]
rrs_412 = dict_image['rrs_2'][400:600][0]
rrs_442 = dict_image['rrs_3'][400:600][0]
rrs_490 = dict_image['rrs_4'][400:600][0]
rrs_510 = dict_image['rrs_5'][400:600][0]
rrs_560 = dict_image['rrs_6'][400:600][0]
rrs_620 = dict_image['rrs_7'][400:600][0]
rrs_665 = dict_image['rrs_8'][400:600][0]
rrs_673 = dict_image['rrs_9'][400:600][0]
rrs_681 = dict_image['rrs_10'][400:600][0]
rrs_708 = dict_image['rrs_11'][400:600][0]
rrs_753 = dict_image['rrs_12'][400:600][0]
rrs_778 = dict_image['rrs_16'][400:600][0]
rrs_865 = dict_image['rrs_17'][400:600][0]
rrs_1010 = dict_image['rrs_21'][400:600][0]

rrs_mixed = [rrs_400, rrs_412, rrs_442, rrs_490, rrs_510, rrs_560, rrs_620,␣
 ↪rrs_665, rrs_673, rrs_681, rrs_708, rrs_753, rrs_778, rrs_865, rrs_1010]

plt.figure(6)
plt.plot(wavelengths_olci, rrs_mixed)  ######################################
plt.ylabel(r'Rrs (sr$^{-1}$)', fontsize=14)
plt.xlabel('Wavelength (nm)', fontsize=14)
plt.ylim(ymin=0)
plt.title('Remote-sensing reflectances of the water and surroundings', y=1.02,␣
 ↪fontsize=14)
plt.xlim(xmin=400, xmax=1010)
plt.show()
```

[10]:
```
#%%
# Rrs for looks promising overall, let's calculate some water quality
# parameters from it.
# We are going to apply 4 different algorithms, covered in the lecture.

# First one: Dall'olmo chlorophyll-a algorithm - 2003
```

```
# a straight forward simple band ratio algorithm that has two empirical extra
# terms (a, b)

# assign variables, makes the reading of the equation below more convenient.
# Of course, one could also use the dictionary rrs bands directly.
rrs_665 = dict_image['rrs_8']
rrs_708 = dict_image['rrs_11']

# empirical coefficients
a = 61.324
b = -37.94

# equation to calculate chla:
chla_dallolmo = a * (rrs_708/rrs_665) + b

# add the calculated chla values as a band to the dictionary of the image
dict_image['dallolmo_chla'] = chla_dallolmo

# Display of Dall'olmo chl-a values
plt.figure(figsize=(20,10))
plt.imshow(dict_image['dallolmo_chla'].T)
plt.title("Dall'olmo algorithm (2003) - Chlorophyll-a values", fontsize = 14, y␣
  ↪= 1.02)
cbar = plt.colorbar(pad=0.01)
cbar.set_label('Chlorophyll-a (mg/l)')
plt.axis("off")
plt.show()
```

```
/var/folders/m8/96r4lc7d7xg7yqs0ccbsx3n00000gp/T/ipykernel_79033/260444608.py:21
: RuntimeWarning: invalid value encountered in divide
  chla_dallolmo = a * (rrs_708/rrs_665) + b
```



[ ]:
```
#%%
# Rrs for looks promising overall, let's calculate some water quality
# parameters from it.
# We are going to apply 4 different algorithms, covered in the lecture.

# First one: Dall'olmo chlorophyll-a algorithm - 2003

# a straight forward simple band ratio algorithm that has two empirical extra
# terms (a, b)

# assign variables, makes the reading of the equation below more convenient.
# Of course, one could also use the dictionary rrs bands directly.
rrs_665 = dict_image['rrs_8']
rrs_708 = dict_image['rrs_11']

# empirical coefficients
a = 61.324
b = -37.94
```

```
# equation to calculate chla:
chla_dallolmo = a * (rrs_708/rrs_665) + b

# add the calculated chla values as a band to the dictionary of the image
dict_image['dallolmo_chla'] = chla_dallolmo

# Display of Dall'olmo chl-a values
plt.figure(figsize=(20,10))
plt.imshow(dict_image['dallolmo_chla'].T)
plt.title("Dall'olmo algorithm (2003) - Chlorophyll-a values", fontsize = 14, y␣
↪= 1.02)
cbar = plt.colorbar(pad=0.01)
cbar.set_label('Chlorophyll-a (mg/l)')
plt.axis("off")
plt.show()
```

[ ]:
```
# OC4 algorithm

# The NASA OC4 algorithm is basically a polynomial with 4 terms.
# the x coefficient is dynamically calculated, based on the highest values of
# rrs at 443nm, 490nm and 510nm.

# We calculate x per pixel and insert it into the following OC4 equation we
# use below:

# oc4_chloropyhll-a = 10**((a+b * x) + (c*x**2) + (d*x**3) + (e*x**4))

# First we select the necessary bands from the dictionary:
rrs_443 = dict_image['rrs_3']
rrs_490 = dict_image['rrs_4']
rrs_510 = dict_image['rrs_5']
rrs_560 = dict_image['rrs_6']

# static coefficients for the OC4 algorithm:
a = 0.3255
b = -2.7677
c = 2.4409
d = -1.1288
e = -0.4990

# each band from our image is a 2D array and to compare the values of the 3
# rrs bands directly for the coefficient 'x' per pixel, we need to iterate
# over the values directly.

print(rrs_510)
print(rrs_510[0][0])
# >>> 0.00470531 is the corresponding value. This is the value of Rrs at 510nm
```

```
# at the first pixel.
# try e.g. to print(rrs_510[0][0]) with different values for 0 (e.g. 1, or 2)
# if you have issues understanding this nested for-loop.

# we assess the values of each band through iteration of the indices, i for
# the outer array (1D), j for the inner (2D)

# it does not matter over which band we iterate for these indices, as they all
# have the same dimensions (417, 417), and hence same indices.
# try rrs_560 for the two for-loop lines below instead of rrs_510 if you want
# to convince yourself.

# This procedure creates an array with the same shape and type as any rrs_band
# (e.g. rrs_510 in this example) 2D array that we fill with new chl-a values
# later on, otherwise the new chl-a pixels could not be displayed.

# empty array with 2D dimensions
chla_oc4_array = np.full_like(rrs_510,0)

# we assess the values of each band through iteration of the indices, i for
# the outer array (1D), j for the inner (2D)
for i in range(len(rrs_510)):
        for j in range(len(rrs_510[i])):

                # the OC4 algorithm requires to use the largest value of rrs
                # at 443nm, 490nm or 510nm that is then divided by rrs at 560nm
                # so we compare each band here and take the log10 of this value␣
 ↪(x)

                # 1. if rrs_510 > 443 and 490:
                 if (rrs_510[i][j] > rrs_443[i][j]) & (rrs_510[i][j] >␣
 ↪rrs_490[i][j]):
                        x = np.log10(rrs_510[i][j] / rrs_560[i][j])

                # 2. if rrs_443 > 490 and 510:
                 elif (rrs_443[i][j] > rrs_490[i][j]) & (rrs_443[i][j] >␣
 ↪rrs_510[i][j]):
                        x = np.log10(rrs_443[i][j] / rrs_560[i][j])

                # 3. if rrs_490 > 510, and rrs_443
                 elif (rrs_490[i][j] > rrs_443[i][j]) & (rrs_490[i][j] >␣
 ↪rrs_510[i][j]):
                        x = np.log10(rrs_490[i][j] / rrs_560[i][j])

                # OC4 algorithm is defined as below and per pixel changes of
                # the value of x influences the final chl-a value (based on
                # which of the 3 Rrs is higher)
```

```
                # the coefficients (a, b, c, d, e) were defined before already

                chla_per_pixel = 10**((a+b * x) + (c*x**2) + (d*x**3) +␣
 ↪(e*x**4))

                # we then fill our empty 2D array with each new chl-a value:
                chla_oc4_array[i][j] = chla_per_pixel


# the array with the OC4 chl-a values
print(chla_oc4_array)

# Display of OC4 chl-a values
plt.figure(7)
plt.imshow(chla_oc4_array.T)
plt.title("OC4 chloropyhll-a", fontsize = 14, y = 1.02)
cbar = plt.colorbar(pad=0.01)
cbar.set_label('Chlorophyll-a (mg/l)')
plt.axis("off")
plt.show()


# compare the chl-a values to the Dall'olmo values.
# Can you think of general explanations why the values are lower?
# Which water types has the OC4 algorithm been designed for and how applicable
# is the algorithm to this area?
```

[ ]:
```
# TSM after Nechad et al., 2010
# In order to assess turbidity, one can calculate total suspended matter
# concentrations (TSM)

# One band algorithm to calculate TSM
rrs_665 = dict_image['rrs_8']

# Equation:
tsm = 9418.39 * (rrs_665 / (1 - rrs_665/17.28)) + 1.41
dict_image['TSM_nechad'] = tsm

# Display of TSM values

plt.figure(8)
plt.imshow(dict_image['TSM_nechad'].T)
plt.title("Total suspended matter (TSM) concentrations - Nechad et al., 2010␣
 ↪algorithm", fontsize = 14)
# bring the color bar closer to the
cbar = plt.colorbar(pad=0.01)
#latex syntax to plot correct labels
```

```
cbar.set_label(r'TSM (gm$^{-3}$)')
plt.axis("off")
plt.show()
```

[ ]: