# fogpy Documentation

*Release 1.2.0*

**Thomas Leppelt**

**May 12, 2020**

# Contents

This package provide algorithmns and methods for satellite based detection and nowcasting of fog and low stratus clouds (FLS).

Related FogPy Version: 1.1.3

It utilizes several functionalities from the pytroll project for weather satellite data processing in Python. The remote sensing algorithmns are currently implemented for the geostationary Meteosat Second Generation (MSG) satellites. But it is designed to be easly extendable to support other meteorological satellites in future.

Contents:

# CHAPTER 1

# Installation instructions

## 1.1 Getting the files and installing them

First you need to get the files from github:

```
cd /path/to/my/source/directory/
git clone https://github.com/m4sth0/fogpy
```

You can also retreive a tarball from there if you prefer, then run:

```
tar zxvf tarball.tar.gz
```

Then you need to install fogpy on you computer:

```
cd fogpy
python setup.py install [--prefix=/my/custom/installation/directory]
```

You can also install it in develop mode to make it easier to hack:

```
python setup.py develop [--prefix=/my/custom/installation/directory]
```

CHAPTER 2

# Fogpy usage in a nutshell

The package uses OOP extensively, to allow higher level metaobject handling.

For this tutorial, we will use a MSG scene for creating different fog products.

## 2.1 Import satellite data first

We start with the PyTroll package *satpy*. This package provide all functionalities to import and calibrate a MSG scene from HRIT files. Therefore you should make sure that mpop is properly configured and all environment variables like *PPP_CONFIG_DIR* are set and the HRIT files are in the given search path. For more guidance look up in the **'satpy'_** documentation

**Note:** Make sure *satpy* is correctly configured!

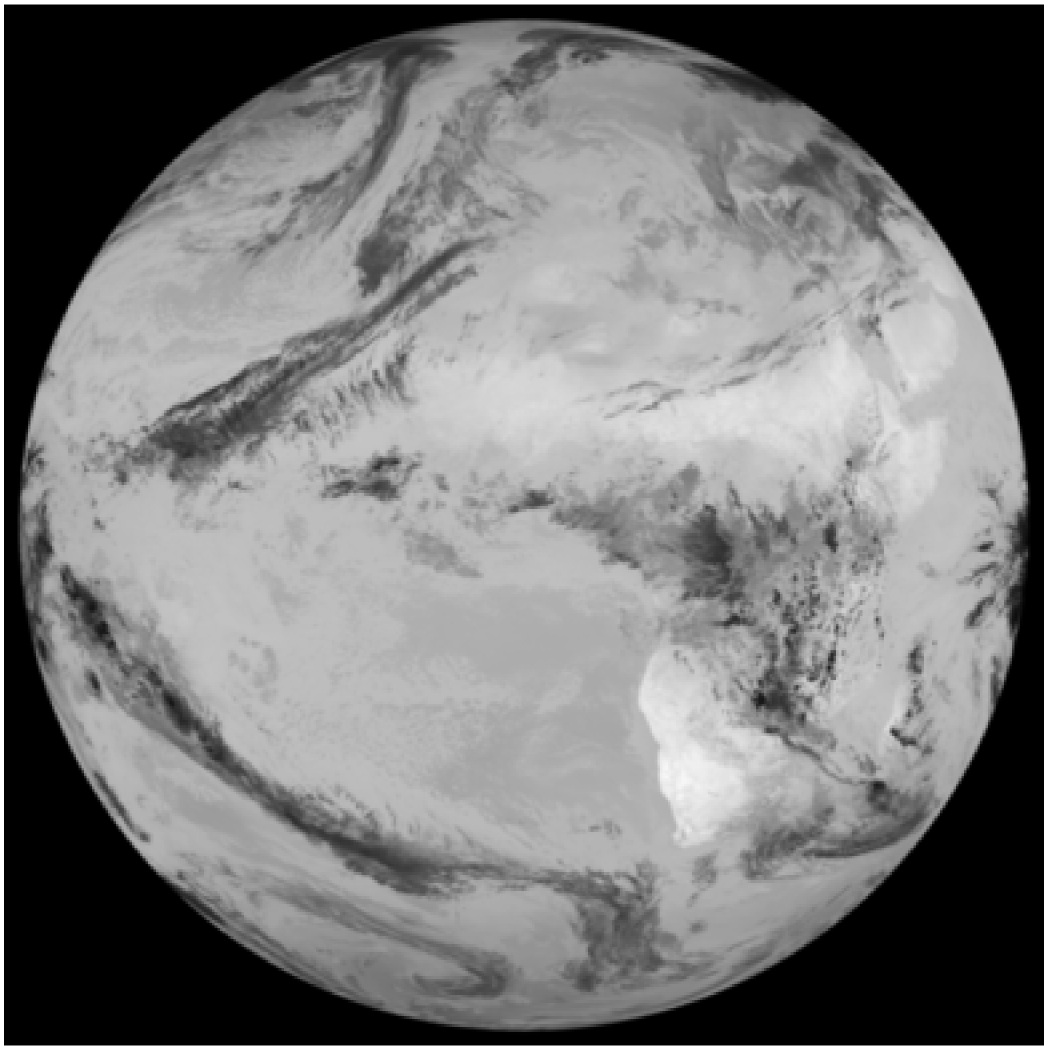
Ok, let's get it on:

```
>>> from satpy import Scene
>>> from glob import glob
>>> filenames = glob("/path/to/seviri/H-000*20131212000*")
>>> msg_scene = Scene(reader="seviri_l1b_hrit", filenames=filenames)
>>> msg_scene.load([10.8])
>>> msg_scene.load(["fog"])
```

We imported a MSG scene from 12. December 2013 and loaded the 10.8 μm channel and the built-in simple fog composite into the scene object.

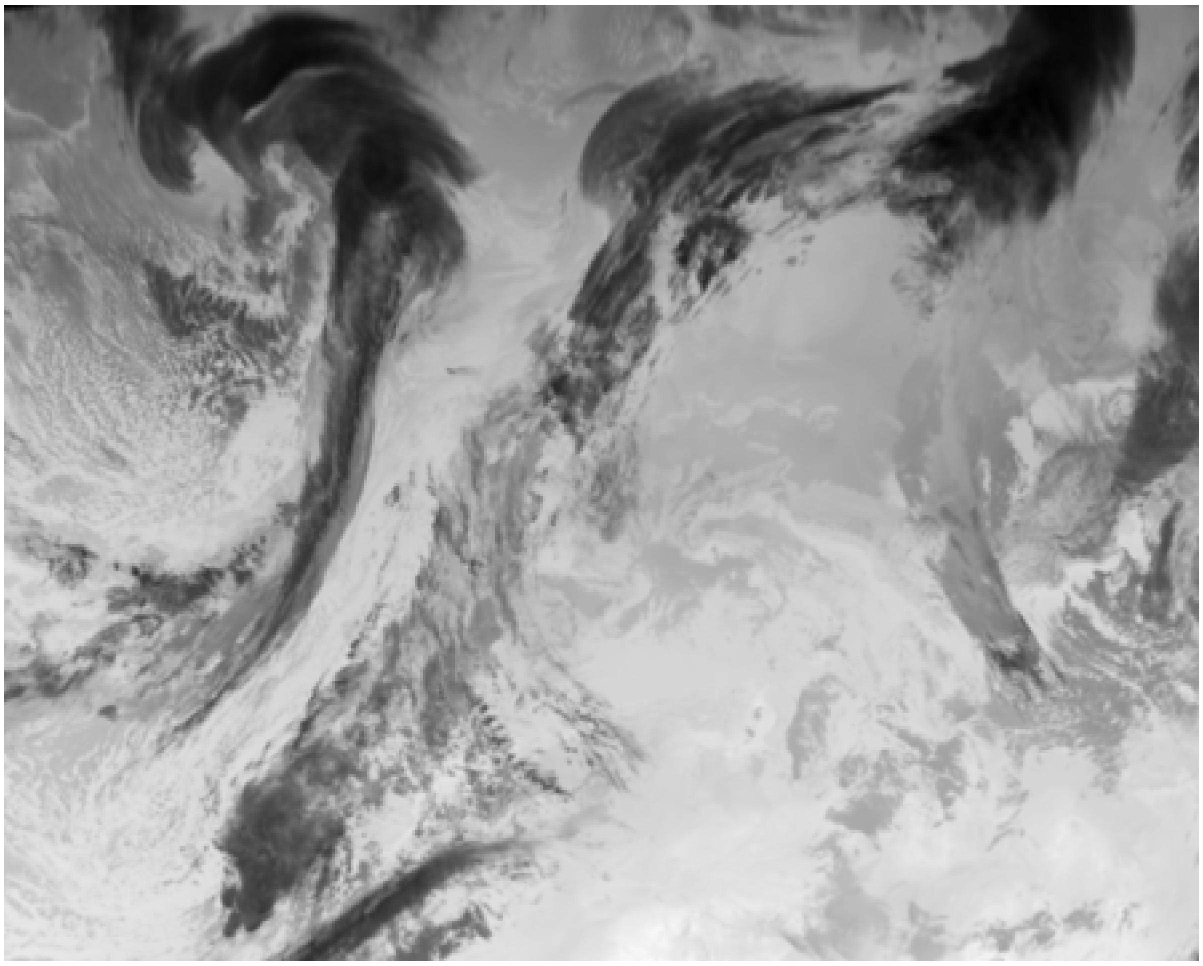
Now we want to look at the IR 10.8 channel:

```
>>> msg_scene.show(10.8)
```



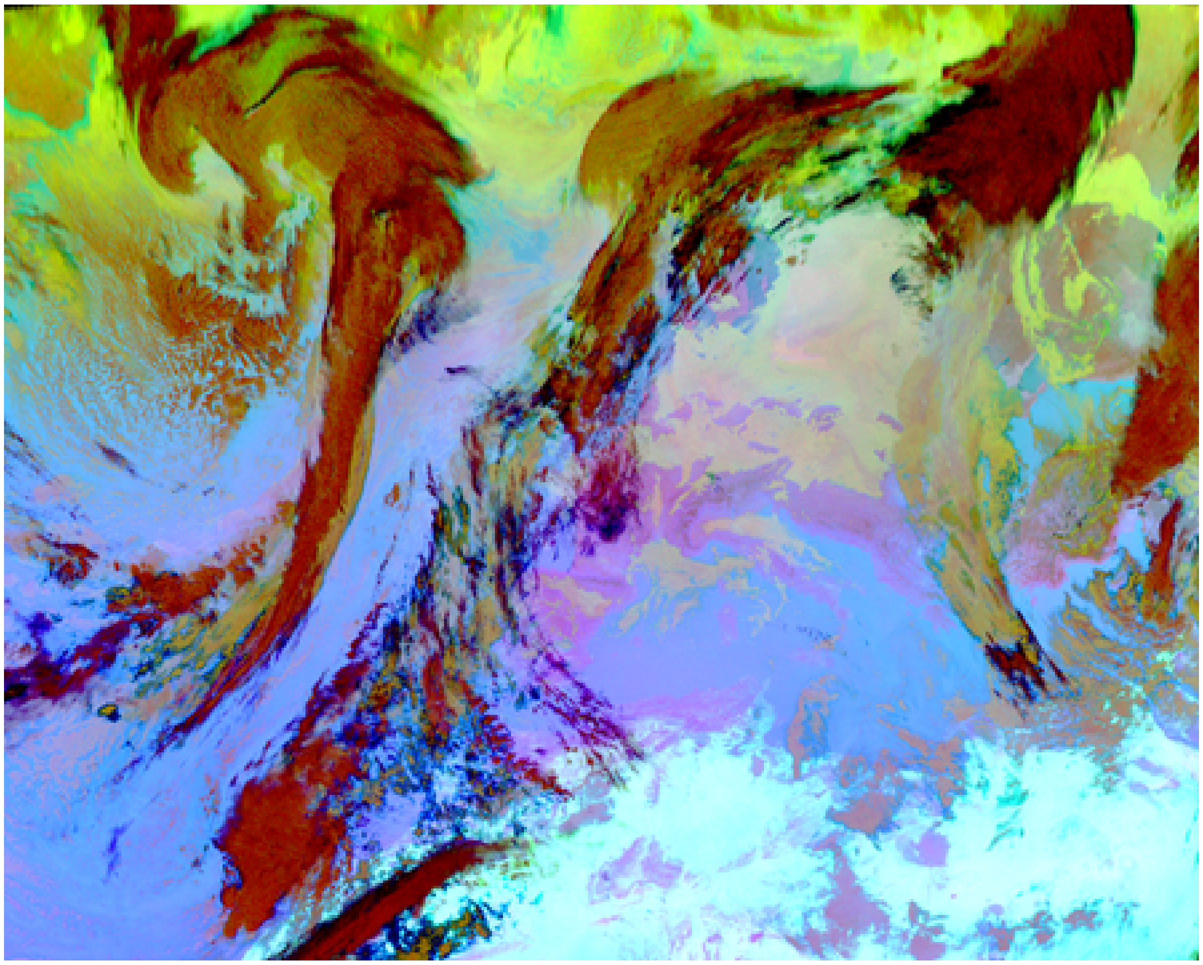
Everything seems correctly imported. We see a full disk image. So lets see if we can resample it to a central European region:

```
>>> eu_scene = msg_scene.resample("eurol")
>>> eu_scene.show(10.8)
```



A lot of clouds are present over central Europe. Let’s test a fog RGB composite to find some low clouds:

```
>>> eu_scene.show("fog")
```



The reddish and dark colored clouds represent cold and high altitude clouds, whereas the yellow-greenish color over central and eastern Europe is an indication for low clouds and fog.

## 2.2 Continue with more metadata

In the next step we want to create a fog and low stratus (FLS) composite for the imported scene. For this we need:

- Seviri L1B data, read by Satpy with the `seviri_l1b_hrit` reader.
- Cloud microphysical data, read by Satpy with the `nwcsaf-geo` reader. In principle, CMSAF data could also be used, but as of May 2019, there is no CM-SAF reader within Satpy.
- A digital elevation model. This can derived from data available from the European Environmental Agency (EEA). Although this can be read by Satpy using the `generic_image` reader, the Fogpy composite reads this as a static image. The path to this image needs to be defined in the Fogpy `etc/composites/seviri.yaml` file.

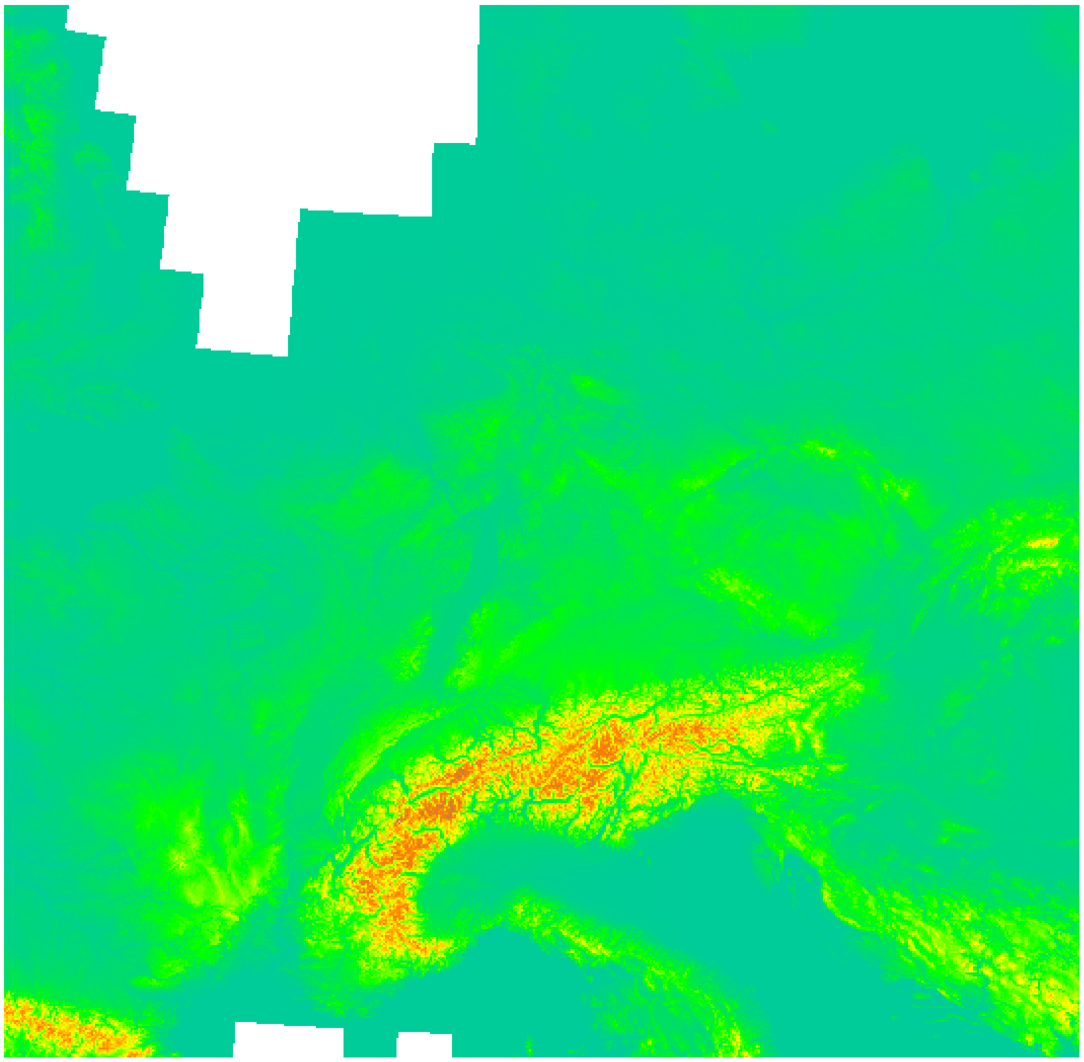
We create a scene in which we load datasets using both the `seviri_l1b_hrit` and `nwcsaf-geo` readers. Here we choose to load all required channels and datasets explicitly:

```
>>> fn_nwcsaf = glob("/media/nas/x21308/scratch/NWCSAF/*100000Z.nc")
>>> fn_sev = glob("/media/nas/x21308/scratch/SEVIRI/*201904151000*")
>>> sc = Scene(filenames={"seviri_l1b_hrit": fn_sev, "nwcsaf-geo": fn_nwcsaf})
```

(continues on next page)

(continued from previous page)

```
>>> sc.load(["cmic_reff", "IR_108", "IR_087", "cmic_cot", "IR_016", "VIS006",
              "IR_120", "VIS008", "cmic_lwp", "IR_039"])
```

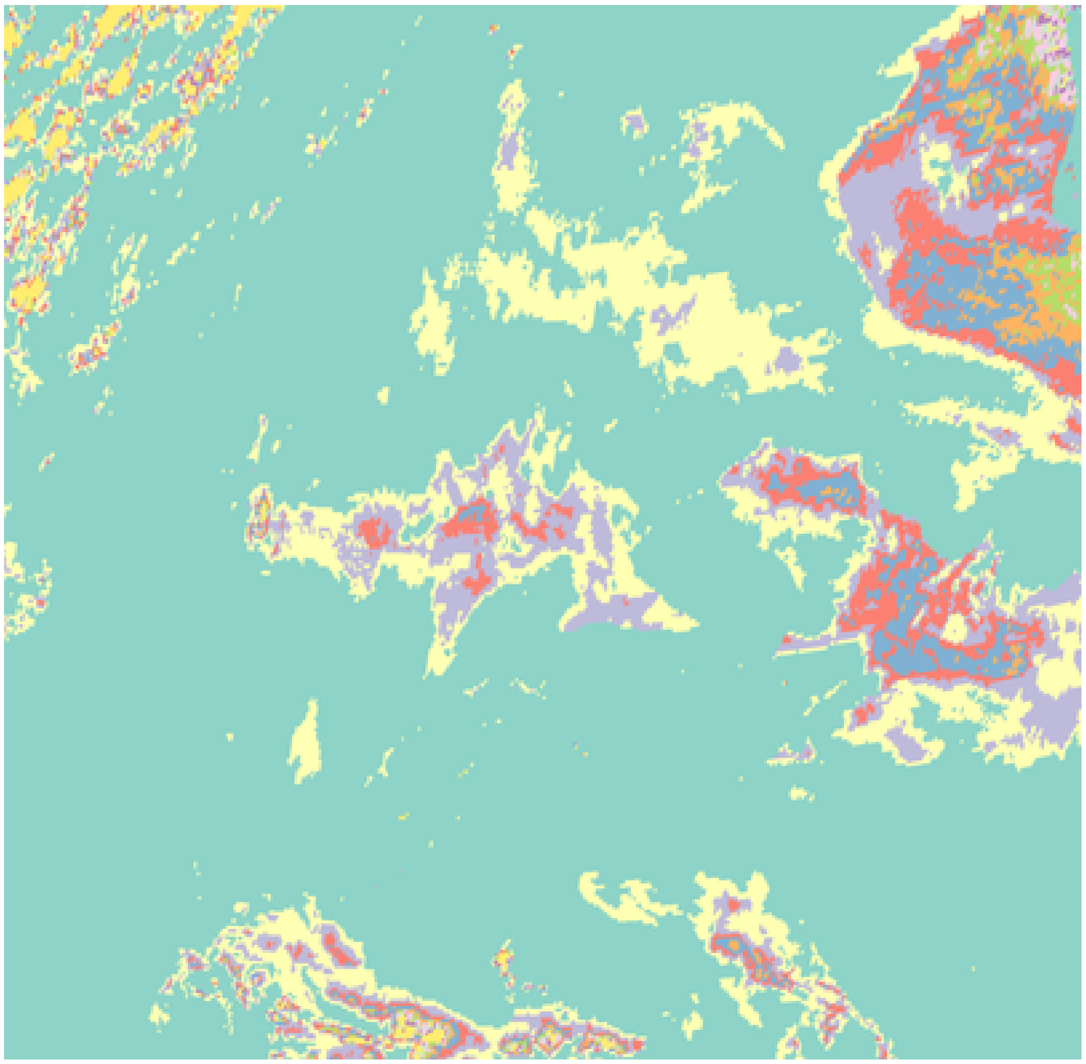


We can now visualise any of those datasets using the regular pytroll visualisation toolkit. Let's first resample the scene again:

```
>>> ls = sc.resample("eurol")
```

And then inspect the cloud optical thickness product:

```
>>> from trollimage.xrimage import XRImage
>>> from trollimage.colormap import set3
>>> xrim = XRImage(ls["cmic_cot"])
>>> set3.set_range(0, 100)
>>> xrim.palettize(set3)
>>> xrim.show()
```



## 2.3 Get hands-on fogpy at daytime

After we imported all required metadata we can continue with a fogpy composite.

**Note:** Make sure that the `PPP_CONFIG_DIR` includes `fogpy/etc/` directory!

Fogpy comes with its own `etc/composites/seviri.yaml`. By setting `PPP_CONFIG_DIR=/path/to/fogpy/etc`, Satpy will find the fogpy composites and all fogpy composites can be used directly in Satpy.

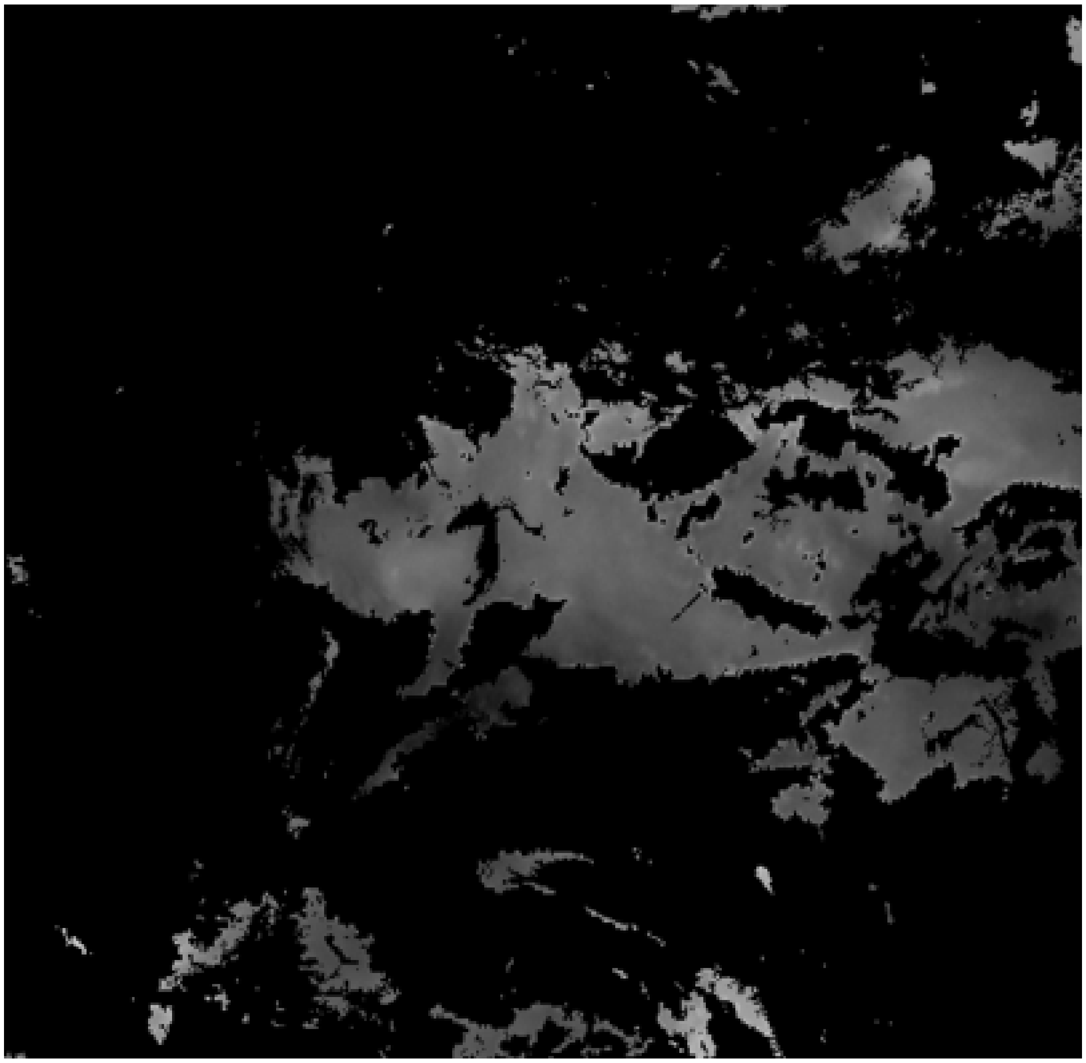
Let's try it with the *fls_day* composite. This composite determines low clouds and ground fog cells from a satellite scene. It is limited to daytime because it requires channels in the visible spectrum to be successfully applicable. We create a fogpy composite for the resampled MSG scene:

```
>>> ls.load(["fls_day"])
```

This may take a while to complete. You see that we don't have to import the fogpy package manually. It's done automagically in the background after the satpy configuration.

The *fls_day* composite function calculates a new dataset, that is now available like any other Satpy dataset, such as by `ls["fls_day"]` or `ls.show("fls_day")`. The dataset has two bands:

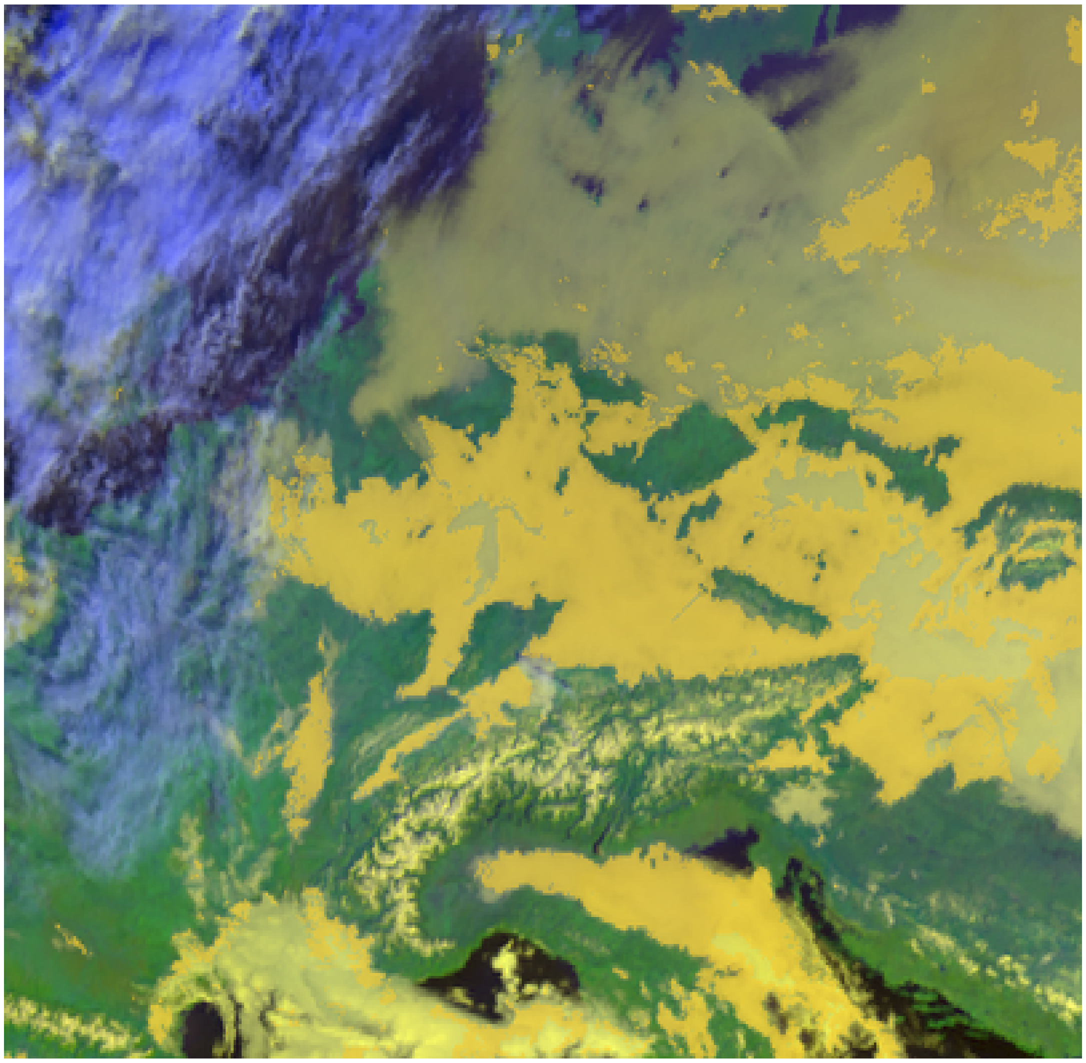
- Band `L` is an image of a selected channel (Default is the 10.8 IR channel) where only the detected ground fog cells are displayed
- Band `A` is an image for the fog mask



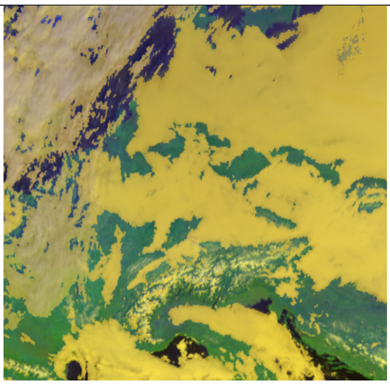
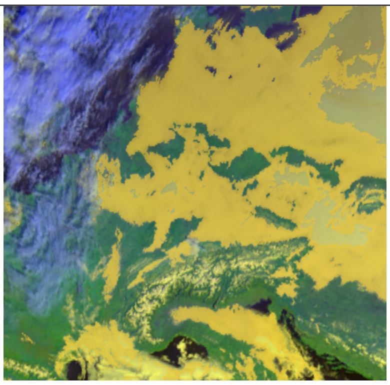
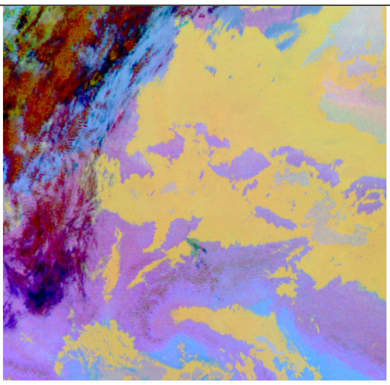
The result image shows the area with potential ground fog calculated by the algorithm, fine. But the remaining areas are missing… maybe a different visualization could be helpful. We can improve the image output by colorize the fog mask and blending it over an overview composite using trollimage:

```
>>> ov = satpy.writers.get_enhanced_image(ls["overview"]).convert("RGBA")
>>> A = ls["fls_day"].sel(bands="A")
>>> Ap = (1-A).where(1-A==0, 0.5)
>>> im = XRImage(Ap)
>>> im.stretch()
>>> im.colorize(fogcol)
>>> RGBA = xr.concat([im.data, Ap], dim="bands")
>>> blend = ov.blend(XRImage(RGBA))
```

**Note:** Images not yet updated!



Here are some example algorithm results for the given MSG scene. As described above, the different masks are blendes over the overview RGB composite in yellow, except the right image where the fog RGB is in the background:

| | | |
|---|---|---|
|  |  |  |
| Cloud mask | Low cloud mask | Low cloud mask + Fog RGB |

It looks like the cloud mask works correctly, except of some missclassified snow pixels in the Alps. But this is not a problem due to the snow filter which successfully masked them out later in the algorithm. Interestingly low cloud areas that are found by the algorithm fit quite good to the fog RGB yellowish areas.

## 2.4 On a foggy night . . .

We saw how daytime fog detection can be realized with the fogpy *fls_day* composite. But mostly fog occuring during nighttime. So let's continue with another composite for nighttime fog detection **fls_night**:.

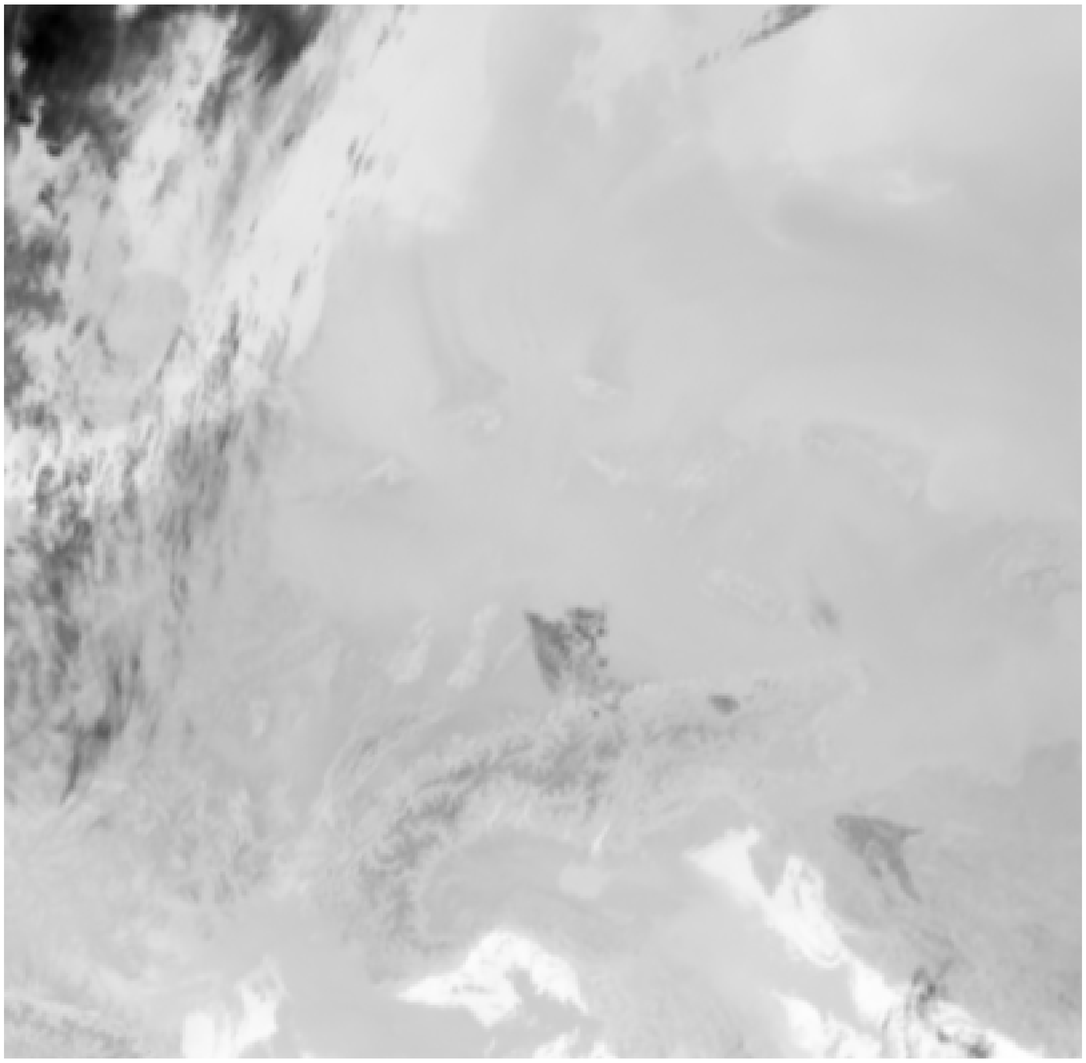
**Note:** Again make sure that the fogpy composites are made available in satpy!

First we need the nighttime MSG scene:

```
>>> fn_nwcsaf = glob("/media/nas/x21308/scratch/NWCSAF/*100000Z.nc") # FIXME: UPDATE!
>>> fn_sev = glob("/media/nas/x21308/scratch/SEVIRI/*201904151000*") # FIXME: UPDATE!
>>> sc = Scene(filenames={"seviri_l1b_hrit": fn_sev, "nwcsaf-geo": fn_nwcsaf})
>>> sc.load(["IR_108", "IR_039", "night_fog"])
```

Reproject it to the central European section from above and have a look at the infrared channel:

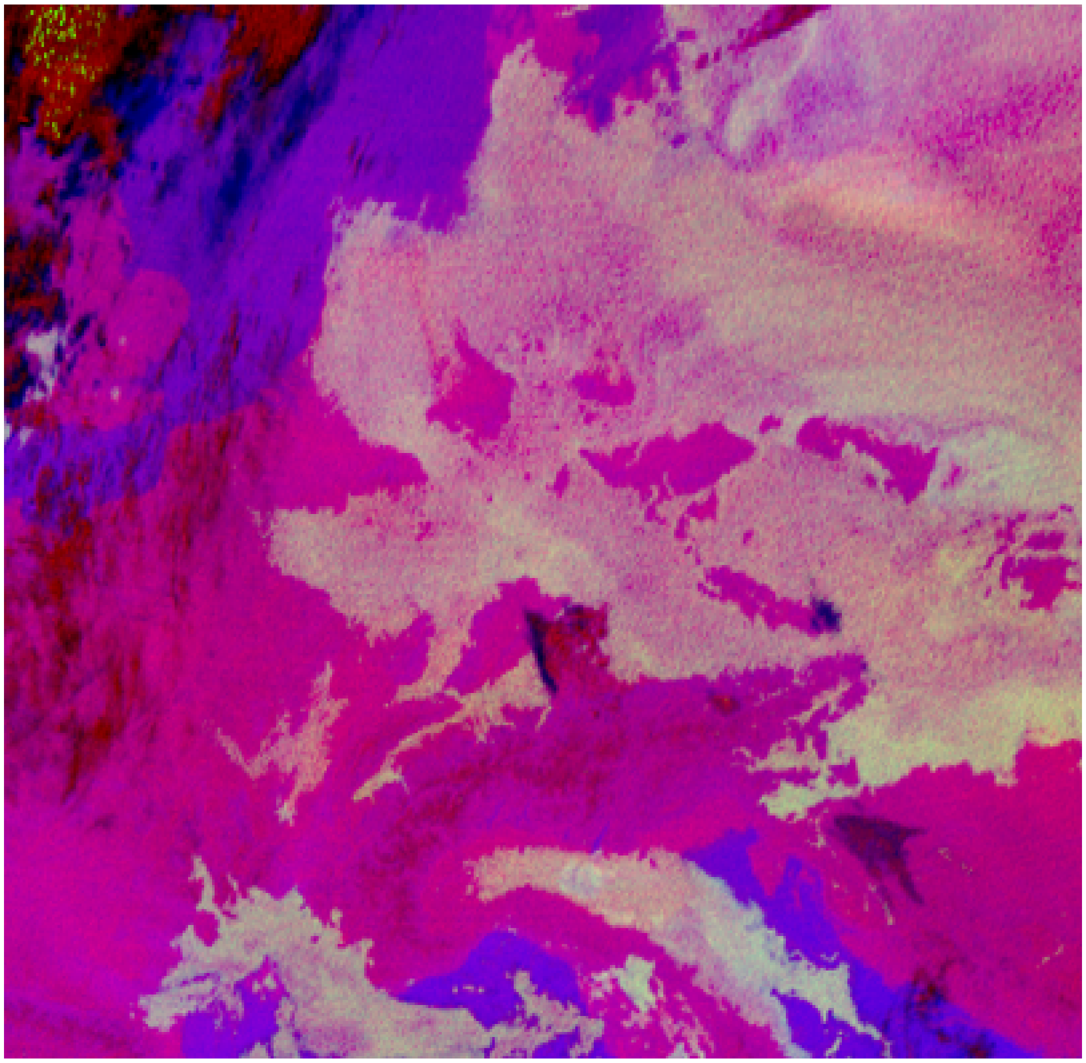
```
>>> ls = sc.resample("eurol")
>>> ls.show(10.8)
```



We took the same day (12. December 2017) as above. Now we could check whether the low clouds, that are present at 10 am, already can be seen early in the the morning (4 am) before sun rise.

So let's look at the nighttime fog RGB product:

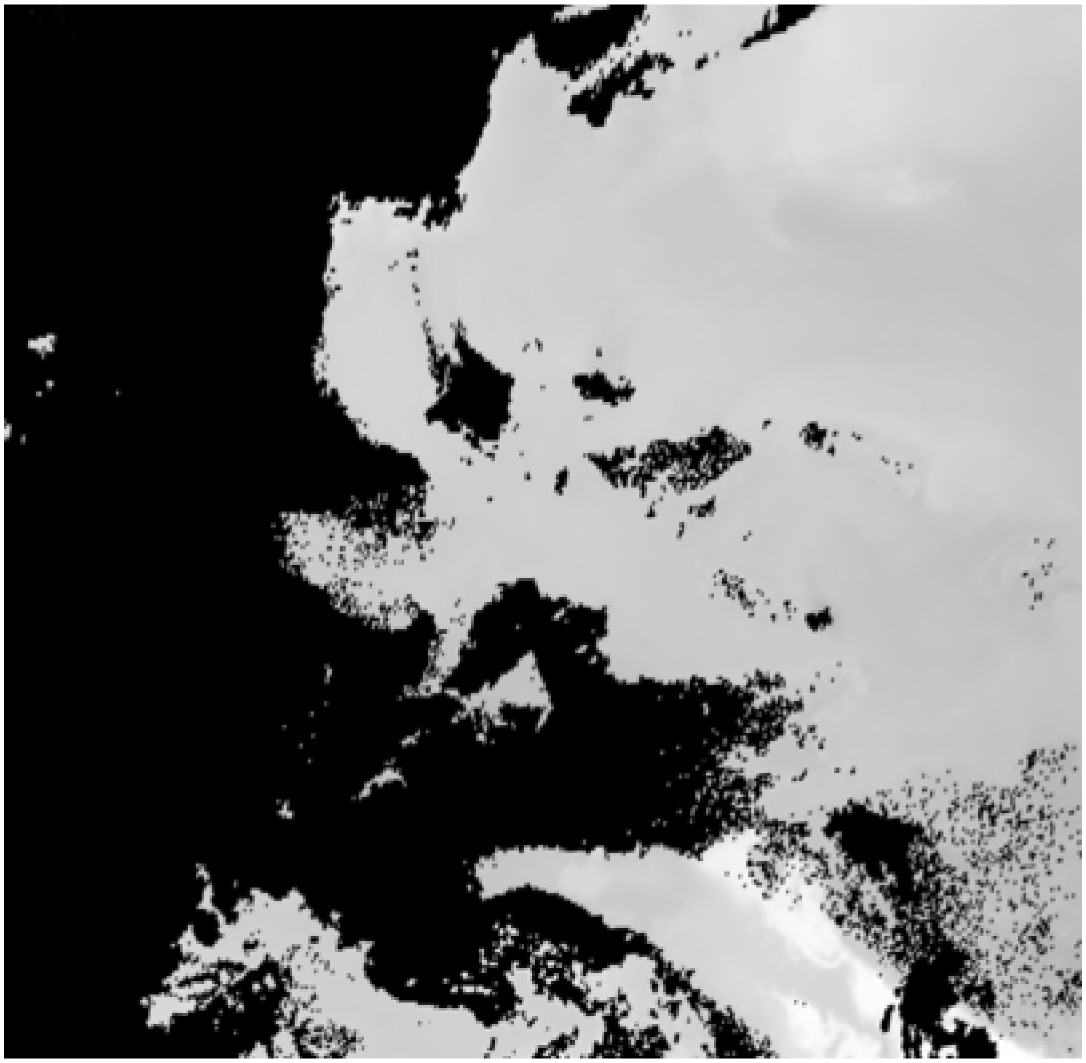
```
>>> ls.show("night_fog")
```



As we see, a lot of greenish-yellow colored pixels are present in the night scene. This is a clear indication for low clouds and fog. In addition these areas have a similar form and distribution as the low clouds in the daytime scene. We can conclude that these low clouds should have formed during the night.

So let's create the fogpy nighttime composite. Fogpy will use the PyTroll package pyorbital for solar zenith angle calculations, so make sure this one is installed. The nightime composite for the resampled MSG scene is generated in the same way like the daytime composite with **'satpy'_**:

```
>>> ls.load(["fls_night"])
>>> ls.show("fls_night")
```



It seems, the detected low cloud cells in the composite overestimate the presence of low clouds, if we compare the RGB product to it. In general, the nighttime algorithm exhibit higher uncertainty for the detection of low clouds than the daytime approach. Therefore a comparison with weather station data could be useful.

## 2.5 Gimme some ground truth!

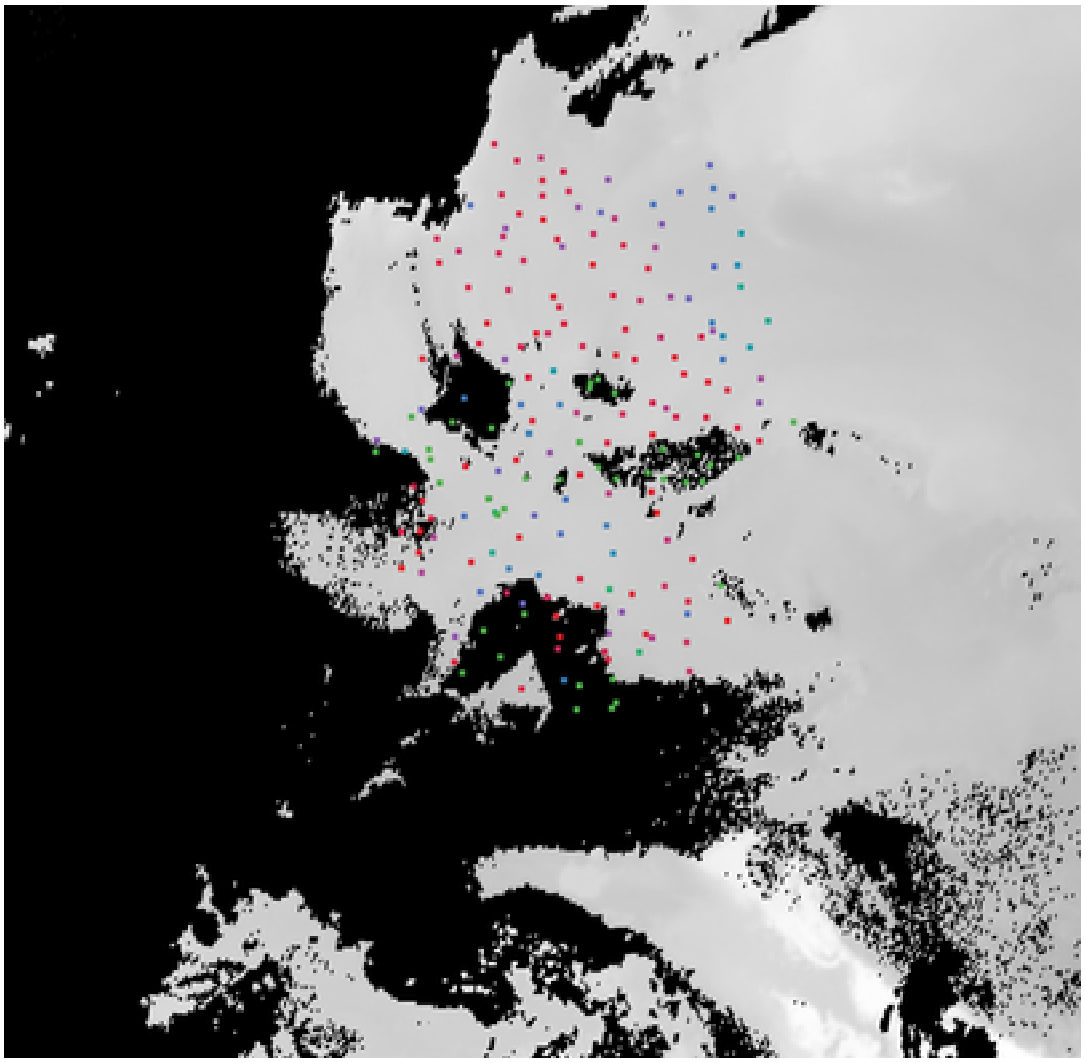
Fogpy features some additional utilities for validation and comparison attempts. This include methods to plot weather station data from Bufr files over the FLS image results. The Bufr data is thereby processed by the trollbufr PyTroll package and the images are generated with trollimage. Here we load visibility data from German weather stations for the nighttime scene:

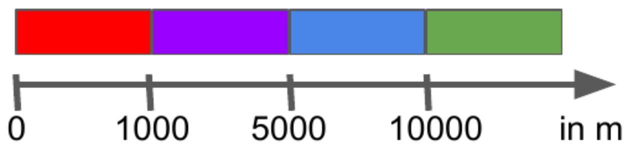
```
>>> import os
>>> from fogpy.utils import add_synop
    # Define search path for bufr file
>>> bufr_dir = '/path/to/bufr/file/'
>>> nbufr_file = "result_{}_synop.bufr".format(ntime.strftime("%Y%m%d%H%M"))
>>> inbufrn = os.path.join(bufr_dir, nbufr_file)
    # Create station image
```

(continues on next page)

(continued from previous page)

```
>>> station_nimg = add_synop.add_to_image(nfls_img, tiffarea, ntime, inbufrn,␣
↪ptsize=4)
>>> station_nimg.show()
```

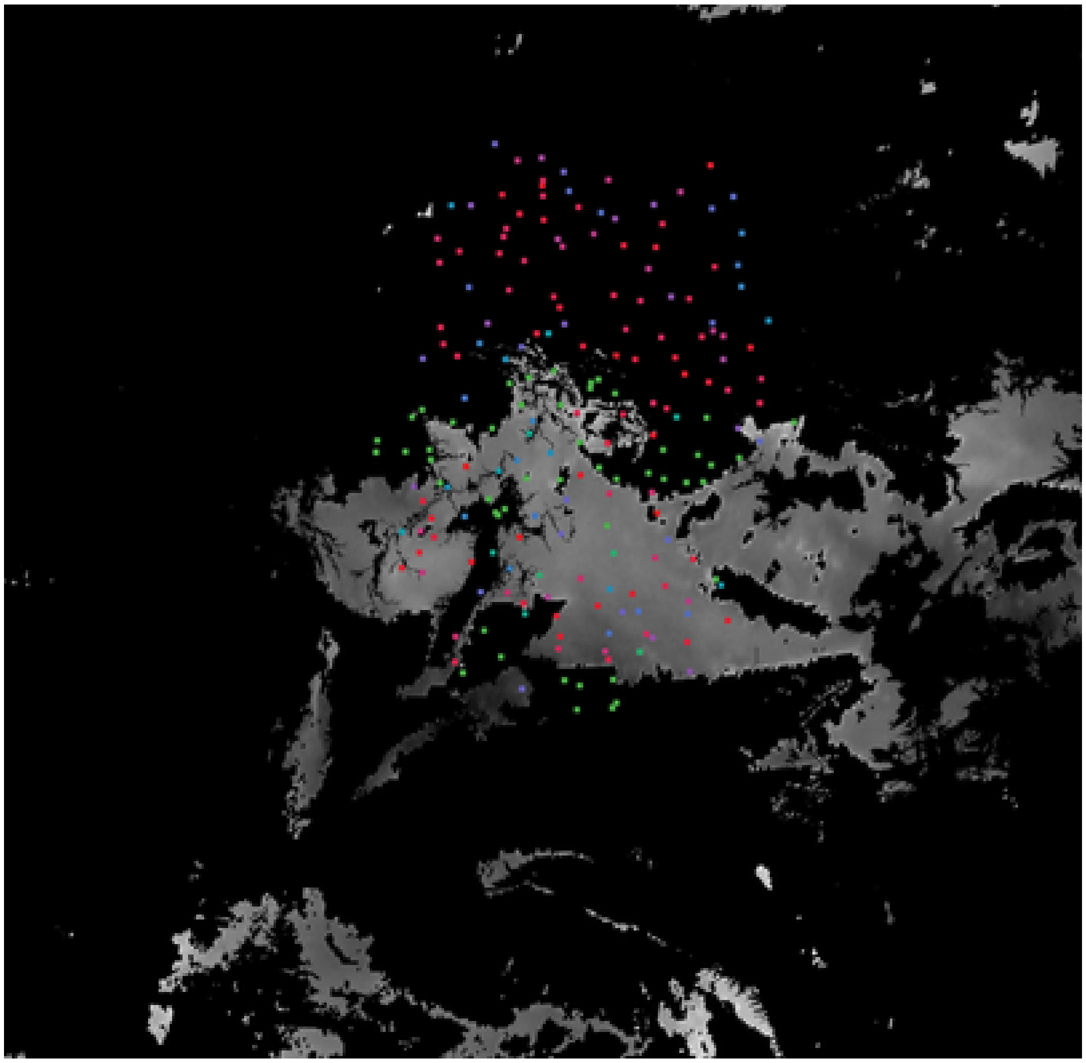




The red dots represent fog reports with visibilities below 1000 meters (compare with legend), whereas green dots show high visibility situations at ground level. We see that low clouds, classified by the nighttime algorithm not always correspond to ground fog. Here the station data is a useful addition to distinguish between ground fog and low stratus.

At daytime we can make the same comparison with station data:

```
>>> bufr_file = "result_{}_synop.bufr".format(time.strftime("%Y%m%d%H%M"))
>>> inbufr = os.path.join(bufr_dir, bufr_file)
    # Create station image
>>> station_img = add_synop.add_to_image(fls_img, tiffarea, time, inbufr, ptsize=4)
>>> station_img.show()
```



We see that the low cloud area in Northern Germany has not been classified as ground fog by the algorithm, whereas the southern part fits quite good to the station data. Furthermore some mountain stations within the area of the ground fog mask exhibit high visibilities. This difference is induced by the averaged evelation from the DEM, the deviated lower cloud height and the real altitude of the station which could lie above the expected cloud top. In addition the low cloud top height assignment can exhibit uncertainty in cases where a elevation based height assignment is not possible and a fixed temperature gradient approach is applied. These missclassifications could be improved by using ground station visibility data as algorithm input. The usage of station data as additional filter could refine the ground fog mask.

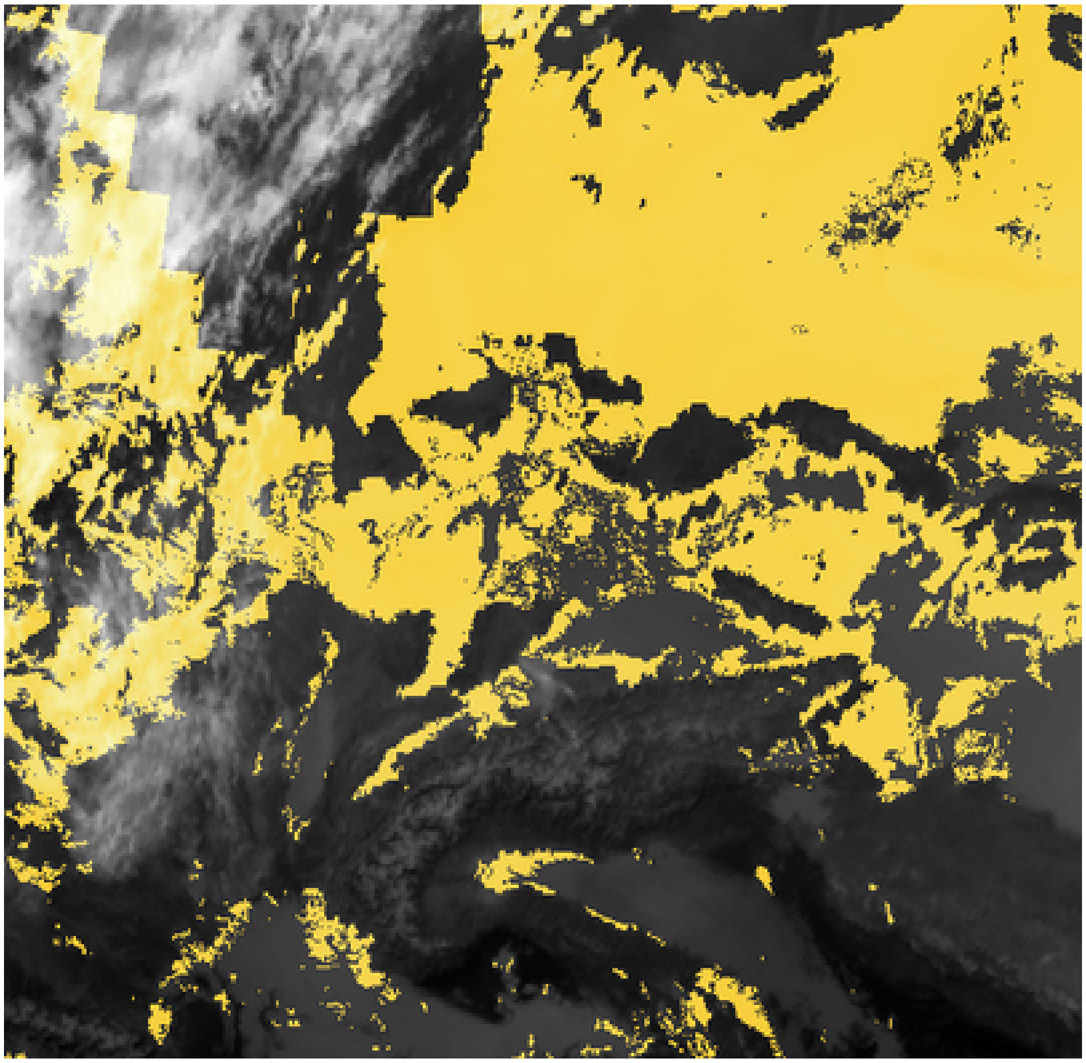
Luckily we can use the StationFusionFilter class from fogpy to combine the satellite mask with ground station visibility data. We use several dataset that had been calculated through out the tour as filter input and plot the filter result:

```
>>> from fogpy.filters import StationFusionFilter
    # Define filter input
```

(continues on next page)

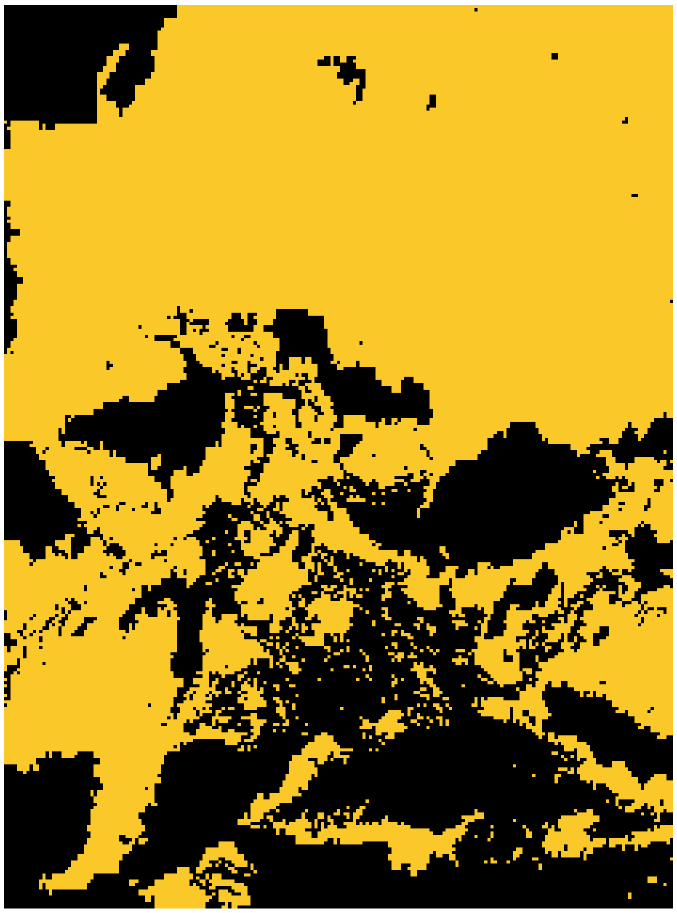
(continued from previous page)

```
>>> flsoutmask = np.array(fogmask.channels[0], dtype=bool)
>>> filterinput = {'ir108': dem_scene[10.8].data,
>>>                'ir039': dem_scene[3.9].data,
>>>                'lowcloudmask': flsoutask,
>>>                'elev': elevation.image_data,
>>>                'bufrfile': inbufr,
>>>                'time': time,
>>>                'area': tiffarea}
    # Create fusion filter
>>> stationfilter = StationFusionFilter(dem_scene[10.8].data, **filterinput)
>>> stationfilter.apply()
>>> stationfilter.plot_filter()
```



The data fusion revise the low cloud clusters in Northern Germany and East Europe as ground fog again. The filter uses ground station data to correct false classification and add missing ground fog cases by utilising a DEM based interpolation. Furthermore cases under high clouds are also extrapolated by elevation information. This cloud lead to low cloud confidence levels. For example the fog mask over France and England. The applicatin of this filter should be limited to a region for which station data is available to achieve a high qualitiy data fusion product. In this case the area should be cropped to Germany, which can be done by setting the *limit* attribute to *True*:

```
>>> filterinput['limit'] = True
    # Create fusion filter with limited region
>>> stationfilter = StationFusionFilter(dem_scene[10.8].data, **filterinput)
>>> stationfilter.apply()
>>> stationfilter.plot_filter()
```



The output is now limited automagically to the area for which station data is available.

The above station fusion filter example can be used to code any other filter application in fogpy. The command sequence more or less looks like the same:

- Prepare filter input
- Instantiate filter class object
- Run the filter
- Enjoy the results

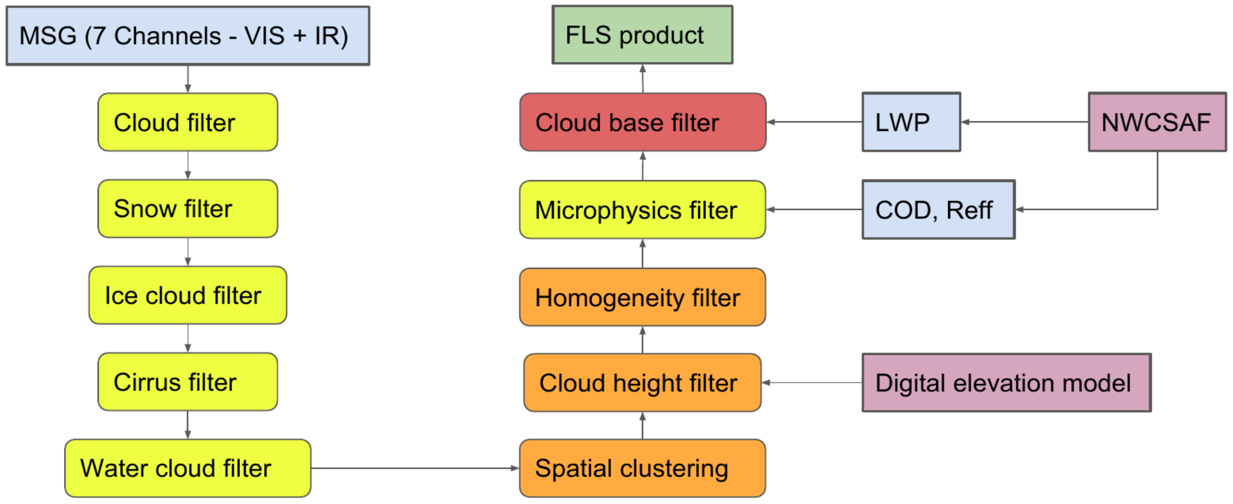
All available filters are listed in the chapter *Filters in fogpy*. Whereas the algorithms that can be directly applied to PyTroll *Scene* objects can be found in the *Algorithms in fogpy* section.

CHAPTER 3

# Algorithms in fogpy

The package provide different algorithms for fog and low stratus cloud detection and nowcasting. The implemented fog algorithms are inherited from a base algorithm class, which defines basic common functionalities for remote sensing procedures.

The fog and low stratus detection algorithm consists of a sequence of different filter approaches that are successively applicated to the given satellite images. The sequence of filters and required inputs are shown in the scheme below:



The cloud microphysical products liquid water path (LWP), cloud optical depth (COD) and effective droplet radius (Reff) can be obtained from the software provided by the Nowcasting Satellite Application Facility (NWCSAF) for example.

## 3.1 Fogpy algorithms

# CHAPTER 4

# Filters in fogpy

The FLS algorithms are based on filter methods for different meteorlogical phenomena or physical variables. All implemented filter methods are inherited from a base filter class, which defines basic common filter functionalities.

## 4.1 Fogpy filters

# CHAPTER 5

# Low cloud model in fogpy

A low water cloud model has been implemented to derive the cloud base height from satellite retrievable variables like liquid water path, cloud top height and temperature.

## 5.1 Low water cloud model

This module implements a class for a 1D low water cloud model. The approach can be used to determine fog cloud base heights by known cloud top height and temperature and cloud liquid water path, e.g. from satellite retrievals. The implemented approch is based on a publication:

- Detecting ground fog from space – a microphysics-based approach Jan Cermak and Joerg Bendi, 2010

**class** `fogpy.lowwatercloud.`**`CloudLayer`**(*bottom*, *top*, *lowcloud*, *add=True*)

This class represent a cloud layer - 1D representation of a cloud section from its vertical profile with defined extent and homogenius cloud parameters. The layer is defined by the bottom and top height in the cloud profile

**classmethod** **`check_temp`**(*temp*, *unit='celsius'*, *debug=False*)

Check for plausible range of temperature value for given unit. Convert if required

**`get_layer_info`**()

**class** `fogpy.lowwatercloud.`**`HeightBounds`**(*xmax=2000*, *xmin=-1000*)

**class** `fogpy.lowwatercloud.`**`LowWaterCloud`**(*cth=None*, *ctt=None*, *cwp=None*, *cbh=0*, *reff=None*, *cbt=None*, *upthres=50.0*, *lowthres=75.0*, *thickness=10.0*, *debug=False*, *nodata=-9999*)

A class to simulate the water content of a low cloud and calculate its meteorological properties.

**Args:**

cth (`float`): Cloud top height in m.

ctt (`float`): Cloud top temperature in K.

cwp (`float`): Cloud water path in kg / m^2.

cbh (`float`): Cloud base height in m.

reff (`float`): Droplet effective radius in m.

cbt (`float`): Cloud base temperature in K.

upthres (`float`): Top layer thickness with dry air entrainment in m.

lowthres (`float`): Bottom layer thickness with ground coupling in m.

thickness (`float`): Layer thickness in m.

debug (`bool`): Boolean to activate additional debug output.

nodata (`float`): Provide a specific Nodata value. Default is: -9999.

**Returns:** Calibrated cloud base height in m.

**`cbh`**

**`classmethod get_air_pressure`**(*z*, *elevation=0*)
Calculate ambient air pressure for height z [hPa].

**`get_cloud_base_height`**(*start=0*, *method='basin'*)
Calculate cloud base height [m].

**`get_cloud_based_vapour_mixing_ratio`**(*debug=False*)

**`get_effective_radius`**(*z*)
The droplet effective radius in [um] for each level is computed on the assumptions that reff retrieved at 3.9 $\mu$m is the cloud top value, Cloud base reff is at 1 $\mu$m and the intermediate values are scaled linearly in between.

**`get_extinct`**(*lwc*, *reff*, *rho*)
Calculate extingtion coeficient [m-1]

The extinction therefore is a combination of radiation loss by (diffuse) scattering and molecular absorption. Required are the liquid water content, effective radius and liquid water density TODO: Recheck the unit of liquid water density g or kg? Should be in g

**`get_fog_base_height`**(*substitude=False*)
This method calculate the fog cloud base height for low clouds with visibilities below 1000 m.

**Args:**

substitude (`bool`): Optional argument to substitude with cbh if no fbh could be found.

**Returns:** Fog base height

**`classmethod get_incloud_mixing_ratio`**(*z*, *cth*, *cbh*, *lowthres=75.0*, *upthres=50.0*)
Calculate in-cloud mixing ratio for given cloud height parameter.

**`get_liquid_density`**(*temp*, *press*)
Calculate the liquid water density in [kg m-3].

**`classmethod get_liquid_mixing_ratio`**(*cb_vmr*, *vmr*, *debug=False*)
Calculate liquid water mixing ratio for given water vapour mixing ratio in a certain height and the maximum water vapour mixing ratio at

cloud base condensation level [g/kg].

**`get_liquid_water_content`**(*z*, *cth*, *hrho*, *lmr*, *beta*, *thres*, *maxlwc=None*, *debug=False*)
Calculate liquid water content [g m-3] by air density and liquid water mixing ratio.

**`get_liquid_water_path`**()
Calculate liquid water path for given cloud layers [g m-2].

**`classmethod get_moist_adiabatic_lapse_temp`**(*z*, *cth*, *ctt*, *convert=False*)
Calculate air temperature for height z [K] following a moist adiabatic lapse rate.

Requires values for cloud top height and temperature e.g. known from satellite retrievals.

**classmethod get_moist_air_density**(*pa*, *pv*, *temp*, *empiric=False*, *debug=False*)
Calculate air density for humid air with known pressure and water vapour pressure and temperature.

**classmethod get_sat_vapour_pressure**(*temp*, *mode='buck'*, *convert=False*, *debug=False*)
Calculate satured water vapour pressure for temperature [hPa] using different empirical approaches.

Options: Buck, Magnus

Convert temperatures in K to °C

**classmethod get_vapour_mixing_ratio**(*pa*, *pv*)
Calculate water vapour mixing ratio for given ambient pressure and water vapour pressure. Also usabale under saturated conditions.

**classmethod get_vapour_pressure**(*z*, *temp*)
Calculate water vapour pressure for height z [hPa].

**get_visibility**(*extinct*, *contrast=0.02*)
Calculate visibility in [m] for given cloud layer. Extinction is directly related to visibility by Koschmieder's law.

**init_cloud_layers**(*init_cbh*, *thickness*, *overwrite=True*)
Method to initialize cloud layers and corresponding parameters. the method needs a initial cloud base height and thickness in [m].

**minimize_cbh**(*x*)
Minimization function for liquid water path.

**optimize_cbh**(*start*, *method='basin'*, *debug=False*)
Find best fitting cloud base height by comparing calculated liquid water path with given satellite retrieval. Minimization with basinhopping or brute force algorithm from python scipy package.

**plot_lowcloud**(*para*, *xlabel=None*, *save=None*)
Plotting of selected low water cloud parameters.

# CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

# Index