Supplement of Earth Syst. Sci. Data, 13, 5545–5563, 2021
https://doi.org/10.5194/essd-13-5545-2021-supplement


Open Access Earth System Science Data

*Supplement of*

# EUREC$^4$A's *HALO*

**Heike Konow et al.**

*Correspondence to:* Heike Konow (heike.konow@mpimet.mpg.de)


# How to EUREC4A

**EUREC4A community**


**Oct 19, 2021**

# CONTENTS

Welcome to the world of EUREC4A data! This is a collection of python code examples to get you started with the data.

# Idea

The book chapters show datasets that are accessible online, i.e. you don't have to download anything. Most datasets are accessed via the eurec4a intake catalog, which simply said takes care of the links to datasets in their most recent version. The scripts typically contain at minimum how to get a specific dataset and some simple plots of basic quantities. Most chapters include additional information from aircraft flight segments or further eurec4a meta data, sometimes more sophisticated plots, or also a combination of variables from different datasets.

## How can you run the code?

In the chapters you will find a rocket icon () in the top bar to the right. It provides two interactive ways to run the code: `Binder` and `LiveCode`. In both cases a virtual environment is created for you in the background by a click on the respective link and you don't have to take care of any requirements locally on your machine. Of course you can also run all the code locally as described in *running How to EUREC4A locally*.

In principle, we anticipate to have at minimum one example script per instrument and we are very happy about contributions by you :) If you miss some information that could be valuable, feel free to check the link on how to contribute to this book and make a pull request or open an issue on github if you are short on time. Thanks!

## Useful links

A list of links to general information about EUREC4A:

- official EUREC4A webpage
- EUREC4A overview papers:[BSA+17] and [SBF+21]
- eurec4a GitHub repository
- AERIS data server
- AERIS leaflet(data visualization)

## License

CHAPTER

# ONE

# HALO

The German High Altitude and Long-Range Research Aircraft (HALO) operated by the Deutsches Zentrum für Luft- and Raumfahrt (DLR) was equiped with remote sensing instruments and a facility to launch dropsondes. All instruments and PIs are listed on the official eurec4a webpage.

During EUREC4A HALO flew most of the time at about 9 km altitude circular patterns with 200 km diameter centered at 57.72W, 13.30N. On typical flight days an excursion to the NTAS buoy or a satellite underpass separated the first three from the remaining three circles. Typically, the aircraft stayed up in the air for 8-9 hours per day starting at varying times in the morning to capture the diurnal cycle.


Please continue on the next page for an *Instrument Overview*.

## 1.1 Instrument Overview

The instrumentation of the HALO aircraft is based on contributions from various universities and research institutions. Each instrument is developed and operated by an individual group, which leads to a modern instrumentation suite and well trained operators, but also requires to contact a range of people when working with data from HALO's instruments. The purpose of this chapter is to introduce you to the individual contact points.

During the EUREC4A field campaign, the HALO aircraft had been outfitted in an updated "Cloud Observatory" configuration, much like as described by [SAB+19]. As the aircraft is build to be flying at high altitudes, most of the instrumentation observes the atmosphere (and in particular clouds) from the nadir perspective.

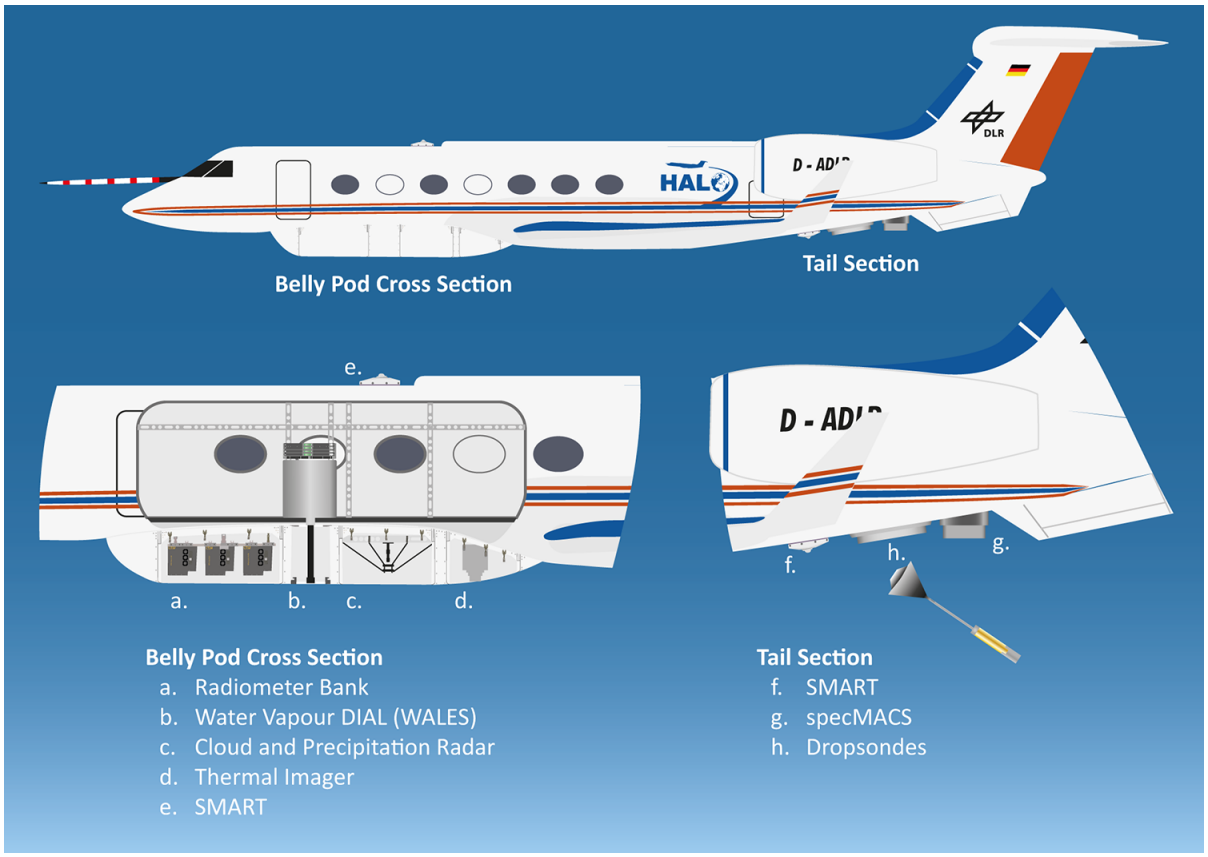



Fig. 1.1: Instruments on HALO as configured for the EUREC4A campaign.

In particular, the instrumentation consists of broadband radiometers (**BACARDI**), basic instrumentation (**BAHAMAS**), a dropsonde launching system (**Dropsondes**), the active (radar) and passive microwave package (**HAMP**), irradiance spectrometers (**SMART**), a thermal imager (**VELOX**), an infrared radiometer (**VELOX-KT19**), a water vapor differential absorption lidar (**WALES**) and spectral and polarization resolving imagers (**specMACS**). You can find more detailed references to the instruments in the tabs below, as well as examples for data access in the upcoming chapters.

### BACARDI

A set of broadband radiometer measuring upward and downward irradiance in the solar (0.2 - 3.6 micrometer) and terrestrial (4.5 - 42 micrometer) spectral range.

**home** Leipzig Institute for Meteorology, Leipzig University

**pi** Manfred Wendisch

**data provider** André Ehrlich

### BAHAMAS

**home** DLR Flugbetrieb, Deutsches Zentrum für Luft- und Raumfahrt

**pi** Andreas Giez

**references** The Transition From FALCON to HALO Era Airborne Atmospheric Research

### Dropsondes

**home** Max-Planck-Institut für Meteorologie

**pi** Sandrine Bony

Bjorn Stevens

Felix Ament

**data provider** Geet George

### HAMP

HAMP consists of a suite of passive microwave radiometers with 26 frequencies ranging between 22.24 and 183.31 GHz, and an active cloud radar measuring at 35.5 GHz. The radiometeres are installed nadir-pointing in a bellypod under the fuselage with opening angles between 2.7 (G-band) and 5.0° (K-band). The almost nadir-pointed radar antenna is located in a bellypod and connected to the electronics in the cabin. Pre-flight calibration was performed before each flight for the radiometers, and before the transfer flight to Barbados for the radar.

**home** Max-Planck-Institut für Meteorologie

Institut für Physik der Atmosphäre, Deutsches Zentrum für Luft- und Raumfahrt

Meteorologisches Institut, Universität Hamburg

**pi** Lutz Hirsch

Felix Ament

Bjorn Stevens

**data provider** Marek Jacob

Heike Konow

Florian Ewald

**references** Calibration of a 35 GHz airborne cloud radar: lessons learned and intercomparisons with 94 GHz cloud radars

HAMP – the microwave package on the High Altitude and LOng range research aircraft (HALO)

### SMART

The Spectral Modular Airborne Radiation measurement sysTem (SMART) measures downward irradiances in the solar spectral range between 300 nm and 2200 nm.

**home** Leipzig Institute for Meteorology, Leipzig University

**pi** Manfred Wendisch

**data provider** Kevin Wolf

**references** An Airborne Spectral Albedometer with Active Horizontal Stabilization

ACRIDICON–CHUVA Campaign: Studying Tropical Deep Convective Clouds and Precipitation over Amazonia Using the New German Research Aircraft HALO

### VELOX

VELOX is a thermal infrared spectral imager (VELOX327k veL, 640 pixel by 512 pixels) with a synchronized filter wheel (at 100 Hz) covering six spectral channels in the thermal infrared wavelength range from 7.7 to 12.0 micrometer. The instrument measures the brightness temperature of upward radiance in a field-of-view of 35.49° by 28.71°.

**home** Leipzig Institute for Meteorology, Leipzig University

**pi** Manfred Wendisch

**data provider** Michael Schäfer

### VELOX - KT19

Thermal infrared radiometer measuring the brightness temperature of nadir radiance between 9.6 and 11.5 micrometer. The field-of-view of 2.3° is located in the center of the VELOX images.

**home** Leipzig Institute for Meteorology, Leipzig University

**pi** Manfred Wendisch

**data provider** Michael Schäfer

#### WALES

**home** Institut für Physik der Atmosphäre, Deutsches Zentrum für Luft- und Raumfahrt

**pi** Martin Wirth

**data provider** Martin Wirth

**references** The airborne multi-wavelength water vapor differential absorption lidar WALES: system design and performance

#### specMACS

specMACS is the hyperspectral imager of the LMU in Munich. It measures from 400nm to 2500nm at a FOV of approximately 32° in a nadir looking perspective

**home** Meteorologisches Institut der LMU, Ludwigs Maximilians Universität

**pi** Tobias Zinner

Bernhard Mayer

Veronika Pörtge

**data provider** Veronika Pörtge

**references** Design and characterization of specMACS, a multipurpose hyperspectral cloud and sky imager

## 1.2 specMACS cloudmask

The following script exemplifies the access and usage of specMACS data measured during EUREC4A.

The specMACS sensor consists of hyperspectral image sensors as well as polarization resolving image sensors. The hyperspectral image sensors operate in the visible and near infrared (VNIR) and the short-wave infrared (SWIR) range. The dataset investigated in this notebook is a cloud mask dataset based on data from the SWIR sensor. More information on the dataset can be found on the macsServer. If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` and `author` list.

Our plan is to analyze a section of the specMACS cloud mask dataset around the first `GOOD` dropsonde on the second HALO circle on the 5th of February (`HALO-0205_c2`). We'll first look at the cloudiness just along the flight track and later on create a map projection of the dataset. This notebook will guide you through the required steps.

### 1.2.1 Obtaining data

In order to work with EUREC4A datasets, we'll use the `eurec4a` library to access the datasets and also use `numpy` and `xarray` as our common tools to handle datasets.

```
import eurec4a
import numpy as np
import xarray as xr
```

### Finding the right sonde

All HALO flights were split up into flight phases or segments to allow for a precise selection in time and space of a circle or calibration pattern. For more information have a look at the respective github repository.

```
all_flight_segments = eurec4a.get_flight_segments()
```

The flight segmentation data is organized by platform and flight, but as we want to look up a segment by its `segment_id`, we'll have to flatten the data and reorganize it by the `segment_id`. Additionally, we want to be able to extract the `flight_id` and `platform_id` back from a selected segment, so we add this information to each individual segment:

```
segments_by_segment_id = {
    s["segment_id"]: {
        **s,
        "platform_id": platform_id,
        "flight_id": flight_id
    }
    for platform_id, flights in all_flight_segments.items()
    for flight_id, flight in flights.items()
    for s in flight["segments"]
}
```

We want to extract the id of the first dropsonde of the second HALO circle on February 5 (`HALO-0205_c2`). By the way: In the *VELOX* example the measurement of this time is shown as well.

```
segment = segments_by_segment_id["HALO-0205_c2"]
first_dropsonde = segment["dropsondes"]["GOOD"][0]
first_dropsonde
```

```
'HALO-0205_s13'
```

Now we would like to know when this sonde was launched. The JOANNE dataset contains this information. We use level 3 data which contains quality checked data on a common grid and load it using the intake catalog. We then select the correct dropsonde by its `sonde_id` to find out the launch time of the dropsonde.

More information on the data catalog can be found here.

```
cat = eurec4a.get_intake_catalog()
```

```
dropsondes = cat.dropsondes.JOANNE.level3.to_dask()
# this following line should go away in the soon to be released new version of JOANNE
sonde_dt = dropsondes.swap_dims({"sounding": "sonde_id"}).sel(sonde_id=first_
 ↪dropsonde).launch_time.values
str(sonde_dt)
```

```
'2020-02-05T10:48:32.000000000'
```

### Finding the corresponding specMACS dataset

As specMACS data is organized as one dataset per flight, we have to obtain the `flight_id` from our selected segment to obtain the corresponding dataset. From this dataset, we select a two minute long measurement sequence of specMACS data around the dropsonde launch and have a look at the obtained dataset:

```
offset_time = np.timedelta64(60, "s")

ds = cat.HALO.specMACS.cloudmaskSWIR[segment["flight_id"]].to_dask()
ds_selection = ds.sel(time=slice(sonde_dt-offset_time, sonde_dt+offset_time))
ds_selection
```

```
<xarray.Dataset>
Dimensions:     (angle: 318, time: 3564)
Coordinates:
  * time        (time) datetime64[ns] 2020-02-05T10:47:32.015175168 ... 2020-...
  * angle       (angle) float64 18.0 17.9 17.8 17.69 ... -17.07 -17.17 -17.27
    lat         (time) float64 ...
    lon         (time) float64 ...
    alt         (time) float32 ...
Data variables:
    CF_min      (time) float64 ...
    CF_max      (time) float64 ...
    cloud_mask  (time, angle) float32 ...
    vza         (time, angle) float32 ...
    vaa         (time, angle) float32 ...
Attributes: (12/16)
    title:                          cloud mask derived from specMACS SWIR ca...
    version:                        2.1
    variable:                       cloud_mask
    comment:                        The additional information about the vie...
    contact:                        veronika.poertge@physik.uni-muenchen.de
    platform:                       HALO
    ...                             ...
    author:                         Veronika Pörtge, Felix Gödde, Tobias Köl...
    history:                        The original version of the cloud mask w...
    created_on:                     2021-03-12
    Conventions:                    CF-1.8
    references:                     more information about the product can b...
    DODS_EXTRA.Unlimited_Dimension: time
```

You can get a list of available variables in the dataset from `ds_selection.variables.keys()` or by looking them up in the table above.

## 1.2.2 First plots

After selecting our datasets, we want to see what's inside, so here are some first plots.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book", "./mplstyle/wide"])
```

First, we create a little helper to properly display a colorbar for categorical (in CF-Convention terms "flag") variables:

```
def flagbar(fig, mappable, variable):
    ticks = variable.flag_values
```

(continues on next page)

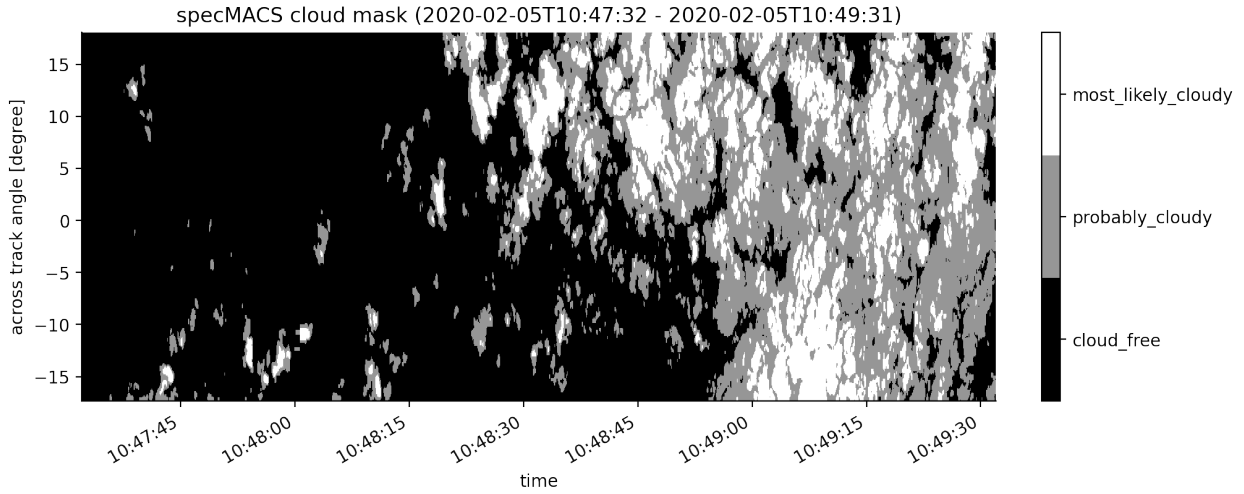
(continued from previous page)

```
    labels = variable.flag_meanings.split(" ")
    cbar = fig.colorbar(mappable, ticks=ticks)
    cbar.ax.set_yticklabels(labels);
```

Figure 1: shows the SWIR camera cloud mask product along the flight track (x axis) for all observations in across track directions (y axis).

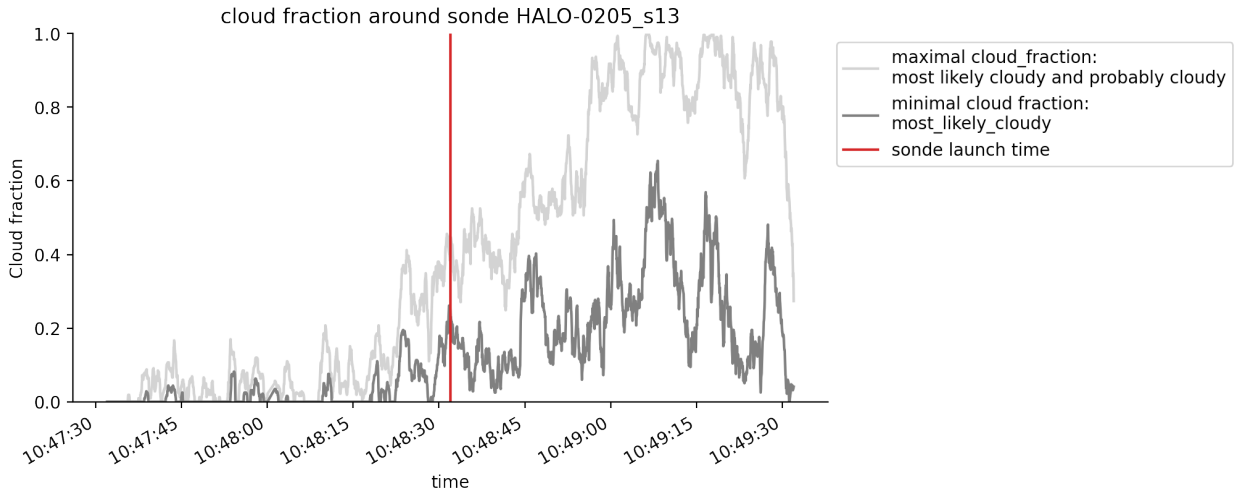
**Note:** fetching the data and displaying it might take a few seconds

```
fig, ax = plt.subplots()
cmplot = ds_selection.cloud_mask.T.plot(ax=ax,
                                        cmap=plt.get_cmap('Greys_r', lut=3),
                                        vmin=-0.5, vmax=2.5,
                                        add_colorbar=False)
flagbar(fig, cmplot, ds_selection.cloud_mask)
start_time_rounded, end_time_rounded = ds_selection.time.isel(time=[0, -1]).values.
 ↪astype('datetime64[s]')
ax.set_title(f"specMACS cloud mask ({start_time_rounded} - {end_time_rounded})");
```



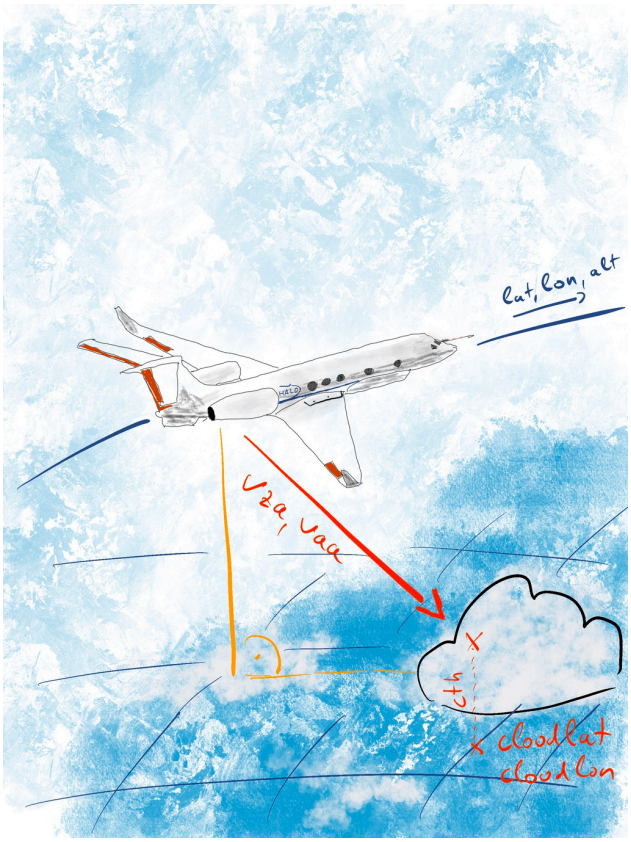

The dataset also contains the minimal and maximal cloud fractions which are calculated for each temporal measurement. The minimal cloud fraction is derived from the ratio between the number of pixels classified as “most likely cloudy” and the total number of pixels in one measurement (times with unknown measurements are excluded). The maximal cloud fraction additionally includes the number of pixels classified as “probably cloudy”.

```
fig, ax = plt.subplots()
ds_selection.CF_max.plot(color="lightgrey",label="maximal cloud_fraction:\nmost␣
 ↪likely cloudy and probably cloudy")
ds_selection.CF_min.plot(color="grey", label="minimal cloud fraction:\nmost_likely_
 ↪cloudy")
ax.axvline(sonde_dt, color="C3", label="sonde launch time")
ax.set_ylim(0, 1)
ax.set_ylabel("Cloud fraction")
ax.set_title(f"cloud fraction around sonde {first_dropsonde}")
ax.legend(bbox_to_anchor=(1,1), loc="upper left");
```




### 1.2.3 Camera view angles to latitude and longitude

The cloud mask is given on a `time` × `angle` grid where `angles` are the internal camera angles. Sometimes it could be helpful to project the data onto a map. This means that we would like to know the corresponding **latitude** and **longitude** coordinates of each pixel of the cloud mask.




The computation involves quite a few steps, so let's take a second to lay out a strategy. We know already the position of the aircraft and the individual viewing directions of each sensor pixel at each point in time. If we would start from the position of the aircraft and continue along each line of sight, until we hit something which we see in our dataset, we

should end up at the location we are interested in. Due to the curvature of the earth, we'll have to do a few coordinate transformations in the process, but no worries, we'll walk you through.

### What we know

We start out with five variables:

- position of the airplane: latitude, longitude and height above WGS84 (`ds.lat, ds.lon, ds.alt`)
- viewing directions of the camera pixels: viewing zenith angle and viewing azimuth angle (`ds.vza, ds.vaa`)

Let's have a look a these variables

```
attr_table([ds_selection[key] for key in ["lat", "lon", "alt", "vza", "vaa"]])
```

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

- The position of the HALO is saved in ellipsoidal coordinates. It is defined by the latitude (`lat`), longitude (`lon`) and height (`alt`) coordinates with respect to the WGS-84 ellipsoid.
- The viewing zenith (`vza`) and azimuth (`vaa`) angles are given with respect to the local horizon (`lh`) coordinate system at the position of the HALO. This system has its center at the `lat`/`lon`/`alt` position of the HALO and the x/y/z axis point into North, East and down directions.

As the frame of reference rotates with the motion of HALO, a convenient way to work with such kind of data is to transform it into the Earth-Centered, Earth-Fixed (ECEF) coordinate system. The origin of this coordinate system is the center of the Earth. The z-axis passes through true north, the x-axis through the Equator and the prime meridian at 0° longitude. The y-axis is orthogonal to x and z. This cartesian system makes computations of distances and angles very easy.

---

**Note:** The use of "altitude" and "height" can be confusing and is sometimes handled inconsistently. Usually, "altitude" describes the vertical distance above the geoid and "height" describes the vertical distance above the surface.

In this notebook, we only use the vertical distance above the WGS84 reference ellipsoid. Neither "altitude" nor "height" as defined above matches this distance exactly, but both are commonly used. We try to use the term "height" as it seems to be more commonly used in coordinate computations and fall back to the term `alt` when talking about the variable in HALO datasets. Keep in mind that we actually mean vertical distance above WGS84 in both cases.

---

### Transformation to ECEF

In a first step we want to transform the position of the HALO into the ECEF coordinate system. We use the method from ESA navipedia:

With these functions it is easy to transform the `lat`/`lon`/`alt` position of the HALO into the ECEF-coordinate system.

```
HALO_ecef = ellipsoidal_to_ecef_ds(ds_selection)
```

### Computing viewing lines

As a next step we want to set up the vector of the viewing direction. We will need the rotation matrices `Rx`, `Ry` and `Rz` for this. Using these fundamental rotation matrices, we can compute vectors along the instruments viewing direction in the local horizon (NED) coordinate system.

```
view_lh = vector_lh(ds_selection)
```

We need an approximation of the cloud top height and will use 1000 m as a first guess. If you have better cloud height data just put in your data. You can also use different values along `time` and `angle` dimensions, xarray will take care of this.

```
height_guess = 1000. #m
cth = xr.DataArray(height_guess, dims=())
```

Now let's calculate the length of the vector connecting HALO and the cloud. We need the height of the HALO, the cloudheight and the viewing direction for this. `view_lh` provides the viewing direction while the distance between HALO and cloud normalized by the down-component of the viewing direction provides the distance along the view path.

```
viewpath_lh = view_lh * (ds_selection.alt - cth) / view_lh.sel(NED="D")
```

Now we would like to transform this viewpath also into the ECEF coordinate system. We will stick to the method described in navipedia:

```
viewpath_ecef = lh_to_ecef(viewpath_lh,
                           ds_selection["lat"],
                           ds_selection["lon"])
```

### Cloud locations in 3D space

If we add the viewpath to the position of the HALO the resulting point gives us the ECEF coordinates of the point on the cloud.

```
cloudpoint_ecef = HALO_ecef + viewpath_ecef
x_cloud, y_cloud, z_cloud = cloudpoint_ecef.transpose("xyz_ecef", "time", "angle")
```

### Back to ellipsoidal coordinates

The last step is to transform these coordinates into ellipsoidal coordinates. Again we stick to the description at navipedia:

```
ds_selection = ds_selection.assign_coords(
    {"cloud" + k: v for k, v in ecef_to_ellipsoidal(x_cloud, y_cloud, z_cloud,␣
 ↪iterations=10).items()}
)
ds_selection
```

```
<xarray.Dataset>
Dimensions:      (angle: 318, time: 3564)
Coordinates:
  * time         (time) datetime64[ns] 2020-02-05T10:47:32.015175168 ... 2020...
  * angle        (angle) float64 18.0 17.9 17.8 17.69 ... -17.07 -17.17 -17.27
    lat          (time) float64 14.3 14.3 14.3 14.3 ... 14.26 14.26 14.26 14.26
    lon          (time) float64 -57.67 -57.67 -57.67 ... -57.42 -57.42 -57.42
```

(continues on next page)

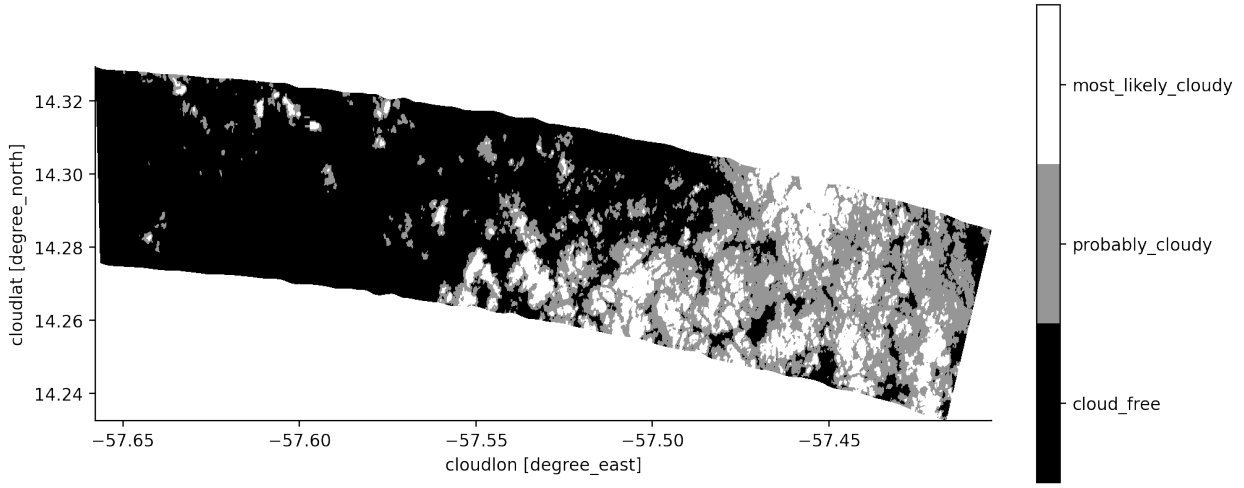
(continued from previous page)

```
    alt           (time) float32 1.026e+04 1.026e+04 ... 1.026e+04 1.026e+04
    cloudlat      (time, angle) float64 14.28 14.28 14.28 ... 14.28 14.28 14.28
    cloudlon      (time, angle) float64 -57.66 -57.66 -57.66 ... -57.4 -57.4
    cloudheight   (time, angle) float64 1.001e+03 1.001e+03 ... 1.001e+03
Data variables:
    CF_min        (time) float64 0.0 0.0 0.0 0.0 ... 0.03774 0.03774 0.04088
    CF_max        (time) float64 0.0 0.0 0.0 0.0 ... 0.3396 0.3491 0.3302 0.2736
    cloud_mask    (time, angle) float32 ...
    vza           (time, angle) float32 16.09 15.98 15.89 ... 20.47 20.57 20.67
    vaa           (time, angle) float32 159.0 158.9 158.8 ... 28.77 28.69 28.61
Attributes: (12/16)
    title:                              cloud mask derived from specMACS SWIR ca...
    version:                            2.1
    variable:                           cloud_mask
    comment:                            The additional information about the vie...
    contact:                            veronika.poertge@physik.uni-muenchen.de
    platform:                           HALO
    ...                                 ...
    author:                             Veronika Pörtge, Felix Gödde, Tobias Köl...
    history:                            The original version of the cloud mask w...
    created_on:                         2021-03-12
    Conventions:                        CF-1.8
    references:                         more information about the product can b...
    DODS_EXTRA.Unlimited_Dimension:     time
```

### 1.2.4 Display on a map

`ds.cloudlon` and `ds.cloudlat` are the projected longitude and latitude coordinates of the cloud. Now it is possible to plot the cloudmask on a map!

```
fig, ax = plt.subplots()
cmplot = ds_selection.cloud_mask.plot(ax=ax,
                                      x='cloudlon', y='cloudlat',
                                      cmap=plt.get_cmap('Greys_r', lut=3),
                                      vmin=-0.5, vmax=2.5,
                                      add_colorbar=False)
# approximate aspect ratio of latitude and longitude length scales
ax.set_aspect(1./np.cos(np.deg2rad(float(ds_selection.cloudlat.mean()))))
flagbar(fig, cmplot, ds_selection.cloud_mask);
```




In this two-minute long sequence we can see the curvature of the HALO circle. By the way: you could also calculate the swath width of the measurements projected onto the cloud height that you specified.

### 1.2.5 Swath width

The haversine formula is such an approximation. It calculates the great-circle distance between two points on a sphere. We just need the respective latitudes and longitudes of the points. We will use the coordinates of the two edge pixels of the first measurement.

```
distance_haversine_formula = haversine_distance(lat1, lat2, lon1, lon2, R = 6371)
print('The across track swath width of the measurements is about {} km'.format(np.
 ↪round(distance_haversine_formula.values,2)))
```

```
The across track swath width of the measurements is about 5.96 km
```

## 1.3 HALO UNIFIED dataset

The HALO UNIFEID dataset is a combination of HAMP radar and radiometer measurements, BAHAMAS aircraft position data and dropsonde measurements, all on a unified temporal and spatial grid.

More information on the dataset can be found at [Kon21]. If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` or `author`.

### 1.3.1 Get data

- To load the data we first load the EUREC4A meta data catalogue. More information on the catalog can be found here.

```
import eurec4a
```

```
cat = eurec4a.get_intake_catalog()
list(cat.HALO.UNIFIED)
```

```
['dropsondes',
 'HAMPradar',
 'HAMPradiometer',
 'BAHAMAS',
 'HAMPradiometer_cloudmask',
 'HAMPradar_cloudmask',
 'HAMPradiometer_retrievals']
```

- We can further specify an instrument and a flight and obtain the dataset using `to_dask`.

**Note:** Have a look at the attributes of the xarray dataset `ds` for all relevant information on the dataset, such as author, contact, or citation infromation.

```
ds_radar = cat.HALO.UNIFIED.HAMPradar["HALO-0205"].to_dask()
ds_radiometer = cat.HALO.UNIFIED.HAMPradiometer["HALO-0205"].to_dask()
ds_bahamas = cat.HALO.UNIFIED.BAHAMAS["HALO-0205"].to_dask()
```

### 1.3.2 Load HALO flight phase information

All HALO flights were split up into flight phases or segments to allow for a precise selection in time and space of a circle or calibration pattern. For more information have a look at the respective github repository.

```
meta = eurec4a.get_flight_segments()
```

We select the flight phase we are interested in, e.g. the second circle on February 5 by it's segment_id.

```
segments = {s["segment_id"]: {**s, "flight_id": flight["flight_id"]}
             for platform in meta.values()
             for flight in platform.values()
             for s in flight["segments"]
            }
seg = segments["HALO-0205_c2"]
```

We transfer the information from our flight segment selection to our radar and radiometer data in the xarray dataset.

```
ds_radar_selection = ds_radar.sel(time=slice(seg["start"], seg["end"]))
ds_radiometer_selection = ds_radiometer.sel(time=slice(seg["start"], seg["end"]))
ds_bahamas_selection = ds_bahamas.sel(time=slice(seg["start"], seg["end"]))
```

### 1.3.3 Plots

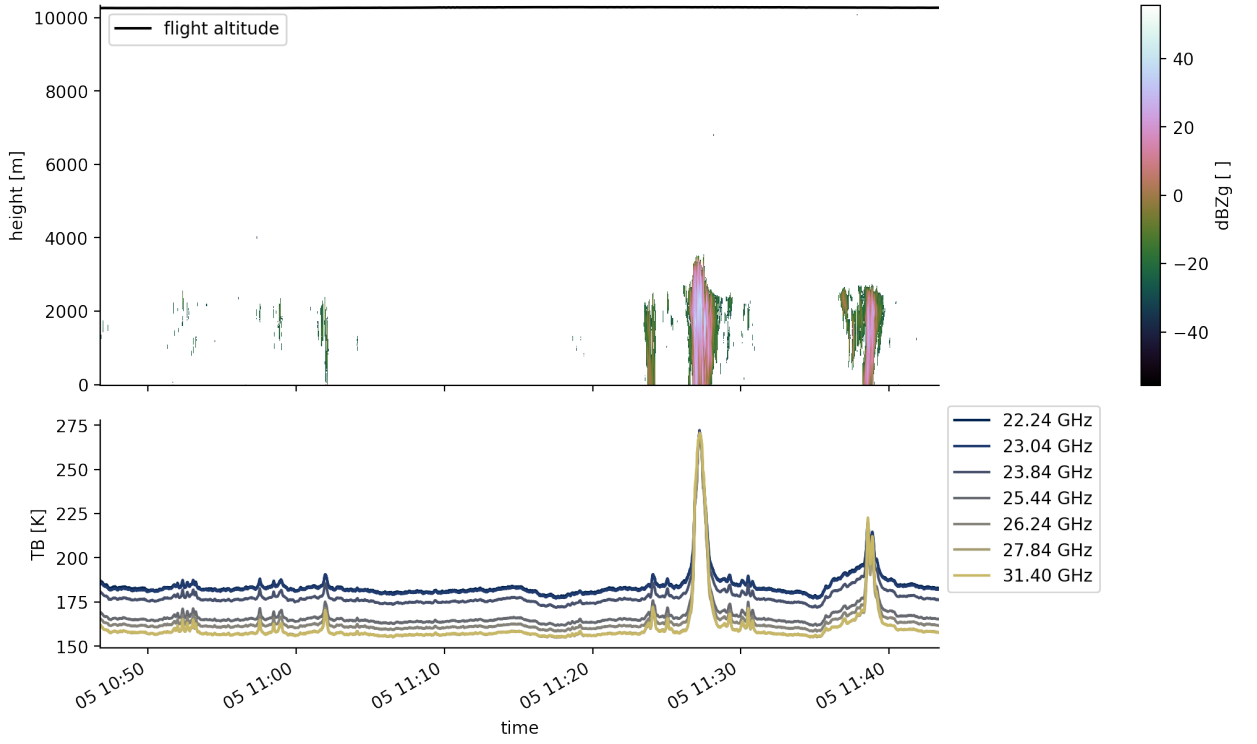
We plot reflectivity from the HAMP Radar, the flight altitude of HALO and brightness temperatures from the low-frequency channels along the 22 GHz water vapor line (K band) from the HAMP radiometer.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True, gridspec_kw={'height_ratios':(2, 1.
↪2)})
```

(continues on next page)

(continued from previous page)

```
# 1st plot: Radar dBZ and flight altitude
ds_bahamas_selection.altitude.plot(ax=ax1, x='time', color='black', label='flight␣
 ↪altitude')
ax1.legend(loc ='upper left')
ds_radar_selection.dBZ.plot(ax=ax1, x='time', cmap='cubehelix' )
ax1.set_xlabel('')

# 2nd plot: Radiometer TB
## select low frequency channels along the 22 GHz water vapor line
low_freq = ds_radiometer_selection.frequency < 32
ds_radiometer_low_freq = ds_radiometer_selection.isel(frequency=low_freq)
## set line colors for 2nd plot
colors2 = plt.get_cmap("cividis")(np.linspace(0, 0.8, low_freq.values.sum()))
ax2.set_prop_cycle(color=colors2)
for frequency, data_radiometer in ds_radiometer_low_freq.groupby("frequency"):
    data_radiometer.tb.plot(ax=ax2, x='time', label=f'{frequency:.2f} GHz')
ax2.set_title('')
ax2.legend(bbox_to_anchor=(1,1.1))
None
```



## 1.4 interactive HALO tracks

This notebook shows how to plot the flight tracks of the HALO aircraft. First, we'll have only a quick look and later on we do some more work in order to get the data onto an interactive map.

### 1.4.1 Preparations

First, we'll import pylab and the EUREC4A data catalog.

```
import eurec4a
cat = eurec4a.get_intake_catalog()
```

#### Inspecting the dataset

```
ds = cat.HALO.BAHAMAS.PositionAttitude['HALO-0122'].to_dask()
ds
```

```
<xarray.Dataset>
Dimensions:     (time: 331760)
Coordinates:
  * time        (time) datetime64[ns] 2020-01-22T14:57:36 ... 2020-01-23T00:1...
    lat         (time) float64 ...
    lon         (time) float64 ...
    alt         (time) float32 ...
Data variables:
    roll        (time) float32 ...
    pitch       (time) float32 ...
    heading     (time) float32 ...
    tas         (time) float32 ...
    trajectory  |S128 ...
Attributes: (12/14)
    title:                          EUREC4A position and attitude data
    description:                    10Hz subset based on DLR BAHAMAS data, p...
    mission:                        EUREC4A
    platform:                       HALO
    instrument:                     BAHAMAS
    flight_id:                      HALO-0122
    ...                             ...
    featureType:                    trajectory
    Conventions:                    CF-1.7
    version:                        1.1
    history:                        acquired by HALO BAHAMAS, processed and ...
    file_created:                   File created by M. Klingebiel (email: ma...
    DODS_EXTRA.Unlimited_Dimension:  time
```

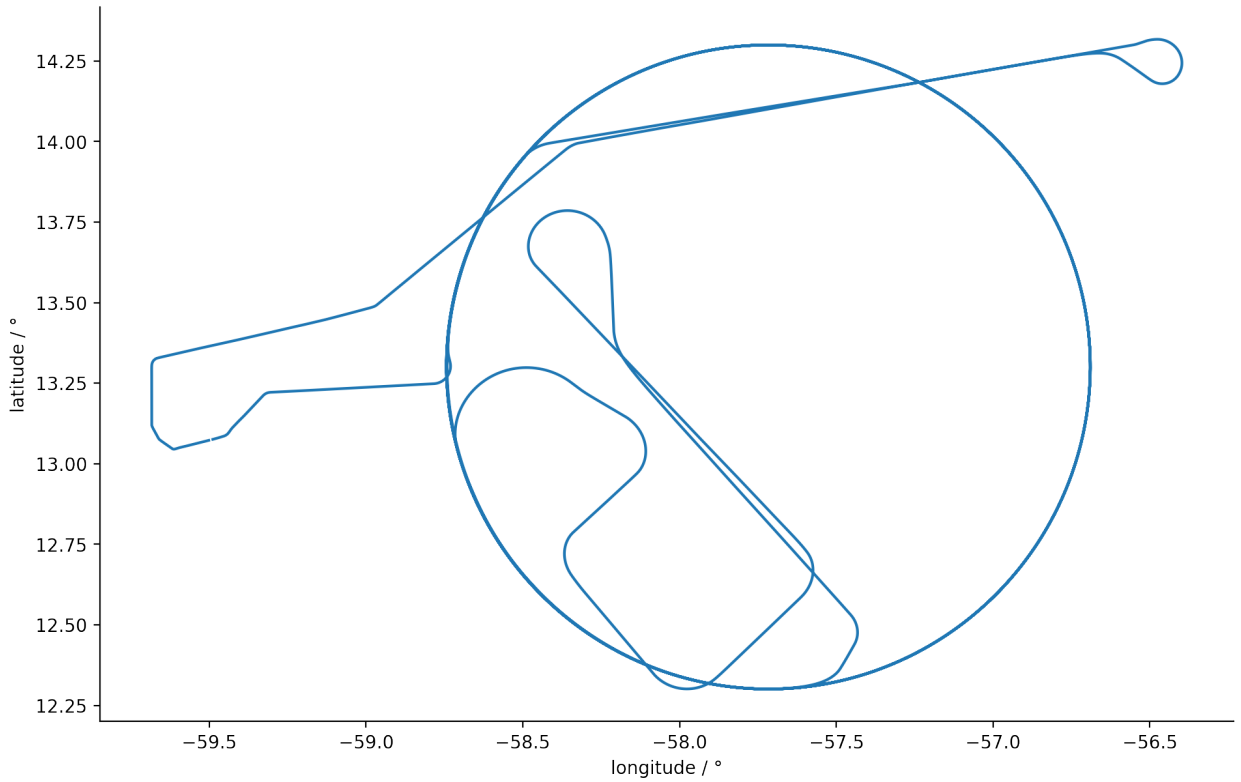
Just to have a better visual impression, we can create a quick overview plot of the data:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

plt.plot(ds.lon, ds.lat)
```

(continues on next page)

(continued from previous page)

```
center_lat = float(ds.lat.mean())
plt.gca().set_aspect(1./np.cos(np.deg2rad(center_lat)))
plt.xlabel("longitude / °")
plt.ylabel("latitude / °")
plt.show()
```




### Reducing the size of the dataset

Later on, we want to plot all flights on an interactive map. Currently the dataset is rather large as the aircraft location has been recorded continuously at a high data rate. While this is good for quantitative analysis, this leads to poor interactive performance. So before going further, it is a good idea to reduce the amount of data while keeping the visual impression.

A possible idea to reduce the amount of required data is that plotting a line already does linear interpolation between two coordinate points. So if we would remove those points which are close to the linear interpolation between its neighboring points, the visual impression will stay almost the same. This idea has already been stated by Ramer Douglas and Peucker and is illustrated at Wikipedia. While the algorithm is not hard to write, it is difficult to do it efficiently in python. Thus, I've skipped it and use the `simplification` library in stead.

---

**Note:** Many algorithms for shape processing are contained in the beautiful shapely library. In general, I'd recommend using that library, but it requies the GEOS library which is a bit tricky to install.

---

```
from simplification.cutil import simplify_coords_idx
def simplify_dataset(ds, tolerance):
    indices_to_take = simplify_coords_idx(np.stack([ds.lat.values, ds.lon.values],␣
↪axis=1), tolerance)
```

(continues on next page)

(continued from previous page)

```
    return ds.isel(time=indices_to_take)
```

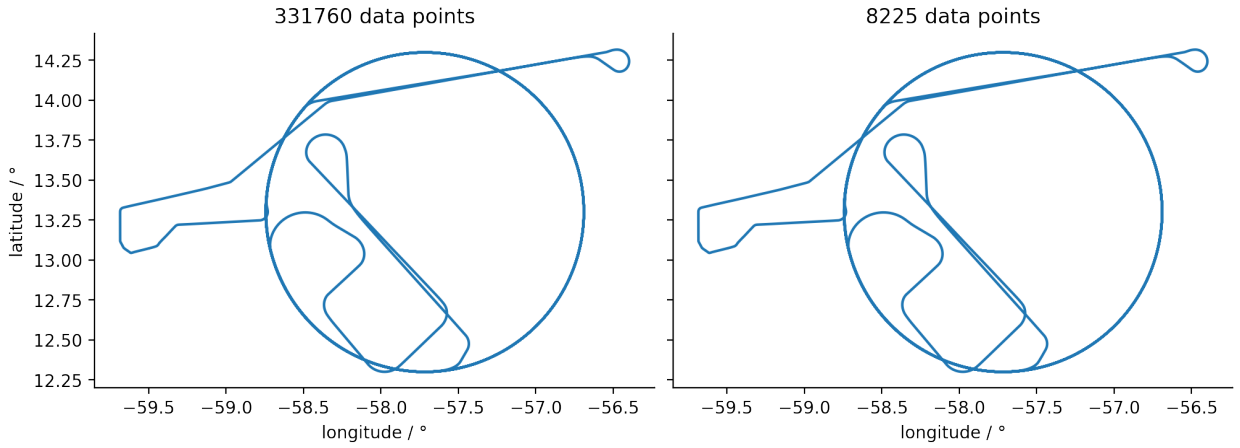
We can now use that algorithm to generate a simplified version of the dataset with a tolerance of $10^{-5}$ degrees, which otherwise looks the same to the previous version.

```
dssimplified = simplify_dataset(ds, 1e-5)
dssimplified
```

```
<xarray.Dataset>
Dimensions:     (time: 8225)
Coordinates:
  * time        (time) datetime64[ns] 2020-01-22T14:57:36 ... 2020-01-23T00:1...
    lat         (time) float64 13.08 13.08 13.08 13.08 ... 13.07 13.07 13.07
    lon         (time) float64 -59.49 -59.48 -59.48 ... -59.52 -59.51 -59.5
    alt         (time) float32 ...
Data variables:
    roll        (time) float32 ...
    pitch       (time) float32 ...
    heading     (time) float32 ...
    tas         (time) float32 ...
    trajectory  <U4 'HALO'
Attributes: (12/14)
    title:                             EUREC4A position and attitude data
    description:                       10Hz subset based on DLR BAHAMAS data, p...
    mission:                           EUREC4A
    platform:                          HALO
    instrument:                        BAHAMAS
    flight_id:                         HALO-0122
    ...                                ...
    featureType:                       trajectory
    Conventions:                       CF-1.7
    version:                           1.1
    history:                           acquired by HALO BAHAMAS, processed and ...
    file_created:                      File created by M. Klingebiel (email: ma...
    DODS_EXTRA.Unlimited_Dimension:  time
```

We can now compare those two tracks side by side while keeping a look at the number of data points required.

```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.plot(ds.lon, ds.lat)
ax1.set_title(f"{len(ds.time)} data points")
ax2.plot(dssimplified.lon, dssimplified.lat)
ax2.set_title(f"{len(dssimplified.time)} data points")

for ax in (ax1, ax2):
    ax.set_aspect(1./np.cos(np.deg2rad(center_lat)))
    ax.set_xlabel("longitude / °")
    ax.set_ylabel("latitude / °")
    ax.label_outer()


plt.show()

ratio = len(dssimplified.time) / len(ds.time)
print(f"compression ratio: {ratio*100:.2f} %")
```




```
compression ratio: 2.48 %
```

The dataset size has been substantially reduced while the visual impression stayed the same.

### 1.4.2 First interactive map

In order to show the map a little bit more interactively, we use the `ipyleaflet` library which creates a bridge between ipython and the Leaflet JavaScript library.

```
import ipyleaflet
```

As we will need to convert many tracks to ipyleaflet layers later on, the easiest is to create a little function for that purpose right away:

```
def track2layer(track, color="green", name=""):
    return ipyleaflet.Polyline(
        locations=np.stack([track.lat.values, track.lon.values], axis=1).tolist(),
        color=color,
        fill=False,
        weight=2,
        name=name
    )
```

With the help of that little function, creating a map is now like a breeze:

```
testmap = ipyleaflet.Map(center=(13.3, -57), zoom=7)
testmap.add_layer(track2layer(dssimplified))
display(testmap)
```

```
Map(center=[13.3, -57], controls=(ZoomControl(options=['position', 'zoom_in_text',
 ↪'zoom_in_title', 'zoom_out_…
```

### 1.4.3 All in one

Let's see if we can add all the flights and provide a layer switcher so that we can have a look at all flights individually. We'll start by loading and simplifying all out track data into a local dictionary. Requesting the datasets from the server is IO bound and thus can simply be accellerated using a ThreadPool:

```
from multiprocessing.pool import ThreadPool
pool = ThreadPool(20)

def get_dataset(flight_id):
    ds = cat.HALO.BAHAMAS.PositionAttitude[flight_id].to_dask()
    return flight_id, ds.load()

full_tracks = dict(pool.map(get_dataset, cat.HALO.BAHAMAS.PositionAttitude))
```

We still have to simplify the dataset, which is done here:

```
tracks = {flight_id: simplify_dataset(ds, 1e-5)
          for flight_id, ds in full_tracks.items()}
```

Let's also quickly grab some colors from a matplotlib colorbar, such that we can show all tracks in individual colors:

```
import matplotlib
colors = [matplotlib.colors.to_hex(c)
          for c in plt.cm.inferno(np.linspace(0, 1, len(tracks)))]
```

We can now start with a new empty map. Let's also have a different basemap.

```
m = ipyleaflet.Map(
    basemap=ipyleaflet.basemaps.Esri.NatGeoWorldMap,
    center=(13.3, -57), zoom=7
)
```

We'll add all the tracks as individual layers to the map

```
for (flight_id, track), color in zip(tracks.items(), colors):
    m.add_layer(track2layer(track, color, flight_id))
```

and add a scale, a legend, layer controls and a full screen button to the map and show it. If you want to zoom in, you can for example shift-click and drag a rectangle over the area you want to zoom in more closely.

```
m.add_control(ipyleaflet.ScaleControl(position='bottomleft'))
m.add_control(ipyleaflet.LegendControl(dict(zip(tracks, colors)),
                                       name="Flights",
                                       position="bottomright"))
m.add_control(ipyleaflet.LayersControl(position='topright'))
m.add_control(ipyleaflet.FullScreenControl())
display(m)
```

```
Map(center=[13.3, -57], controls=(ZoomControl(options=['position', 'zoom_in_text',
 ↪'zoom_in_title', 'zoom_out_…
```

## 1.5 SMART dataset

The following script exemplifies the access and usage of SMART data measured during EUREC4A. The Spectral Modular Airborne Radiation measurement sysTem (SMART) measures downward irradiances in the solar spectral range between 300 nm and 2200 nm.

More information on the dataset can be found in [SAB+19] and [WMSH01]. If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` or `author`.

### 1.5.1 Get data

- To load the data we first load the EUREC4A meta data catalogue. More information on the catalog can be found here.

```
import eurec4a
```

```
cat = eurec4a.get_intake_catalog()
list(cat.HALO.SMART)
```

```
['spectral_irradiances']
```

- We can further specify a product and a flight and obtain the dataset using `to_dask`.

**Note:** Have a look at the attributes of the xarray dataset `ds_smart` for all relevant information on the dataset, such as author, contact, or citation infromation.

```
ds_smart = cat.HALO.SMART.spectral_irradiances['HALO-0205'].to_dask()
ds_smart
```

```
<xarray.Dataset>
Dimensions:                (time: 32730)
Coordinates:
  * time                   (time) datetime64[ns] 2020-02-05T09:15:53 ... 2020-...
Data variables:
    alt                    (time) float32 ...
    lat                    (time) float32 ...
    lon                    (time) float32 ...
    sza                    (time) float32 ...
    saa                    (time) float32 ...
    F_down_solar_wl_422    (time) float32 ...
    F_down_solar_wl_532    (time) float32 ...
    F_down_solar_wl_648    (time) float32 ...
    F_down_solar_wl_858    (time) float32 ...
    F_down_solar_wl_1238   (time) float32 ...
    F_down_solar_wl_1638   (time) float32 ...
Attributes:
    Title:                 Spectral downward irradiance measured by SMART duri...
    campaign:              EUREC4A
    platform:              HALO
    Research_Flight_Day:   2020/20/05
    Version:               Revision 3 from 2021/01/24
```

(continues on next page)

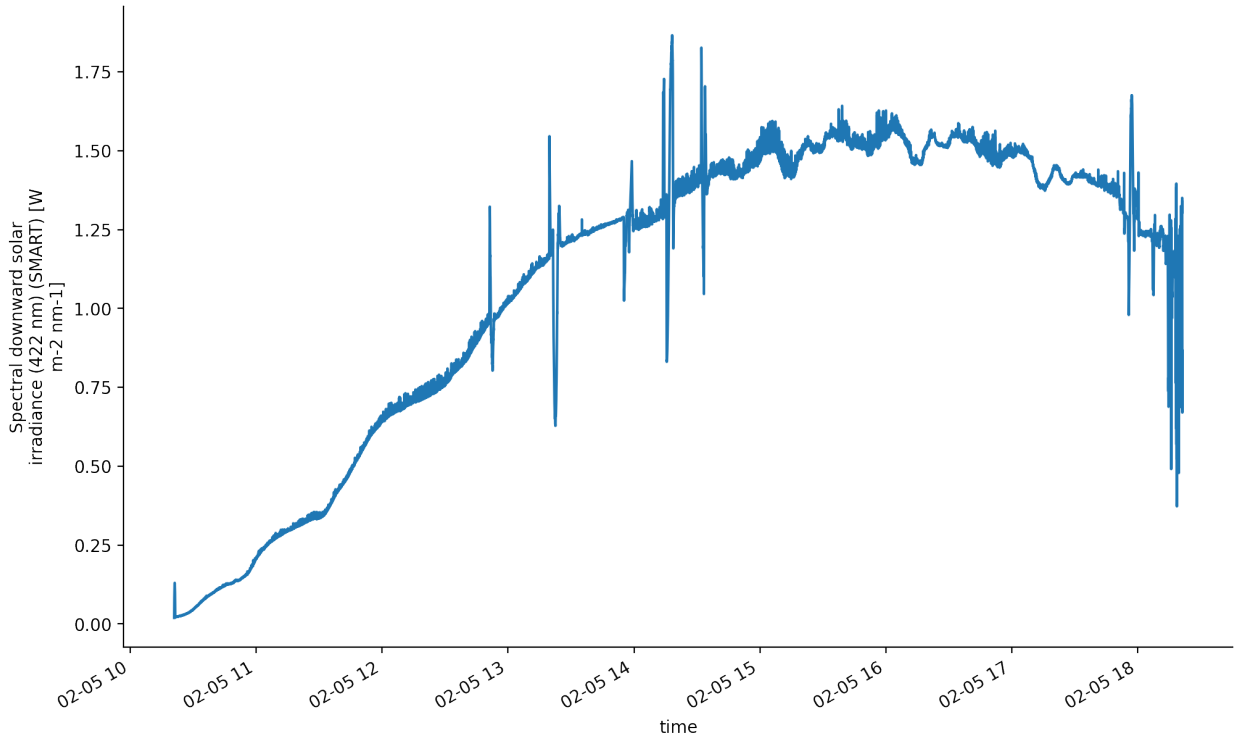
(continued from previous page)

```
    comment_1:              BAHAMAS data was processed by DLR, contact Andreas ...
    Contact:                Andre Ehrlich, University Leipzig, a.ehrlich@uni-le...
```

The available dataset includes irradiances for six selected wavelengths (422nm, 532nm, 648nm, 858nm, 1238nm, 1638nm). The full dataset is available on demand. Contact the dataset authors as stated in the dataset attributes `contact`.

First Quickplot of whole flight (one wavelength)

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

ds_smart.F_down_solar_wl_422.plot();
```




### 1.5.2 Load HALO flight phase information

All HALO flights were split up into flight phases or segments to allow for a precise selection in time and space of a circle or calibration pattern. For more information have a look at the respective github repository.

```
meta = eurec4a.get_flight_segments()
```

We select the flight phase we are interested in, e.g. the second circle on February 5 by it's segment_id.

```
segments = {s["segment_id"]: {**s, "flight_id": flight["flight_id"]}
            for platform in meta.values()
            for flight in platform.values()
            for s in flight["segments"]
```

(continues on next page)

(continued from previous page)

```
            }
seg = segments["HALO-0205_c2"]
```

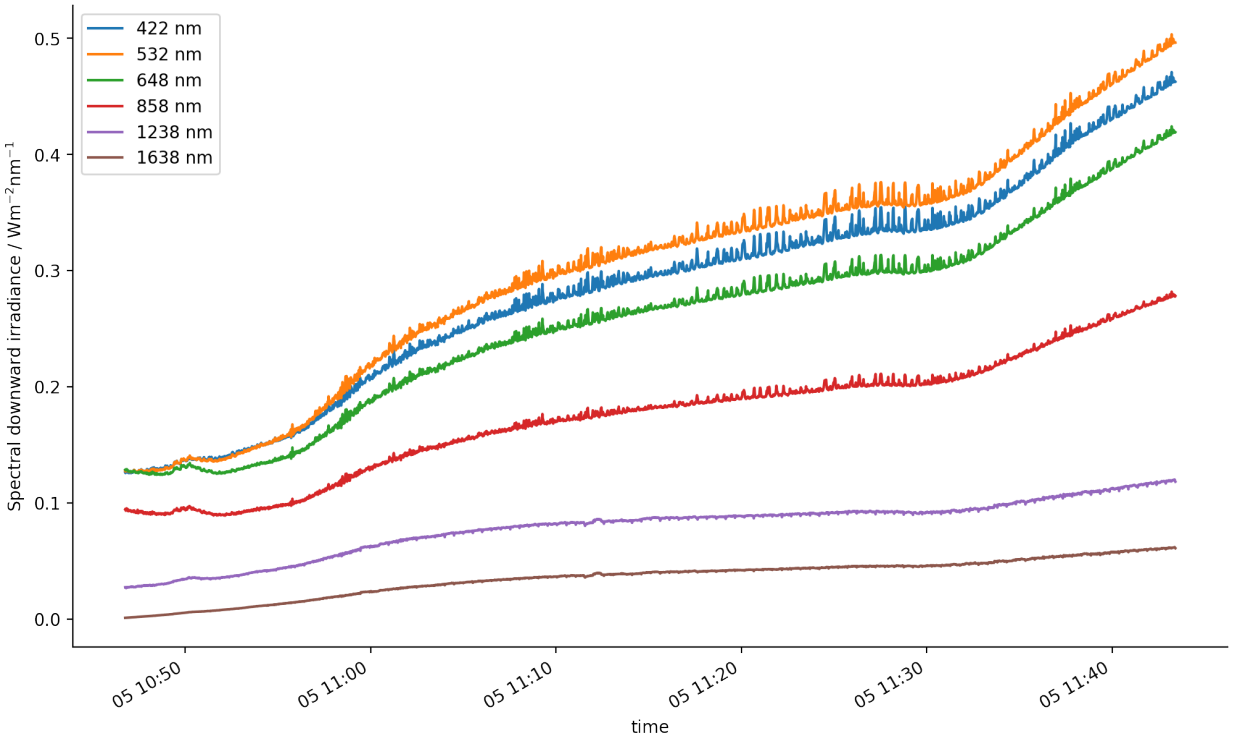
We transfer the information from our flight segment selection to our radar and radiometer data in the xarray dataset.

```
ds_smart_selection = ds_smart.sel(time=slice(seg["start"], seg["end"]))
```

### 1.5.3 Plots

We plot the spectral irradiances from different wavelengths measured with SMART during the selected flight segment.

```
fig, ax = plt.subplots()
wl_list=[422,532,648,858,1238,1638]
for i in wl_list:
    ds_smart_selection[f'F_down_solar_wl_{i}'].plot(label =f'{i} nm')
ax.legend()
ax.set_ylabel('Spectral downward irradiance / Wm$^{-2}$nm$^{-1}$')
None
```



## 1.6 VELOX

Video airbornE Longwave Observations within siX channels

```
import eurec4a
from IPython.display import Markdown

meta = eurec4a.get_meta()
Markdown(meta["VELOX"]["description"])
```

VELOX is a thermal infrared spectral imager (VELOX327k veL, 640 pixel by 512 pixels) with a synchronized filter wheel (at 100 Hz) covering six spectral channels in the thermal infrared wavelength range from 7.7 to 12.0 micrometer. The instrument measures the brightness temperature of upward radiance in a field-of-view of 35.49° by 28.71°.

The PI during EUREC4A was Michael Schäfer (University Leipzig).

If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` or `author`.

**Note:** No data available for the transfer flights (first and last) and the first local research flight.

### 1.6.1 Get data

To load the data we first load the EUREC4A intake catalog and list the available datasets from VELOX.Currently, there is a cloud mask product available.

```
cat = eurec4a.get_intake_catalog()
list(cat.HALO.VELOX)
```

```
['cloudmask']
```

```
ds = cat.HALO.VELOX.cloudmask["HALO-0205"].to_dask()
ds
```

```
<xarray.Dataset>
Dimensions:     (time: 29894, x: 640, y: 512)
Coordinates:
  * time        (time) datetime64[ns] 2020-02-05T09:21:46 ... 2020-02-05T18:0...
    lat         (time) float32 ...
    lon         (time) float32 ...
    alt         (time) float32 ...
Dimensions without coordinates: x, y
Data variables:
    vza         (y, x) float32 ...
    vaa         (y, x) float32 ...
    cloud_mask  (y, x, time) int8 ...
    CF_min      (time) float32 ...
    CF_max      (time) float32 ...
Attributes: (12/22)
    title:                Two-dimensional cloud mask and cloud fraction with ...
    research_flight_day:  20200205
    version:              Version 3 from 2021-02-12
    comment_1:            The cloud mask is derived with 1 Hz temporal resolu...
```

(continues on next page)

(continued from previous page)

```
    comment_2:              Four different thresholds (0.5 K, 1.0 K, 1.5 K, and...
    comment_3:              The final cloud mask logically combines the differe...
    ...                     ...
    source:                 Airborne imaging with the VELOX system
    institution:            University of Leipzig, Leipzig Institute for Meteor...
    author:                 Michael Schäfer, André Ehrlich, Anna Luebke, Jakob ...
    history:                2021-02-06 flag_values, flag_meanings, and comments...
    created_on:             2021-02-12
    Conventions:            "CF-1.8"
```

### 1.6.2 What does the VELOX image look like at a specific dropsonde launch?

We can use the flight segmentation information to select a dropsonde and further extract the corresponding VELOX image taken at the time of the dropsonde launch.In particular, we select the first dropsonde with the quality flag `GOOD` from the second `circle` on February 5.

#### Get dropsonde ID

(0) we read the meta data and extract the segment IDs

```
meta = eurec4a.get_flight_segments()
```

```
segments = [{**s,
             "platform_id": platform_id,
             "flight_id": flight_id
            }
            for platform_id, flights in meta.items()
            for flight_id, flight in flights.items()
            for s in flight["segments"]
           ]
```

(1) we extract the segment ID of the second `circle`

```
import datetime

segments_ordered_by_start_time = list(sorted(segments, key=lambda s: s["start"]))
circles_Feb05 = [s["segment_id"]
                 for s in segments_ordered_by_start_time
                 if "circle" in s["kinds"]
                 and s["start"].date() == datetime.date(2020,2,5)
                 and s["platform_id"] == "HALO"
                ]
second_circle_Feb05 = circles_Feb05[1]
second_circle_Feb05
```

```
'HALO-0205_c2'
```

(2) we extract the ID of the first dropsonde with flag `GOOD` in this circle

```
segments_by_segment_id = {s["segment_id"]: s for s in segments}
first_dropsonde = segments_by_segment_id[second_circle_Feb05]["dropsondes"]["GOOD"][0]
first_dropsonde
```

```
'HALO-0205_s13'
```

### What is the corresponding launch time of the selected sonde?

So far, we only made use of the flight segmentation meta data. The launch time to a given sonde is stated in the JOANNE dataset (see also book chapter *Dropsondes dataset JOANNE*). We use again the intake catalog to load the JOANNE dataset and extract the launch time to the selected dropsonde by it's sonde ID.

```
dropsondes = cat.dropsondes.JOANNE.level3.to_dask()
```

```
sonde_dt = dropsondes.swap_dims({"sounding": "sonde_id"}).sel(sonde_id=first_
↪dropsonde).launch_time.values
sonde_dt
```

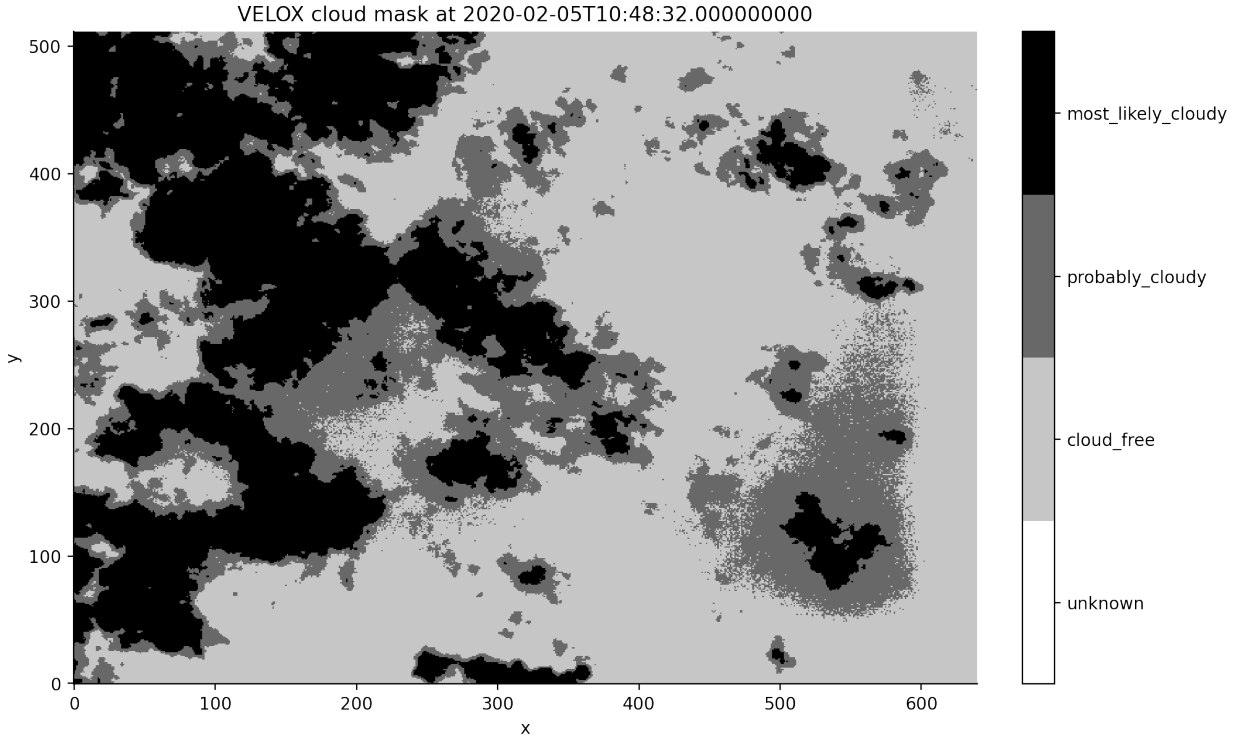
```
numpy.datetime64('2020-02-05T10:48:32.000000000')
```

### Cloud mask plot

Finally, we plot the VELOX image closest in time to the dropsonde launch.The y-axis is in flight direction and the image consists of 640 pixel by 512 pixels. The corresponding pixel viewing sensor zenith and azimuth angles are given by the dataset variables `vza` and `vaa`.

```
ds_sel = ds.cloud_mask.sel(time=sonde_dt, method="nearest")
```

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

fig, ax = plt.subplots()

cax = ds_sel.plot(ax=ax,
                  cmap=plt.get_cmap('Greys', lut=4),
                  vmin=-1.5, vmax=2.5,
                  add_colorbar=False
                  )
cbar = fig.colorbar(cax, ticks=ds.cloud_mask.flag_values)
cbar.ax.set_yticklabels(ds.cloud_mask.flag_meanings.split(" "));

ax.set_title(f"VELOX cloud mask at {sonde_dt}");
```




What are the image cloud fraction lower and upper bounds?

```
cfmin = ds.CF_min.sel(time=sonde_dt, method="nearest").values
cfmax = ds.CF_max.sel(time=sonde_dt, method="nearest").values
print(f"Image minimum cloud fraction (most likely cloudy): {cfmin:.2f}")
print(f"Image maximum cloud fraction (most likely and probably cloudy) :{cfmax:.2f}")
```

```
Image minimum cloud fraction (most likely cloudy): 0.24
Image maximum cloud fraction (most likely and probably cloudy) :0.47
```

### 1.6.3 Cloud fraction time series from the second circle on February 5

We also want to show a time series of cloud fraction and use the segment ID from the second circle to extract and save the segment information to the variable `seg`. We can later use the segments start and end times to select the data in time.

```
seg = segments_by_segment_id[second_circle_Feb05]
```

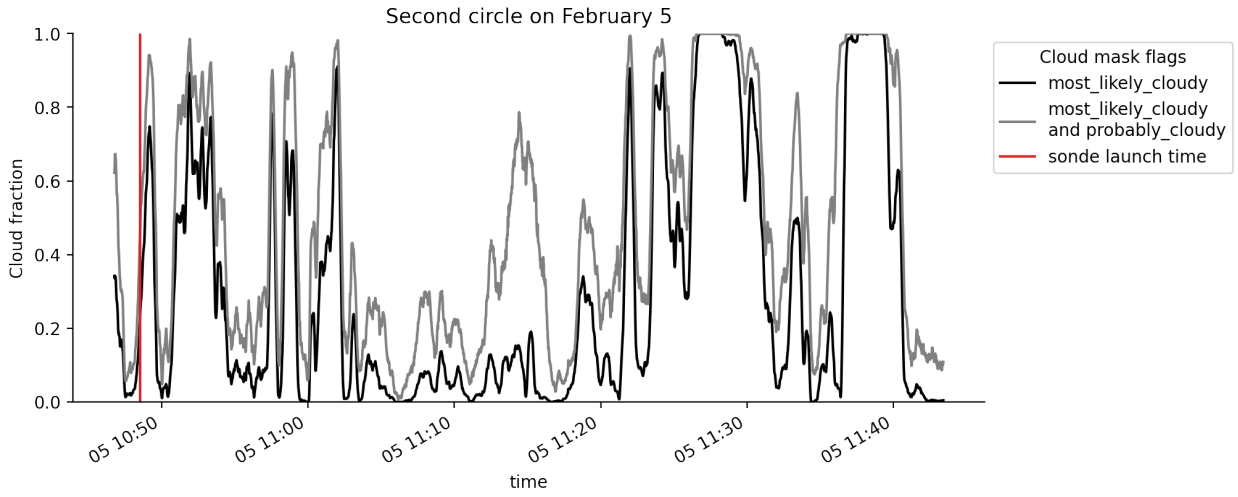
The dataset variable `CF_min` provides a lower bound to cloud fraction estimates based on the cloud mask flag `most_likely_cloudy`, while `CF_max` provides an upper bound by including the uncertain pixels labeled `probably cloudy`.

```
selection = ds.sel(time=slice(seg["start"], seg["end"]))

with plt.style.context("mplstyle/wide"):
    fig, ax = plt.subplots()
    selection.CF_min.plot(color="k", label="most_likely_cloudy")
    selection.CF_max.plot(color="grey", label="most_likely_cloudy\nand probably_cloudy
↪")
    ax.axvline(sonde_dt, color="C3", label="sonde launch time")
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylim(0, 1)
    ax.set_ylabel("Cloud fraction")
    ax.set_title("Second circle on February 5")
    ax.legend(title="Cloud mask flags", bbox_to_anchor=(1,1), loc="upper left");
```




## 1.7 BACARDI dataset

The following script exemplifies the access and usage of the Broadband AirCrAft RaDiometer Instrumentation (BACARDI), that combines two sets of Kipp and Zonen broadband radiometer measuring upward and downward irradiance in the solar (pyranometer model CMP-22, 0.2 - 3.6 µm) and terrestrial (pyrgeometer model CGR-4, 4.5 - 42 µm) spectral range.

The dataset is published under Ehrlich et al. (2021). If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` or `author`.

### 1.7.1 Get data

- To load the data we first load the EUREC4A meta data catalogue. More information on the catalog can be found here.

```
import eurec4a
```

```
cat = eurec4a.get_intake_catalog()
list(cat.HALO.BACARDI)
```

```
['irradiances']
```

- We can further specify the platform, instrument, if applicable dataset level or variable name, and pass it on to dask.

**Note:** Have a look at the attributes of the xarray dataset `ds` for all relevant information on the dataset, such as author, contact, or citation infromation.

```
ds = cat.HALO.BACARDI.irradiances['HALO-0205'].to_dask()
ds
```

```
<xarray.Dataset>
Dimensions:              (time: 327320)
Coordinates:
  * time                 (time) datetime64[ns] 2020-02-05T09:15:52 ... 2020-02...
Data variables:
    alt                  (time) float32 ...
    lat                  (time) float32 ...
    lon                  (time) float32 ...
    sza                  (time) float32 ...
    saa                  (time) float32 ...
    F_down_solar         (time) float32 ...
    F_down_solar_diff    (time) float32 ...
    F_up_solar           (time) float32 ...
    F_down_terrestrial   (time) float32 ...
    F_up_terrestrial     (time) float32 ...
    F_down_solar_sim     (time) float32 ...
    cloud_mask           (time) int8 ...
Attributes:
    title:                 Broadband radiation mesured by BACARDI during EUREC4A
    campaign:              EUREC4A
    platform:              HALO
    instrument:            Broadband AirCrAft RaDiometer Instrumentation (BACA...
    research_flight_day:   2020/02/05
    version:               Revision 1 from 2021/02/04
    comment_1:             BACARDI Raw data was processed by DLR, contact Andr...
    contact:               Andre Ehrlich, University Leipzig, a.ehrlich@uni-le...
```

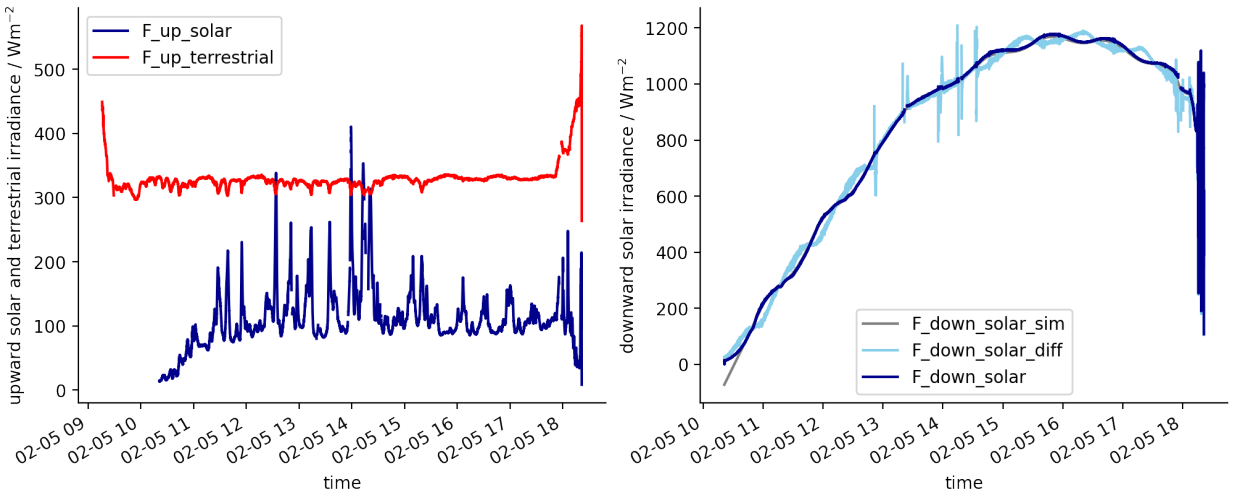
The data from EUREC4A is of 10 Hz measurement frequency and corrected for dynamic temperature effects. For the downward solar irradiance, data are provided with and without aircraft attitude correction corresponding to cloud-free and cloudy conditions, respectively, above HALO.

### 1.7.2 Plots

We plot the upward and downward irradiances in two panels.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book", "./mplstyle/wide"])
```

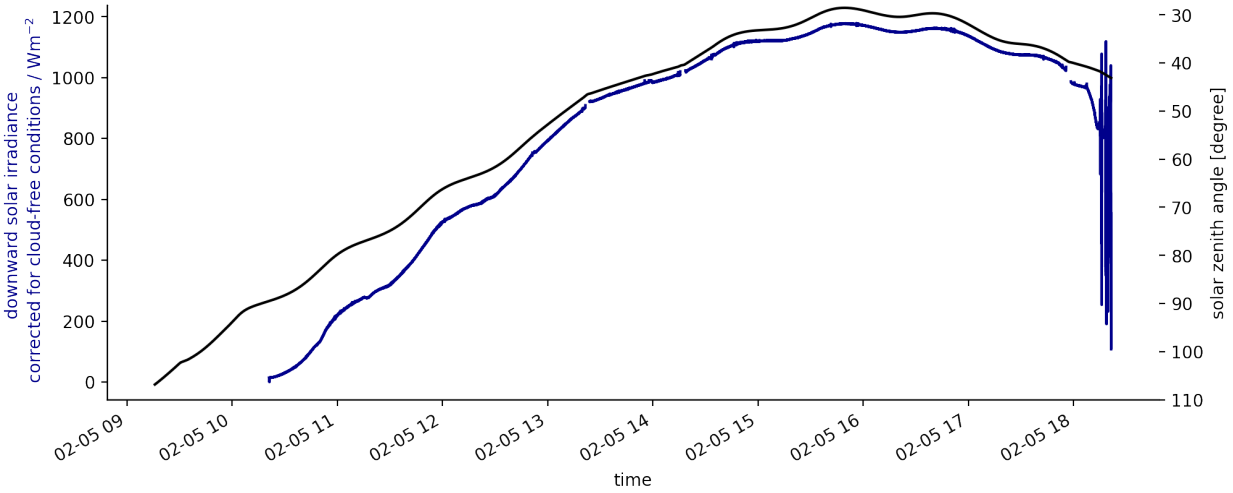
```
fig, (ax1, ax2) = plt.subplots(1, 2)
ax1.set_prop_cycle(color=['darkblue', 'red'])
for var in ['F_up_solar', 'F_up_terrestrial']:
    ds[var].plot(ax=ax1,label= var)
ax1.legend()
ax1.set_ylabel('upward solar and terrestrial irradiance / Wm$^{-2}$')

ax2.set_prop_cycle(color=['grey', 'skyblue', 'darkblue'])
for var in ['F_down_solar_sim', 'F_down_solar_diff', 'F_down_solar']:
    ds[var].plot(ax=ax2, label= var)
ax2.legend()
ax2.set_ylabel('downward solar irradiance / Wm$^{-2}$');
```




The attitude correction of downward solar irradiance does not account for the present cloud situation above HALO. Instead, two data sets, one assuming cloud-free and one assuming overcast (diffuse illumination) conditions, are provided. Depending on the application, the user needs to choose between both data sets. For the downward solar irradiance assuming cloud-free conditions, the data are filtered for turns of HALO, high roll and pitch angles. This filter is not applied for the data assuming overcast/diffuse conditions to provide the full data. However, data during turns of HALO need to be analysed with care. As shown in the example some artifical spikes due to turns are present in the data.

The wiggles originate from the about 200km change in location and therewith solar zenith angle within one circle / hour as can be seen in a plot.

```
fig, ax = plt.subplots()
ds.F_down_solar.plot(ax=ax, color='darkblue')
ax.set_ylabel('downward solar irradiance \n corrected for cloud-free conditions / Wm$^
↪{-2}$',
              color='darkblue')
ax2 = ax.twinx()
ds.sza.plot(ax=ax2, color='black')
ax2.set_ylim(110, 28);
```



## 1.8 WALES Lidar

The water vapour differential absorption lidar WALES. WALES operates at four wave-lengths near 935 nm to measure water-vapor mixing ratio profiles covering the whole atmosphere below the aircraft. The system also contains additional aerosol channels at 532 nm and 1064 nm with depolarization. WALES uses a high-spectral resolution technique, which distinguishes molecular from particle backscatter.

At typical flight speeds of 200 m/s the backscatter product from the HSRL has a resolution of 200m in the horizontal and 15m in the vertical, while the water vapor product has approximately 3km horizontal and 250m vertical. The PIs during EUREC4A were Martin Wirth and Heike Gross (DLR).

More information on the instrument can be found in Wirth et al., 2009. If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` or `author`.

---

**Note:** Due to safety regulations the Lidar can only be operated above 6 km which leads to data gaps in about the first and last 30 minutes of each flight.

---

```
import eurec4a
import xarray as xr
```

### 1.8.1 Get data

To load the data we first load the EUREC4A meta data catalog and list the available datasets from WALES.

More information on the catalog can be found here.

```
cat = eurec4a.get_intake_catalog()
print(cat.HALO.WALES.cloudparameter.description)
```

```
WALES-Lidar cloud top height, cloud optical thickness and cloud flag data.
```

### 1.8.2 Cloud parameter

```
ds_cloud = cat.HALO.WALES.cloudparameter["HALO-0205"].to_dask()
ds_cloud
```

```
<xarray.Dataset>
Dimensions:      (time: 121200)
Coordinates:
  * time         (time) datetime64[ns] 2020-02-05T09:34:00.167000064 ... 2020-...
    lat          (time) float32 ...
    lon          (time) float32 ...
Data variables:
    cloud_mask   (time) float32 ...
    cloud_top    (time) float32 ...
    cloud_ot     (time) float32 ...
    pbl_top      (time) float32 ...
    target_lat   (time) float64 ...
    target_lon   (time) float64 ...
Attributes: (12/24)
```

(continues on next page)

(continued from previous page)

```
    convention:             CF-1.8
    location_name:          HALO
    platform:               HALO
    instrument:             WALES
    system:                 WALES H2O-DIAL
    title:                  WALES lidar cloud mask
    ...                     ...
    ongoing_subset:         10
    featureType:            trajectory
    author:                 Martin Wirth
    contact:                martin.wirth@dlr.de
    source:                 airborne observation
    history:                2021-03-09 17:02:35 Generated by generate_cloudmas...
```

We select 1min of flight `time`. You can freely change the times or put `None` instead of the slice start and/or end to get up to the full flight data. We explicitly `load()` the dataset at this stage as want to use this subset multiple times in the following.

**Note:** `load()` should always be called late and in particular after subsetting the data to prevent loading more than necessary.

```
ds_cloud_sel = ds_cloud.sel(time=slice("2020-02-05T13:06:30", "2020-02-05T13:07:030
↪")).load()
```

In order to work with the different cloud mask flags, we extract the meanings into a dictionary, which we can later use to select relevant data:

```
cm_meanings = dict(zip(ds_cloud_sel.cloud_mask.flag_meanings.split(" "),
                       ds_cloud_sel.cloud_mask.flag_values))
cm_meanings
```

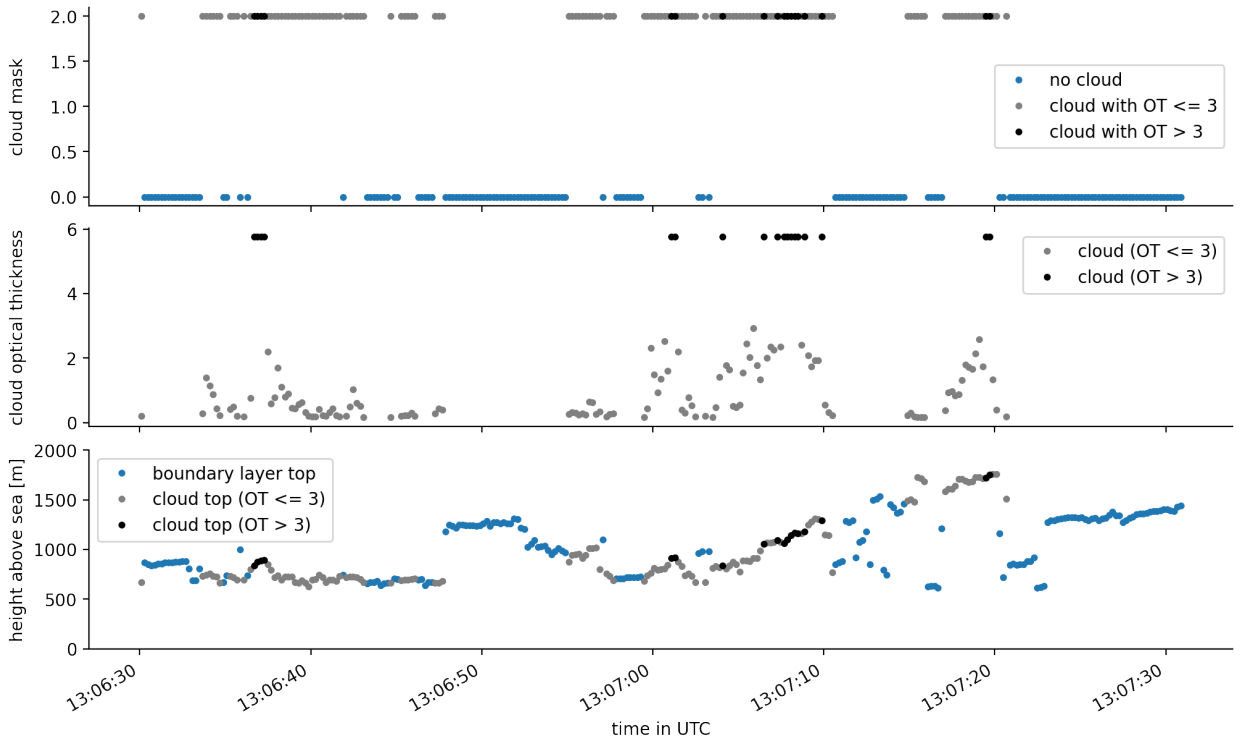
```
{'cloud_free': 0, 'probably_cloudy': 1, 'most_likely_cloudy': 2}
```

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

fig, axes = plt.subplots(3, 1, sharex=True)
ax1, ax2, ax3 = axes

cloud_free = ds_cloud_sel.cloud_mask[ds_cloud_sel.cloud_mask == cm_meanings["cloud_
↪free"]]
cloud_free.plot(ax=ax1, x="time", ls="", marker=".", color="C0", label="no cloud")

thin_cloud = ds_cloud_sel.cloud_mask[ds_cloud_sel.cloud_ot<=3]
thin_cloud.plot(ax=ax1, x="time", ls="", marker=".", color="grey", label="cloud with␣
↪OT <= 3")

thick_cloud = ds_cloud_sel.cloud_mask[ds_cloud_sel.cloud_ot>3]
thick_cloud.plot(ax=ax1, x="time", ls="", marker=".", color="k", label="cloud with OT␣
↪> 3")

ax1.set_ylabel(f"{ds_cloud_sel.cloud_mask.long_name}")
```

(continues on next page)

(continued from previous page)

```
ot_of_thin_cloud = ds_cloud_sel.cloud_ot[ds_cloud_sel.cloud_ot<=3]
ot_of_thin_cloud.plot(ax=ax2, x="time", ls="", marker=".", color="grey", label="cloud␣
↪(OT <= 3)")

ot_of_thick_cloud = ds_cloud_sel.cloud_ot[ds_cloud_sel.cloud_ot>3]
ot_of_thick_cloud.plot(ax=ax2, x="time", ls="", marker=".", color="k", label="cloud␣
↪(OT > 3)")

ax2.set_ylabel(f"{ds_cloud_sel.cloud_ot.long_name}")


pbl_top = ds_cloud_sel.pbl_top
pbl_top.plot(ax=ax3, x="time", ls="", marker=".", color="C0", label="boundary layer␣
↪top")

thin_cloud_top = ds_cloud_sel.cloud_top[ds_cloud_sel.cloud_ot<=3]
thin_cloud_top.plot(ax=ax3, x="time", ls="", marker=".", color="grey", label="cloud␣
↪top (OT <= 3)")

thick_cloud_top = ds_cloud_sel.cloud_top[ds_cloud_sel.cloud_ot>3]
thick_cloud_top.plot(ax=ax3, x="time", ls="", marker=".", color="k", label="cloud top␣
↪(OT > 3)")

ax3.set_ylim(0, 2000)
ax3.set_ylabel("height above sea [m]")
ax3.set_xlabel('time in UTC')

for ax in axes:
    ax.label_outer()
    ax.legend()

fig.align_ylabels()
None
```




### 1.8.3 Cloud fraction

The cloud fraction can be estimated based on how often the instrument detected a cloud versus how often it measured anything. So, in order to compute the cloud fraction correctly, we must take care of missing values, in the original dataset. As Python has no commonly used way of carrying on a missing value through comparisions, we must keep track of it ourselves.

---

**Note:** In contrary, the R language for example knows about the special value `NA` which is carried along through compatisons, such that the result of the comparison `NA == 3` continues to be `NA`.

In Python and in particular using `xarray`, missing data is expressed as `np.nan`, and due to floating point rules `np.nan == 3` evaluates to `False`. As this `False` value is indistinguishable from a value which didn't match out flag (`3` in this case), averaging over the result would lead to a wrong cloud fraction result.

---

To keep track of the missing data, we'll use `DataArray.where` on our boolean results to convert the results back to floating-point numbers (`0` and `1`) and fill missing values back in as `np.nan`. Later on, methods like `.sum()` and `.mean()` will automatically skip the `np.nan` values such that our results will be correct. To finally compute the cloud fraction, we need some form of binary cloud mask, which contains values of either 0 or one. To do so, we have two options, depending on whether we want to include `'probably_cloudy'` or not. These two options are denoted by `min_` (without probably cloudy) and `max_` (with probably cloudy) prefixes.

```
cloudy_flags = xr.DataArray(
    [cm_meanings['probably_cloudy'], cm_meanings['most_likely_cloudy']],
    dims=("flags",))

min_cloud_binary_mask = (ds_cloud.cloud_mask == cm_meanings['most_likely_cloudy']).
↪where(ds_cloud.cloud_mask.notnull())
max_cloud_binary_mask = (ds_cloud.cloud_mask == cloudy_flags).any("flags").where(ds_
↪cloud.cloud_mask.notnull())
```

(continues on next page)

(continued from previous page)

```
cloud_ot_gt3 = (ds_cloud.cloud_ot > 3).where(ds_cloud.cloud_ot.notnull())

print(f"Minimum cloud fraction on Feb 5: {min_cloud_binary_mask.mean().values * 100:.
↪2f} %")
print(f"Total cloud fraction on Feb 5: {max_cloud_binary_mask.mean().values * 100:.2f}
↪ %")
print(f"Fraction of clouds with optical thickness greater than 3: {cloud_ot_gt3.
↪mean().values * 100:.2f} %")
```

```
Minimum cloud fraction on Feb 5: 40.23 %
Total cloud fraction on Feb 5: 40.23 %
Fraction of clouds with optical thickness greater than 3: 28.25 %
```

As it turns out, in this section of the flight, the instrument is very certain about the cloudiness. Yet, this missing value story was a bit complicated, so let's check if we would get something different if we would do it the naive way:

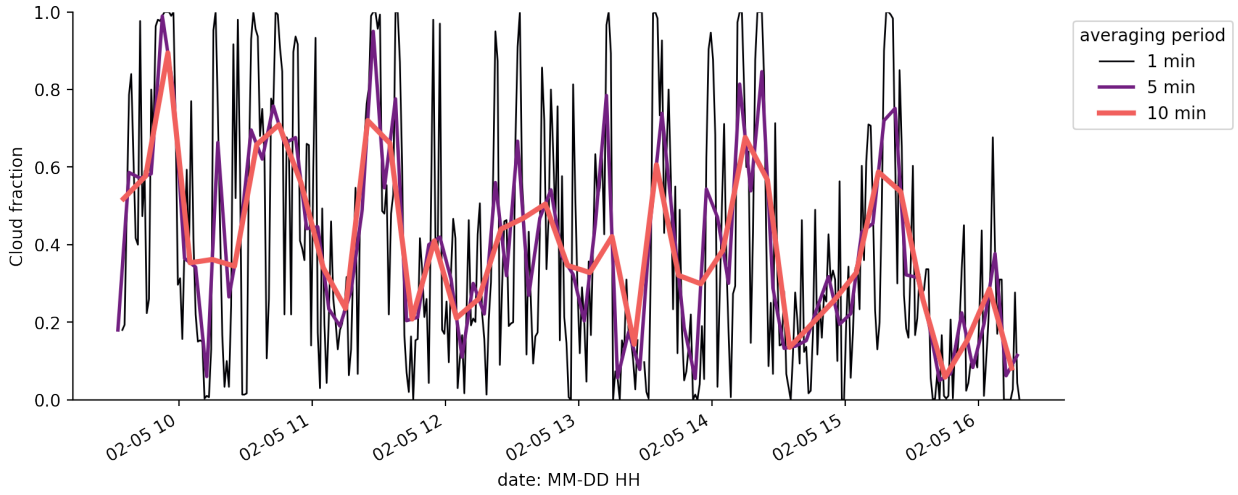
```
cf_wrong = (ds_cloud.cloud_mask == cloudy_flags).any("flags").mean().values
print(f"wrong cloud fraction: {cf_wrong * 100:.2f} %")
```

```
wrong cloud fraction: 39.93 %
```

Indeed, this is different… Good that we have thought about it.

We can now use a time averaging window to derive a time dependent cloud fraction from the `cloud_mask` variable and see how it varies over the course of the flight.

```
with plt.style.context("mplstyle/wide"):
    fig, ax = plt.subplots()

    ax.set_prop_cycle(color=plt.get_cmap("magma")(np.linspace(0, 1, 4)))
    for ind, t in enumerate([1, 5, 10]):
        averaged_cloud_fraction = min_cloud_binary_mask.resample(time=f"{t}min",␣
↪loffset=f"{t/2}min").mean()
        averaged_cloud_fraction.plot(lw=ind + 1, label=f"{t} min")

    ax.set_ylim(0, 1)
    ax.set_ylabel("Cloud fraction")
    ax.set_xlabel("date: MM-DD HH")
    ax.legend(title="averaging period", bbox_to_anchor=(1,1), loc="upper left")
    None
```




## 1.9 HAMP comparison

This notebook shows a comparison of cloud features as detected by different parts of the Halo Microwave Package (HAMP) and includes data from WALES and specMACS for additional context. Another application of the HAMP data is included in the chapter *HALO UNIFIED dataset*.

```
import numpy as np
import eurec4a
cat = eurec4a.get_intake_catalog()
```

We'll define a helper-function to approximately convert cell centers to edges:

And define a multi-linear scale that we will use for the LWP plot later on. This scale allows us to use zoom in a certain y-range. In our case, we will later increase the scale from 20 to 20 by a factor of 10.

### 1.9.1 Collect datasets

We want to look at data from several sensors within a spacific timeframe on flight `HALO-0205`:

```
flight_id = 'HALO-0205'
timeslice = slice('2020-02-05T13:07', '2020-02-05T13:11')

radiometer_cm = cat.HALO.UNIFIED.HAMPradiometer_cloudmask[flight_id].to_dask().
 ↪sel(time=timeslice)
radar_cm = cat.HALO.UNIFIED.HAMPradar_cloudmask[flight_id].to_dask().
 ↪sel(time=timeslice)

retrieval =  cat.HALO.UNIFIED.HAMPradiometer_retrievals[flight_id].to_dask().
 ↪sel(time=timeslice)

radar = cat.HALO.UNIFIED.HAMPradar[flight_id].to_dask().sel(time=timeslice)
wales = cat.HALO.WALES.cloudparameter[flight_id].to_dask().sel(time=timeslice)
specMACS = cat.HALO.specMACS.cloudmaskSWIR[flight_id].to_dask().sel(time=timeslice)
```

Ideally, we'd be handling the cloud flag meanings properly, but for now, let's just check if they are defined similarly:

```
assert np.all(radar_cm.cloud_mask.flag_values == radiometer_cm.cloud_mask.flag_values)
assert np.all(radar_cm.cloud_mask.flag_values == wales.cloud_mask.flag_values)
assert radar_cm.cloud_mask.flag_meanings == 'no_cloud_detectable probably_cloudy most_
 ↪likely_cloudy'
assert radiometer_cm.cloud_mask.flag_meanings == 'no_cloud_detectable probably_cloudy␣
 ↪most_likely_cloudy'
assert wales.cloud_mask.flag_meanings == 'cloud_free probably_cloudy most_likely_
 ↪cloudy'
```

Let's prepare the radar reflectivity (dBZ) time-height plot. Get the dBZ in right shape and adjust the x and y axis according to the requirements of pcolormesh, this means x and y define the pixel edges in the respective direction. Also we convert the height coordinate from the WGS84 ellipsoid height to height above the Geoid. For this, we need y (and x) as 2D fields, as we have to correct the height in every time step as function of lat and lon. The Radar and Lidar cloud top heigh products are already defined above the Geoid or sea surface.

```
def wgs84_height(lon, lat):
    #TODO: find wgs84_height(lon, lat) function that works with the licence
    # this is a good average number for the EUREC4A circle area.
    return np.zeros_like(lon+lat) - 47.5

radar_dBZ = radar.dBZ.transpose('height', 'time').values
radar_x = center_to_edge(radar.time)
radar_y = center_to_edge(radar.height)
radar_x, radar_y = np.meshgrid(radar_x, radar_y)
wgs_correction = center_to_edge(wgs84_height(radar.lon, radar.lat))[np.newaxis, :]
radar_y = radar_y - wgs_correction
```

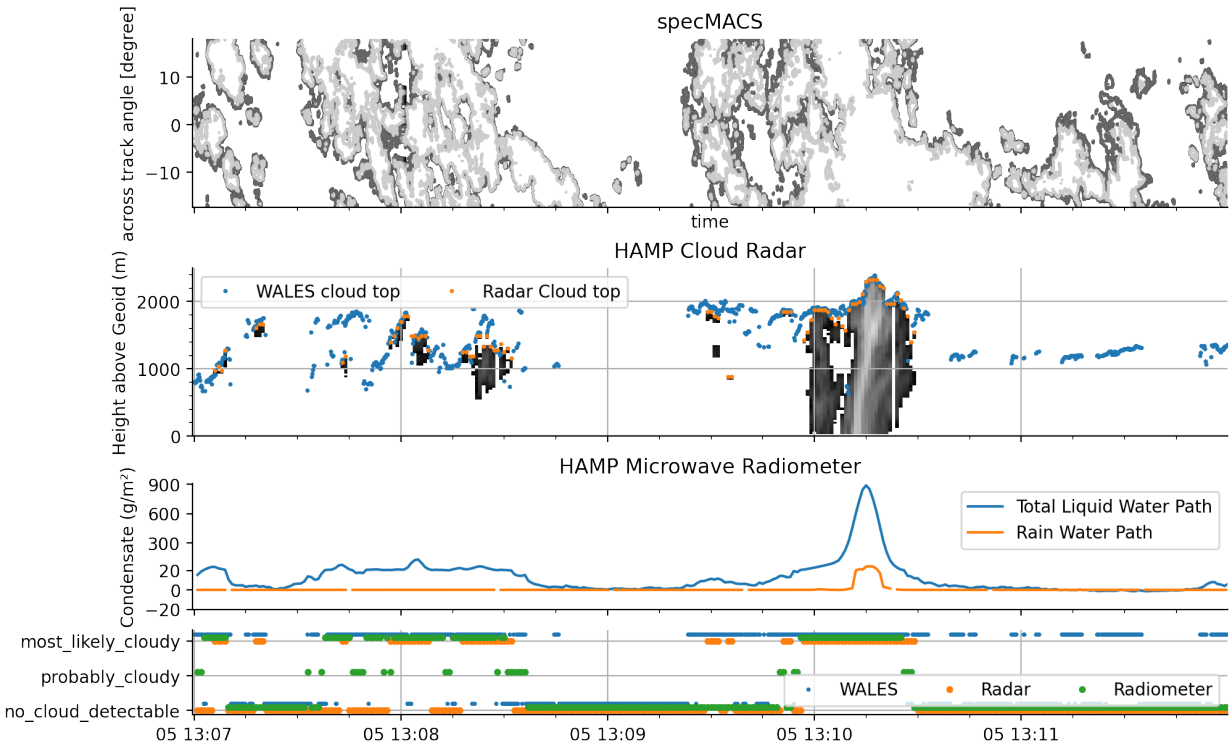
### 1.9.2 Plot timeseries

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

fig, (ax3, ax2, ax1, ax,) = plt.subplots(
    nrows=4, sharex=True,
    gridspec_kw=dict(height_ratios=[1, 1, .75, 0.5]),
)

ax.plot(wales.time, wales.cloud_mask + 0.2, '.', label='WALES', markersize=3)
ax.plot(radar_cm.time, radar_cm.cloud_mask, '.', label='Radar')
ax.plot(radiometer_cm.time, radiometer_cm.cloud_mask + 0.1, '.', label='Radiometer')

# Make nice labels
ax.set_yticks(radar_cm.cloud_mask.flag_values)
ax.set_yticklabels(radar_cm.cloud_mask.flag_meanings.split())
ax.legend(ncol=3, loc='lower right')
ax.grid()

# Plot Radar Curtain
ax2.set_title('HAMP Cloud Radar')
ax2.pcolormesh(radar_x, radar_y, radar_dBZ, vmin=-30, vmax=35, cmap='gray',␣
 ↪rasterized=True)

ax2.plot(
     wales.time,  wales.cloud_top,
```

(continues on next page)

(continued from previous page)

```
    '.', markersize=3,
    label='WALES cloud top'
)
ax2.plot(
    radar_cm.time, radar_cm.cloud_top,
    '.', markersize=3,
    label='Radar Cloud top'
)
ax2.set_ylabel('Height above Geoid (m)')
ax2.set_ylim(0, 2500)
ax2.grid()
ax2.legend(ncol=3)
ax2.minorticks_on()

# specMACS
specMACS.cloud_mask.T.plot.contour(ax=ax3, cmap="gray", add_colorbar=False)
ax3.set_title('specMACS')

# Plot MWR Retrievals
ax1.set_title('HAMP Microwave Radiometer')
ax1.plot(retrieval.time, retrieval.lwp*1000, label='Total Liquid Water Path')
ax1.plot(retrieval.time, retrieval.rwp*1000, label='Rain Water Path')
ax1.set_ylim(-10, 911)
ax1.legend()

ax1.set_yscale('multilinear', segments=[[-20., 20., 10.]])
#ax1.axhline(20, color='k', linewidth=1, linestyle=':', zorder=0)
ax1.set_yticks([-20, 0, 20, 300, 600, 900])
ax1.set_ylabel('Condensate (g/m²)')

#ax.set_xlim(np.datetime64(start), np.datetime64(end))
None
```




## 1.10 Cloud masks

Different instruments, by virtue of their differing measurement principle and footprint, see clouds in different ways. To provide an overview of the cloud fields sampled by HALO during EUREC4A, a cloud mask is created for each cloud sensitive instrument.In the following, we compare the different cloud mask products for a case study on 5 February and further provide a statistical overview for the full campaign period.

More information on the dataset can be found in Konow et al. (in preparation). If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` or `author`.

```
import eurec4a
import numpy as np
import xarray as xr
import matplotlib as mpl

from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib.colors import LogNorm
```

We generally don't support ignoring warnings, however, we do so in this notebook to suppress warnings originating from zero division. Unfortunately, we coudn't find another way to handle the warnings. If you have an idea, please make a pull request and change it :)

```
import warnings
warnings.filterwarnings("ignore")
```

### 1.10.1 Cloud cover functions

We define some utility functions to extract (circle) cloud cover estimates from different datasets assuming that the datasets follow a certain flag meaning convention.

In particular, we define a **minimum cloud cover** based on the cloud mask flag `most_likely_cloudy` and a **maximum cloud cover** combining the flags `most_likely_cloudy` and `probably_cloudy`. The cloud mask datasets slightly vary in their flag definition for cloud free conditions and their handling of unvalid measurements which is taken care of by the following functions.

```
def _isvalid(da):
    meanings = dict(zip(da.flag_meanings.split(" "), da.flag_values))
    return ~(np.isnan(da) | ("unknown" in meanings and da==meanings["unknown"]))

def _cloudy_max(da):
    meanings = dict(zip(da.flag_meanings.split(" "), da.flag_values))
    return (da==meanings["most_likely_cloudy"]) | (da==meanings["probably_cloudy"])

def _cloudy_min(da):
    meanings = dict(zip(da.flag_meanings.split(" "), da.flag_values))
    return da==meanings["most_likely_cloudy"]

def cfmin(ds):
    sumdims = [d for d in ds.cloud_mask.dims if d != "time"]
    return (_cloudy_min(ds.cloud_mask).sum(dim=sumdims)
            / _isvalid(ds.cloud_mask).sum(dim=sumdims))

def cfmax(ds):
    sumdims = [d for d in ds.cloud_mask.dims if d != "time"]
    return (_cloudy_max(ds.cloud_mask).sum(dim=sumdims)
            / _isvalid(ds.cloud_mask).sum(dim=sumdims))

def correct_VELOX(ds):
    return ds.assign(CF_min=ds.CF_min.where((ds.CF_min >=0 ) & (ds.CF_min <= 1)),
                     CF_max=ds.CF_max.where((ds.CF_max >=0 ) & (ds.CF_max <= 1)))

def ensure_cfminmax(ds):
    if "CF_min" not in ds:
        ds = ds.assign(CF_min=cfmin)
    if "CF_max" not in ds:
        ds = ds.assign(CF_max=cfmax)
    return correct_VELOX(ds)

from multiprocessing.pool import ThreadPool

def load_cloudmask_dataset(cat_item):
    # load in parallel as this function is mainly limited by the network roundtrip␣
 ↪time
    p = ThreadPool(20)
    return ensure_cfminmax(xr.concat(list(p.map(lambda v: v.get().to_dask().chunk(),
                                                cat_item.values())),
                                     dim="time"))
```

### 1.10.2 Get data

We use the eurec4a intake catalog to access the data files.

```
cat = eurec4a.get_intake_catalog()
list(cat.HALO)
```

```
['BAHAMAS',
 'specMACS',
 'UNIFIED',
 'SMART',
 'VELOX',
 'KT19',
 'WALES',
 'BACARDI',
 'track']
```

For each instrument, we extract the data from individual flights and concatenate them to campaign-spanning datasets for the cloud mask files.

```
cat_cloudmask = {
    "WALES": cat.HALO.WALES.cloudparameter,
    "HAMP Radar": cat.HALO.UNIFIED.HAMPradar_cloudmask,
    "specMACS": cat.HALO.specMACS.cloudmaskSWIR,
    "HAMP Radiometer": cat.HALO.UNIFIED.HAMPradiometer_cloudmask,
    "KT19": cat.HALO.KT19.cloudmask,
    "VELOX": cat.HALO.VELOX.cloudmask,
}
```

```
data = {k: load_cloudmask_dataset(v) for k, v in cat_cloudmask.items()}
```

We have a look at the time periods spanned by the individual datasets: The datasets `HAMP Radar`, `HAMP Radiometer`, and `specMACS` include measurements from the transfer flights on 19 January to Barbados and on 18 February back over the Atlantic to Europe. The datasets `WALES`, `KT19`, and `VELOX` are limited to the 13 local research flights between 22 January and 15 February.

```
for k, v in data.items():
    print(f"{k}: {v.isel(time=0).time.values} -  {v.isel(time=-1).time.values}")
```

```
WALES: 2020-01-22T15:18:00.155000064 -  2020-02-15T23:30:02.176000000
HAMP Radar: 2020-01-19T09:34:25.000004352 -  2020-02-18T18:55:30.999997440
specMACS: 2020-01-19T09:29:00.063636992 -  2020-02-18T17:27:00.982214912
HAMP Radiometer: 2020-01-19T09:34:25.000004352 -  2020-02-18T18:55:30.999997440
KT19: 2020-01-22T14:57:36.041336059 -  2020-02-15T23:59:53.497467041
VELOX: 2020-01-24T09:33:35.000000000 -  2020-02-15T23:25:18.000000000
```

### 1.10.3 Case study on February 5

We show the cloud masks from the various instruments for an 5 minute time interval around noon on February 5.

We add 2D vertical lidar and radar data, as well as 2D horizontal imager data for a better visualization of the cloud information content provided by the instruments.

#### Data preprocessing

#### Time interval

```
s = slice("2020-02-05T11:22:00", "2020-02-05T11:27:00")
```

#### HAMP radar reflectivities

```
ds_radar = cat.HALO.UNIFIED.HAMPradar["HALO-0205"].to_dask()
da_radar = ds_radar.dBZ.sel(height=slice(0, 4000), time=s)
```

#### WALES lidar backscatter ratio at 1064 nm

The WALES dataset has a `range` coordinate that can be translated into a `height` coordinate by: \begin{equation} height = height_above_sea_level[0] - range \end{equation}

```
def add_height_coordinate_wales(ds):
    ds.coords["height"] = ds.height_above_sea_level[0].values - ds.range
    return ds.swap_dims({"range": "height"})
```

```
ds_bsri = add_height_coordinate_wales(cat.HALO.WALES.bsri["HALO-0205"].to_dask())
da_bsri = ds_bsri.backscatter_ratio.sel(height=slice(4000, 0), time=s)
```

From WALES we also include the cloud top height information

```
da_cth = cat.HALO.WALES.cloudparameter["HALO-0205"].to_dask().cloud_top.sel(time=s)
```

#### SpecMACS imager radiance

From specMACS we include the radiances at 1.6 micron in the short-wave infrared (SWIR).

**Note:** this dataset is only available for the following application on February 5, not for the whole campaign.

```
url = ("https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/SPECMACS-
 ↪CLOUDMASK/"
       + "EUREC4A_HALO_specMACS_cloud_mask_20200205T100000-20200205T182359_v1.1.nc")
ds_swir = xr.open_dataset(url, engine="netcdf4")
da_swir = ds_swir.sel(time=s).isel(radiation_wavelength=0).swir_radiance
```

### VELOX broadband IR brightness temperature

Next to the SpecMACS SWIR radiance, we include broadband brightness temperatures from VELOX (7.7 - 12 μm).

**Note:** this dataset is only available for the following application on February 5, not for the whole campaign.

```
ds_bt = xr.open_zarr("ipfs://QmQEwkhhHdJkiThf4hnj9G3wgqVreBnWGrX2A5kT6CrtY7",
                     consolidated=True,
                    ).assign_coords(va=lambda x: x.va)
```

### Preprocess cloud mask data

We copy the data dictionary and apply the time selection.

For the 2D horizontal imagers we select a region in the center, derive a representative (most frequent) `cloud_mask` flag value and use that in the following intercomparison plot.

- VELOX: we select only the central 11 x 11 pixels, i.e. view angle = 0 ∓ 0.2865 (see below)
- SpecMACS: we select the central 0.6 degrees, i.e. angle = 0 ∓ 0.3

What is the angle of the 11 central pixel in accross track direction for VELOX?

```
xmid = ds_bt.x.size // 2
va_central = ds_bt.isel(time=0, x=slice(xmid - 5, xmid + 6)).va
(va_central.max() - va_central.min()).values
```

```
array(0.57295305, dtype=float32)
```

```
def most_frequent_flag(var, dims):
    flags = xr.DataArray(var.flag_values, dims=("__internal__flags__"))
    flag_indices = (var == flags).sum(dims).argmax("__internal__flags__")
    return xr.DataArray(var.flag_values[flag_indices.data],
                        dims=flag_indices.dims,
                        attrs=var.attrs)

def select_specmacs_cloudmask(ds):
    specmacs = ds.sel(angle=slice(.3, -.3))
    return ds.assign({"cloud_mask": most_frequent_flag(specmacs.cloud_mask, "angle"),
                      "CF_min": cfmin(specmacs),
                      "CF_max": cfmax(specmacs)})

def select_velox_cloudmask(ds):
    xmid = ds.x.size // 2
    ymid = ds.y.size // 2
    velox = ds.isel(x=slice(xmid - 5, xmid + 6), y=slice(ymid - 5, ymid + 6))
    return ds.assign({"cloud_mask": most_frequent_flag(velox.cloud_mask, ("x", "y")),
                      "CF_min": cfmin(velox),
                      "CF_max": cfmax(velox)})

cloudmask_selectors = {
    "WALES": lambda ds: ds,
    "HAMP Radar": lambda ds: ds,
    "specMACS": select_specmacs_cloudmask,
```

(continues on next page)

(continued from previous page)

```
    "HAMP Radiometer": lambda ds: ds,
    "KT19": lambda ds: ds,
    "VELOX": select_velox_cloudmask,
}
```

```
data0205 = {k: ensure_cfminmax(v["HALO-0205"].to_dask()) for k, v in cat_cloudmask.
 ↪items()}
data_feb5 = {k: cloudmask_selectors[k](v.sel(time=s)) for k, v in data0205.items()}
```

## Plot

```
colors={
    "WALES": "darkgreen",
    "HAMP Radar": "navy",
    "specMACS": "darkred",
    "HAMP Radiometer": "palevioletred",
    "KT19": "coral",
    "VELOX": "cadetblue",
       }
```

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book"])
```

```
with plt.style.context("mplstyle/square"):
    fig, (ax0, ax1, ax2, ax3, axLegend, axL1, axL2, axL3, axL4, axL5, axL6) = plt.
 ↪subplots(
        11, 1, sharex=True, gridspec_kw={"height_ratios": [3, 3, 3, 3, 0.5, 0.5, 0.5,
                                                           0.5, 0.5, 0.5, 0.5]}
    )

    ## 2D vertical
    # Wales backscatter ratio
    im0 = da_bsri.plot.pcolormesh(
        ax=ax0, x="time", y="height", norm=LogNorm(vmin=1, vmax=100),
        cmap="Spectral_r", rasterized=True, add_colorbar=False
    )
    cax0 = make_axes_locatable(ax0).append_axes("right", size="1%", pad=-0.05)
    fig.colorbar(im0, cax=cax0, label="backscatter ratio", extend='both')
    # cloud top height
    da_cth.plot(ax=ax0, x="time", ls="", marker=".", color="k", label="Cloud top")
    ax0.legend()

    # Radar reflectivity
    im1 = da_radar.plot(ax=ax1, x="time", rasterized=True, add_colorbar=False)
    cax1 = make_axes_locatable(ax1).append_axes("right", size="1%", pad=-0.05)
    fig.colorbar(im1, cax=cax1, label="reflectivity / dBZ")

    for ax in [ax0, ax1]:
        ax.set_yticks([0, 1000, 2000, 3000])
        ax.set_ylabel("height / m")

    ## 2D horizontal
```

(continues on next page)

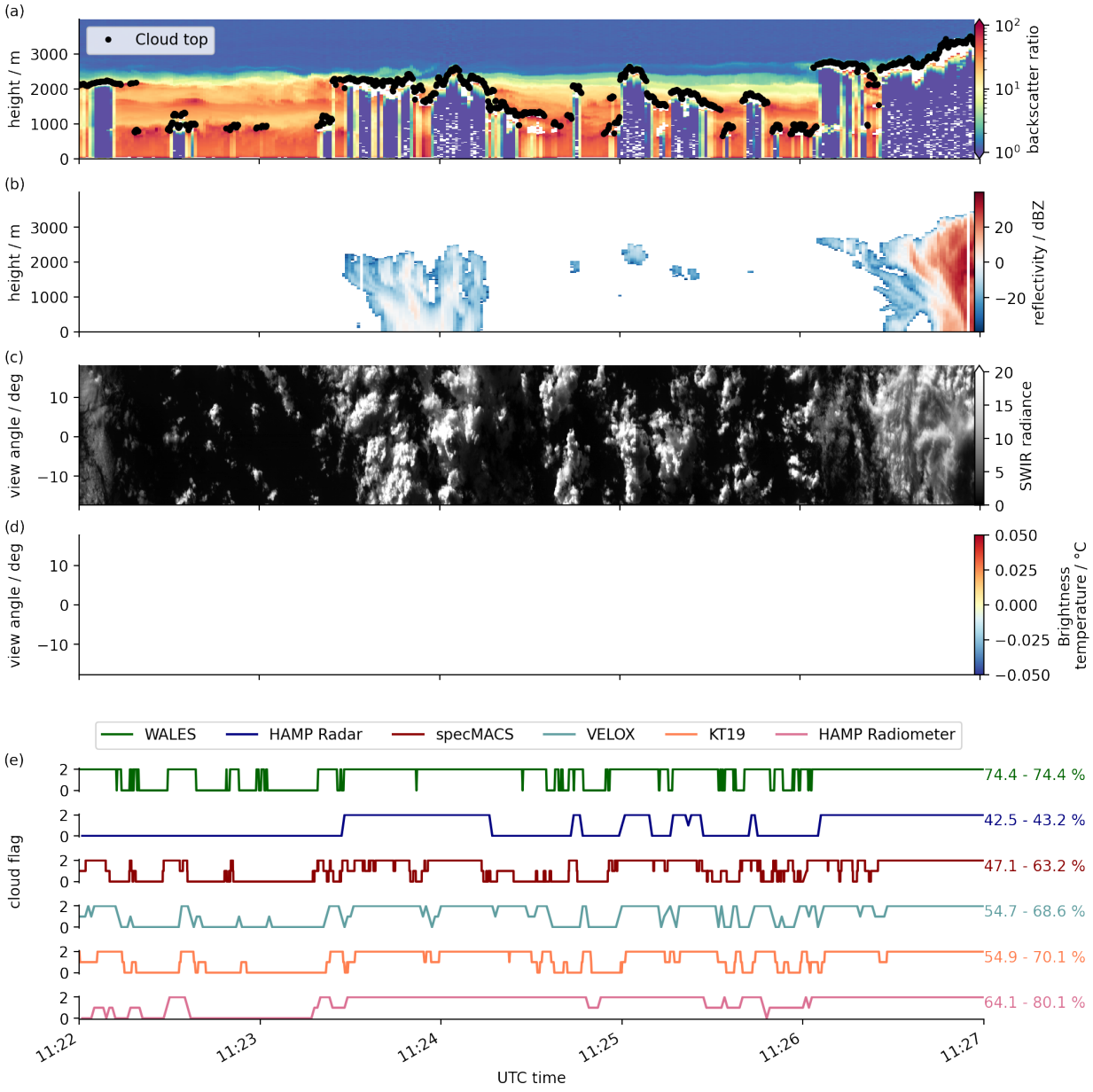
(continued from previous page)

```
    # SpecMACS radiance
    im2 = da_swir.plot.pcolormesh(ax=ax2, x="time", y="angle", cmap="Greys_r",
                                  vmin=0, vmax=20, rasterized=True, add_
↪colorbar=False)
    cax2 = make_axes_locatable(ax2).append_axes("right", size="1%", pad=-0.05)
    fig.colorbar(im2, cax=cax2, label="SWIR radiance", extend='max')
    ax2.set_ylabel("view angle / deg")

    # VELOX brightness temperature
    im3 = (ds_bt.Brightness_temperature - 273.15).plot.pcolormesh(ax=ax3, x="time", y=
↪"va",
           cmap="RdYlBu_r", rasterized=True, add_colorbar=False)
    cax3 = make_axes_locatable(ax3).append_axes("right", size="1%", pad=-0.05)
    fig.colorbar(im3, cax=cax3, label="Brightness\ntemperature / °C")
    ax3.set_ylabel("view angle / deg")


    ## We leave an empty axis to put the legend here
    [s.set_visible(False) for s in axLegend.spines.values()]
    axLegend.xaxis.set_visible(False)
    axLegend.yaxis.set_visible(False)

    ## 1D
    # We plot 1D cloud masks
    # Each we annotate with a total min and max cloud cover for the scene shown and
    # remove disturbing spines
    lines = []
    plot_order = ['WALES', 'HAMP Radar', 'specMACS', 'VELOX', 'KT19', 'HAMP Radiometer
↪']
    axes = dict(zip(plot_order, [axL1, axL2, axL3, axL4, axL5, axL6]))

    for k in plot_order:
        ds = data_feb5[k]
        lines += ds.cloud_mask.plot.line(ax=axes[k], x="time", label=k,␣
↪color=colors[k])
        if axes[k]!=axL6:
            axes[k].spines["bottom"].set_visible(False)
            axes[k].xaxis.set_visible(False)
        axes[k].set_ylabel("")
        axes[k].annotate(f"{ds.CF_min.mean().values * 100:.1f}"
                         + f" - {ds.CF_max.mean().values * 100:.1f} %",
                         (1, 0.5), color=colors[k], xycoords="axes fraction")

    # We add one legend for all 1D cloud masks
    labels = [l.get_label() for l in lines]
    axL1.legend(lines, labels, ncol=7, bbox_to_anchor=(0.01, 1.5))
    axL3.set_ylabel("cloud flag")

    for ax in [ax0, ax1, ax2, ax3, axLegend, axL1, axL2, axL3, axL4, axL5, axL6]:
        ax.set_xlabel("")
        ax.set_title("")

    axL6.set_xlabel("UTC time")
    axL6.set_xticks(np.arange(np.datetime64('2020-02-05T11:22:00'),
                    np.datetime64('2020-02-05T11:28:00'), np.timedelta64(1, 'm')))
    ax0.xaxis.set_major_formatter(mpl.dates.DateFormatter('%H:%M'))
    fig.autofmt_xdate()
```

(continues on next page)

(continued from previous page)

```
    fig.align_ylabels(axs=[ax0, ax1, ax2, ax3, axL3])
    #fig.align_ylabels(axs=[cax0, cax1, cax2])

    label_pos_x = np.datetime64('2020-02-05T11:21:35')
    for ax, label in zip([ax0, ax1, ax2, ax3, axL1], ["(a)", "(b)", "(c)", "(d)", "(e)
↪"]):
        ax.text(label_pos_x, ax.get_ylim()[1], label, verticalalignment='bottom')
```



### 1.10.4 Statistical comparison

We will further compare cloud mask information from all HALO flights during EUREC4A on the basis of circle flight segments. Most of the time, the HALO aircraft sampled the airmass in circles east of Barbados. We use the meta data on flight segments, extract the information on start and end time of individual circles, and derive circle-average cloud cover.

For the 2D imagers VELOX and speMACS we use the full swath. In the case study above we had selected the central measurements for a better comparison with the other instruments. However, in the following we investigate the broad statistics and therefore include as much information on the cloud field as we can get from the full footprints of each instrument.

The following statistics are based on the **minimum cloud cover** including the cloud mask flag `most_likely_cloudy` and **maximum cloud cover** with cloud mask flags $\in$ {`most_likely_cloudy`, `probably_cloudy`}.

```
def midpoint(a, b):
    return a + (b - a) / 2

def cf_circles(ds):
    return xr.concat(
        [
            xr.Dataset({
                "CF_max": ds.sel(time=slice(i["start"], i["end"])).CF_max.mean(dim=
 ↪"time"),
                "CF_min": ds.sel(time=slice(i["start"], i["end"])).CF_min.mean(dim=
 ↪"time"),
                "time": xr.DataArray(midpoint(i["start"], i["end"]), dims=()),
                "segment_id": xr.DataArray(i["segment_id"], dims=())
            })
            for i in segments.values()
        ], dim="time")
```

#### Get meta data

```
meta = eurec4a.get_flight_segments()
```

We extract all flight IDs of HALO's research flights

```
flight_ids = [flight_id
              for platform_id, flights in meta.items()
              if platform_id=="HALO"
              for flight_id, flight in flights.items()
              ]
```

Within each flight we further extract the circle segments

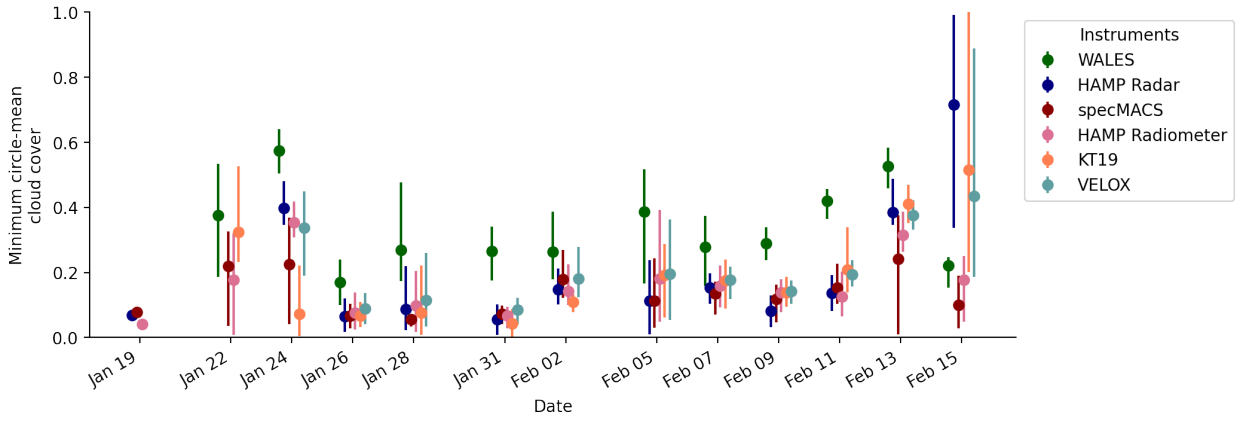
```
segments = {s["segment_id"]: {**s, "flight_id": flight["flight_id"]}
             for platform_id, flight_id in meta.items()
             if platform_id=="HALO"
             for flight in flight_id.values()
             for s in flight["segments"]
            if "circle" in s["kinds"]
            }
print(f"In total HALO flew {len(segments)} circles during EUREC4A")
```

```
In total HALO flew 72 circles during EUREC4A
```

### Time series of circle cloud cover

Time series of circle-mean (minimum) cloud cover estimates. The markers visualize the research-flight average, while the lines span the range of all circle-mean cloud cover estimates within a respective flight.

```
ts = np.timedelta64(2, 'h')

with plt.style.context("mplstyle/wide"):
    fig, ax = plt.subplots()
    for k, v in data.items():
        ds = cf_circles(v)
        ds["date"] = ds.time.astype('<M8[D]')
        ax.errorbar(
            x=np.unique(ds.date) + ts,
            y=ds.groupby("date").mean().CF_min.values,
            yerr=abs(np.array([ds.groupby("date").min().CF_min.values,
                               ds.groupby("date").max().CF_min.values])
                     - ds.groupby("date").mean().CF_min.values),
            fmt='o',
            color=colors[k],
            label=k,
        )
        ts += np.timedelta64(4, 'h')
    ax.set_ylim(0, 1)
    ax.set_xticks(np.unique(ds.date) + np.timedelta64(12, 'h'))
    ax.xaxis.set_major_formatter(mpl.dates.DateFormatter('%b %d'))
    fig.autofmt_xdate()

    ax.set_ylabel("Minimum circle-mean\ncloud cover")
    ax.legend(title="Instruments", bbox_to_anchor=(1,1), loc="upper left")
    ax.set_xlabel("Date")
```

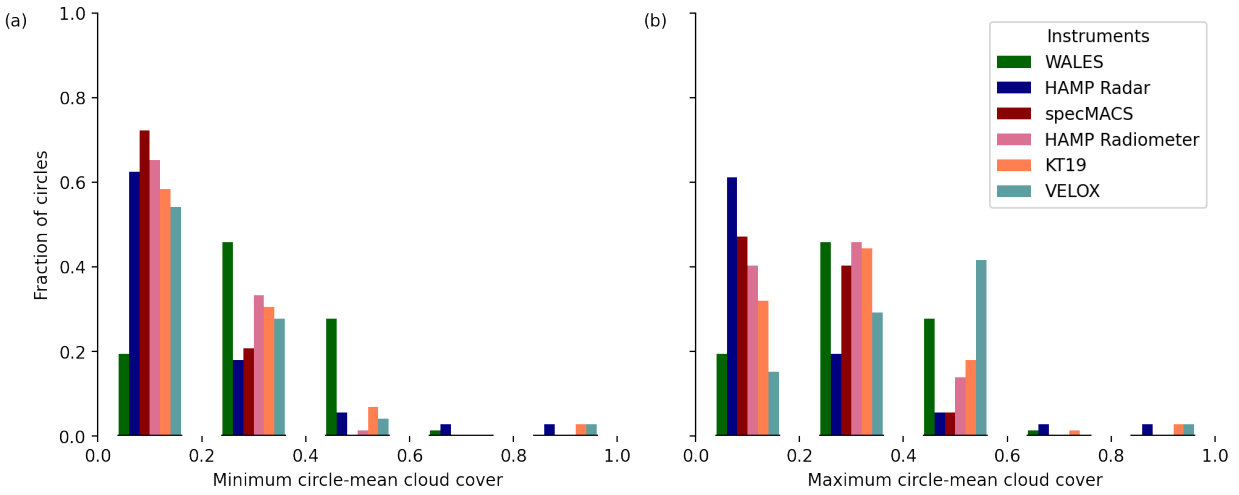


### Histogram of circle-mean cloud cover

We first have a look at the distributions of circle-mean cloud cover based on the minimum (`most_likely_cloudy`) and the maximum estimates (`most_likely_cloudy` & `probably_cloudy`).

```
binedges = np.arange(0, 1.2, .2)
binmids = (binedges[1:] + binedges[:-1]) / 2
```

```
with plt.style.context("mplstyle/wide"):
    fig, (ax0, ax1) = plt.subplots(1, 2, sharey=True)
    count = 0
    width = 0.02
    for k, v in data.items():
        ds = cf_circles(v)
        ax0.bar(x=binmids - 0.05 + count,
                height=np.histogram(ds.CF_min.values, bins=binedges)[0] / ds.time.
↪size,
                width=width, color=colors[k], label=k)
        ax1.bar(x=binmids - 0.05 + count,
                height=np.histogram(ds.CF_max.values, bins=binedges)[0] / ds.time.
↪size,
                width=width, color=colors[k], label=k)
        count += 0.02

    xticks = [0.1, 0.3, 0.5, 0.7, 0.9]

    for ax in [ax0, ax1]:
        ax.set_xlim(0, 1)
        ax.set_ylim(0, 1)
        ax.spines['bottom'].set_visible(False)
        for x in xticks:
            ax.axhline(y=0, xmin=x-0.06, xmax=x+0.06, color="k")
        ax.set_xticks([0, 0.2, 0.4, 0.6, 0.8, 1])


    ax0.set_xlabel("Minimum circle-mean cloud cover")
    ax1.set_xlabel("Maximum circle-mean cloud cover")
    ax0.set_ylabel("Fraction of circles")
    ax1.legend(title="Instruments", bbox_to_anchor=(.55,1), loc="upper left")

    ax0.text(-0.18, ax.get_ylim()[1], "(a)", verticalalignment='top')
    ax1.text(-0.1, ax.get_ylim()[1], "(b)", verticalalignment='top')
```



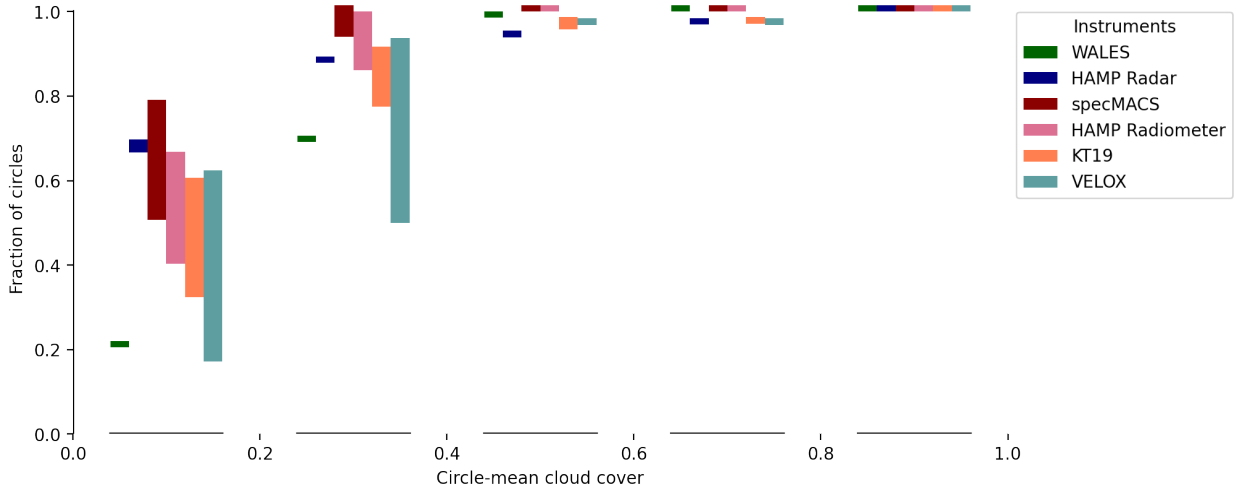

For a better comparison of minimum and maximum circle-mean cloud cover we merge the two above plots and show their difference in the following plot. In particular, we show the cumulative fraction of circle-mean cloud cover estimates. Depending on the instruments and some instrument downtimes, the available circle counts range from 64 to 72. The bins on the x-axis have a bin width of 0.2 respectively. The bars span the range defined by the minimum cloud cover based on cloud flag `most likely cloudy`and the maximum cloud cover based on cloud flags `most likely cloudy` and `probably cloudy`.

```
with plt.style.context("mplstyle/wide"):
    fig, ax = plt.subplots()
    count = 0
    width = 0.02
    for k, v in data.items():
        ds = cf_circles(v)
        hist_min = (np.histogram(ds.CF_min.values, bins=binedges)[0]
                    / (ds.time.size - np.isnan(ds.CF_min.values).sum()))
        hist_max = (np.histogram(ds.CF_max.values, bins=binedges)[0]
                    / (ds.time.size - np.isnan(ds.CF_max.values).sum()))
        # draw a fake default line at the minimum
        ax.bar(x=binmids - 0.05 + count,
                height=0.015,
                width=width,
                bottom=np.cumsum(hist_min),
                color=colors[k], label=k)
        ax.bar(x=binmids - 0.05 + count,
                height=np.cumsum(hist_max) - np.cumsum(hist_min),
                width=width,
                bottom=np.cumsum(hist_min),
                color=colors[k])
        count+=0.02

    # manually set x- and y-axis
    [s.set_visible(False) for s in ax.spines.values()]
    # x-axis
    ax.set_xlim(0, 1)
    xticks = [0.1, 0.3, 0.5, 0.7, 0.9]
    for x in xticks:
        ax.axhline(y=0, xmin=x-0.06, xmax=x+0.06, color="k")
    ax.set_xticks([0, 0.2, 0.4, 0.6, 0.8, 1])
    ax.set_xlabel("Circle-mean cloud cover")
```

(continues on next page)

(continued from previous page)

```
    # y-axis
    ax.set_ylim(0, 1.02)
    ax.plot([0, 0], [0, 1], color="k")
    ax.set_ylabel("Fraction of circles")

    ax.legend(title="Instruments", bbox_to_anchor=(1,1), loc="upper left")
```




### A small interpretation attempt

We highlight a few features that stick out in the above figures showing the cloud cover statistics.

Time series:

- the WALES dataset does not have a `probably_cloudy` flag, and thus, the `CF_min` and `CF_max` variables are the same. The instrument design and methodology used to define the cloud flag seems to be very sensitive to small and optically thin clouds and the cloud cover estimates agree better with the `CF_max` of all other instruments.
- The time series shows that on the transfer flight on 19 January the HAMP radar and radiometer and specMACS datasets were processed and datasets are available including one circle near Barbados. For WALES, VELOX, and KT19 no data is available.
- The disagreement in cloud cover estimates for the flight on 15 February is partly due to a deep stratocumulus layer with a strong reflection at cloud top that blinded the lidar (WALES), while the radar was still able to provide reasonable estimates.

Distributions:

- All instruments, excluding WALES, agree well in the distribution of circle-mean cloud cover estimates according to their `CF_min` variable, while they vary on their definition of uncertain `probably_cloudy` measurements that are included in `CF_max`.
- In the case of VELOX as well as for all other passive instruments, the cloud cover estimates shift to higher numbers when including the uncertain cloud flag (from minimum to maximum cloud cover).
- The fraction of circle-mean cloud cover difference (minimum to maximum) shows a disagreement between the instruments for cloud cover ranges up to about 0.5 due to their different detection principles. Optically thin clouds can have a significant impact on circle-mean estimates in low cloud cover situations and lead to uncertain pixels. WALES can detect such thin clouds and suggest generally higher cloud cover with the change in cumulative fraction being strongest between 0.2 and 0.6.

- In general we find that only few circles have a cloud cover higher than 0.6. At such high cloud cover the instruments agree remarkably well and also, minimum and maximum cloud cover are almost equal.
- About 50 % of the time cloud cover estimates are below 0.2.

### 1.10.5 Camapign mean cloud cover

- from all available data including the transfer flights

```
print("Instrument: min - max")
print("")
for k, v in data.items():
    print(f"{k}: {v.CF_min.mean().values:.2f} - {v.CF_max.mean().values:.2f} ")
```

```
Instrument: min - max
```

```
WALES: 0.34 - 0.34
```

```
HAMP Radar: 0.25 - 0.25
```

```
specMACS: 0.18 - 0.24
```

```
HAMP Radiometer: 0.17 - 0.25
```

```
KT19: 0.20 - 0.31
```

```
VELOX: 0.21 - 0.39
```

- only from local research flights

```
localRF = slice("2020-01-22T00:00:00", "2020-02-15T23:59:59")
```

```
print("Instrument: min - max")
print("")
for k, v in data.items():
    print(f"{k}: {v.sel(time=localRF).CF_min.mean().values:.2f} - "
          + f"{v.sel(time=localRF).CF_max.mean().values:.2f} ")
```

```
Instrument: min - max

WALES: 0.34 - 0.34
HAMP Radar: 0.21 - 0.22
```

```
specMACS: 0.16 - 0.22
HAMP Radiometer: 0.16 - 0.25
KT19: 0.20 - 0.31
VELOX: 0.21 - 0.39
```

CHAPTER

TWO

# NOAA P3 "MISS PIGGY"

During EUREC4A and ATOMIC the US agency NOAA operated a Lockheed WP-3D Orion research aircraft from the island of Barbados during the period Jan 17 - Feb 11 2020. The aircraft, known formally as N43RF and informally as "Miss Piggy," is one of two such aircraft in NOAA's Hurricane Hunter fleet. (The other is known as "Kermit.")

ATOMIC included a cruise by the NOAA ship Ronald H. Brown (RHB). Both the P-3 and the RHB primarily operated east of the EUREC4A area (i.e. east of 57E), nominally upwind, within the "Tradewind Alley" extending eastwards from Barbados towards the Northwest Tropical Atlantic Station (NTAS) buoy near 15N, 51W. Many of the eleven P-3 flights included excursions to the location of the RHB and sampling of atmospheric and oceanic conditions around the ship and other ocean vehicles. Because of its large size and long endurance (most flights were 8-9 hours long) the P-3 was tasked with obtaining a wide array of observations including remote sensing of clouds and the ocean surface, *in situ* measurements within clouds and of isotopic composition throughout the lower troposphere, and the deployment of expendable profiling instruments in the atmosphere and ocean.

Most observations obtained from the P-3 are described in [PFB+21].

## 2.1 Flight tracks

The P3 flew 95 hours of observations over eleven flights, many of which were coordinated with the NOAA research ship R/V Ronald H. Brown and autonomous platforms deployed from the ship. Each flight contained a mixture of sampling strategies including: high-altitude circles with frequent dropsonde deployment to characterize the large-scale environment; slow descents and ascents to measure the distribution of water vapor and its isotopic composition; stacked legs aimed at sampling the microphysical and thermodynamic state of the boundary layer; and offset straight flight legs for observing clouds and the ocean surface with remote sensing instruments and the thermal structure of the ocean with *in situ* sensors dropped from the plane.

As a result of this diverse sampling the flight tracks are much more variable then for most of the other aircraft.

General setup:

```
import xarray as xr
import numpy as np
import datetime
#
# Related to plotting
#
import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book"])
%matplotlib inline
```

Now access the flight track data.


N43RF

```
import eurec4a
cat = eurec4a.get_intake_catalog()
```

Mapping takes quite some setup. Maybe we'll encapsulate this later but for now we repeat code in each notebook.

What days did the P-3 fly on? We can find out via the flight segmentation files.

```
# On what days did the P-3 fly? These are UTC date
all_flight_segments = eurec4a.get_flight_segments()
flight_dates = np.unique([np.datetime64(flight["takeoff"]).astype("datetime64[D]")
                          for flight in all_flight_segments["P3"].values()])
```

Now set up colors to code each flight date during the experiment. One could choose a categorical palette so the colors were as different from each other as possible. Here we'll choose from a continuous set that spans the experiment so days that are close in time are also close in color.

For plotting purposes it'll be handy to define a one-day time window and to convert between date/time formats

```
one_day = np.timedelta64(1, "D")

def to_datetime(dt64):
    epoch = np.datetime64("1970-01-01")
    second = np.timedelta64(1, "s")
    return datetime.datetime.utcfromtimestamp((dt64 - epoch) / second)
```

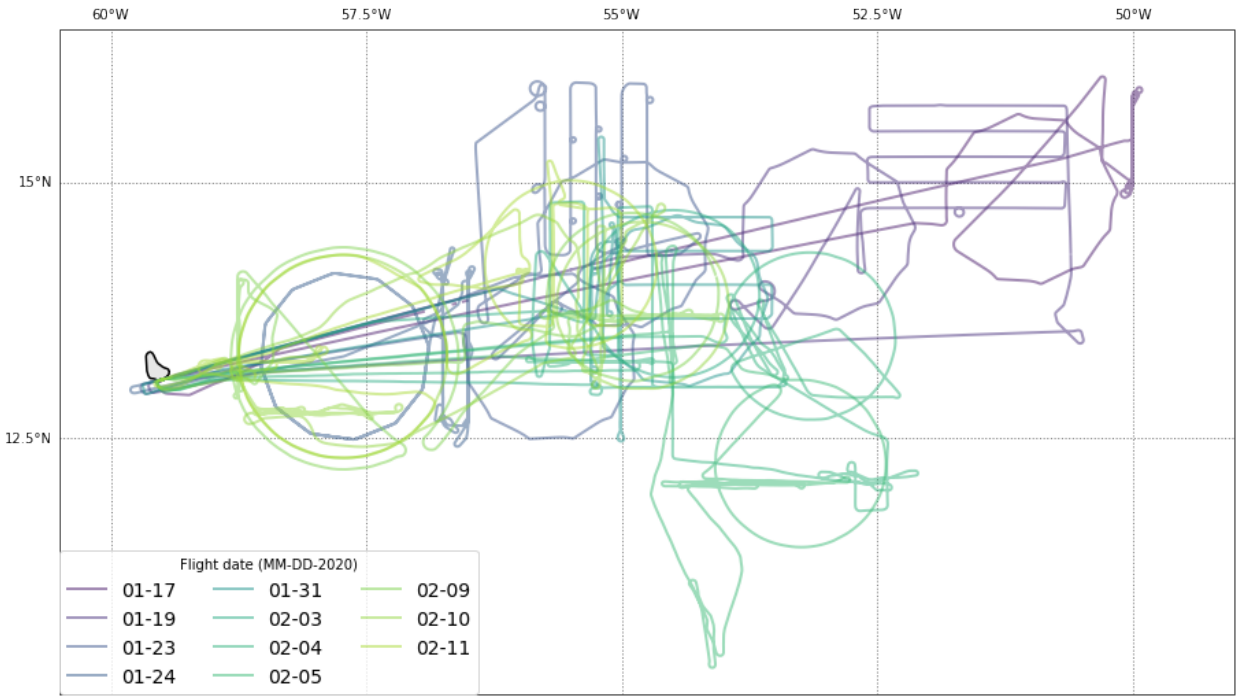
Most platforms available from the EUREC4A `intake` catalog have a `tracks` element but we'll use the `flight-level` data instead. We're using `xr.concat()` with one `dask` array per day to avoid loading the whole dataset into memory at once.

```
nav_data = xr.concat([entry.to_dask().chunk() for entry in cat.P3.flight_level.
 ↪values()],
                     dim = "time")
```

A map showing each of the eleven flight tracks:

```
fig = plt.figure(figsize = (12,13.6))
ax  = set_up_map(plt)
add_gridlines(ax)

for d in flight_dates:
    flight = nav_data.sel(time=slice(d, d + one_day))
    ax.plot(flight.lon, flight.lat,
            lw=2, alpha=0.5, c=color_of_day(d),
            transform=ccrs.PlateCarree(), zorder=7,
            label=f"{to_datetime(d):%m-%d}")

plt.legend(ncol=3, loc=(0.0,0.0), fontsize=14, framealpha=0.8, markerscale=5,
           title="Flight date (MM-DD-2020)")
```

```
<matplotlib.legend.Legend at 0x137110310>
```

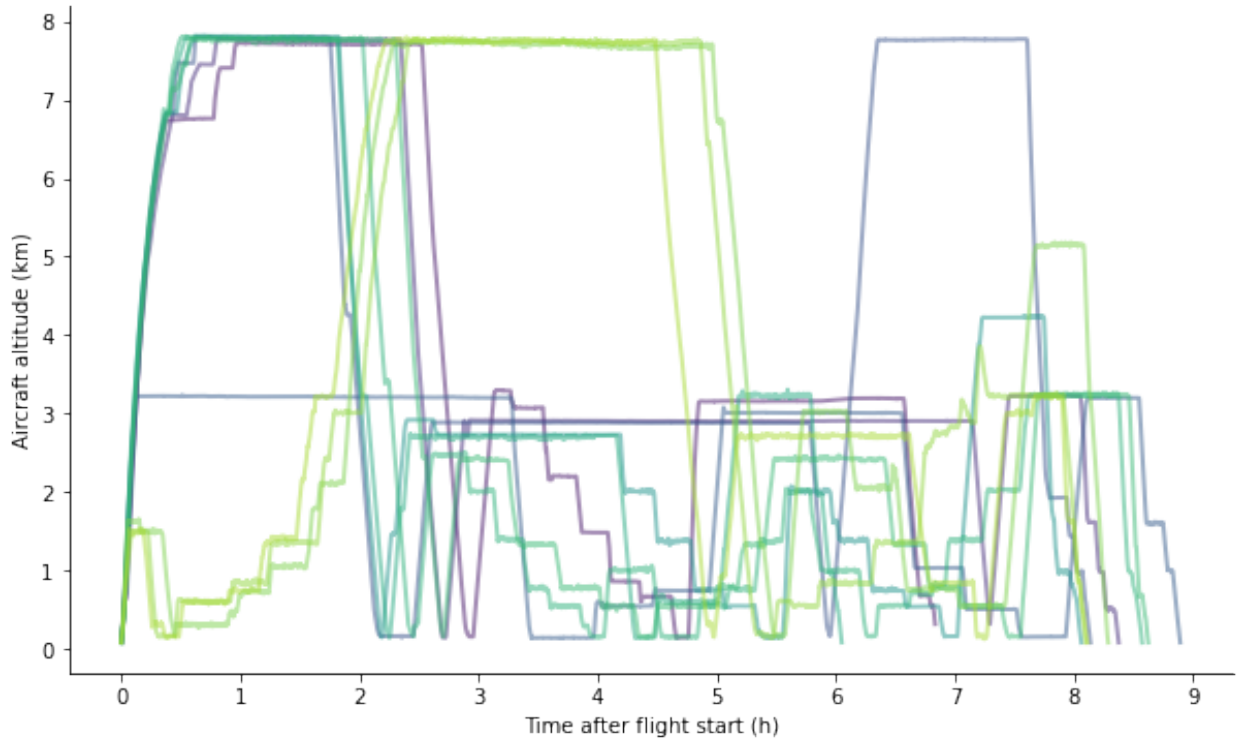



Most dropsondes were deployed from regular dodecagons during the first part of the experiment with short turns after each dropsonde providing an off-nadir look at the ocean surface useful for calibrating the W-band radar. A change in pilots midway through the experiment led to dropsondes being deployed from circular flight tracks starting on 31 Jan. AXBTs were deployed in lawnmower patterns (parallel offset legs) with small loops sometimes employed to lengthen the time between AXBT deployment to allow time for data acquisition given the device's slow fall speeds. Profiling and especially *in situ* cloud sampling legs sometimes deviated from straight paths to avoid hazardous weather.

Side view using the same color scale:

```
fig = plt.figure(figsize = (8.3,5))
ax = plt.axes()

for d in flight_dates:
    flight = nav_data.sel(time=slice(d, d + one_day))
    flight = flight.where(flight.alt > 80, drop=True) # Proxy for take-off time
    plt.plot(flight.time - flight.time.min(), flight.alt/1000.,
             lw=2, alpha=0.5, c=color_of_day(d),
             label=f"{to_datetime(d):%m-%d}")

plt.xticks(np.arange(10) * 3600 * 1e9, labels = np.arange(10))
ax.set_xlabel("Time after flight start (h)")
ax.set_ylabel("Aircraft altitude (km)")
```

```
Text(0, 0.5, 'Aircraft altitude (km)')
```




Sondes were dropped from the P-3 at about 7.5 km, with each circle taking roughly an hour; transits were frequently performed at this level to conserve fuel. Long intervals near 3 km altitude were used to deploy AXBTs and/or characterize the ocean surface with remote sensing. Stepped legs indicate times devoted to *in situ* cloud sampling. On most flights the aircraft climbed quickly to roughly 7.5 km, partly to deconflict with other aircraft participating in the experiment. On the three night flights, however, no other aircraft were operating at take-off times and cloud sampling was performed first, nearer Barbados than on other flights.

## 2.2 Humidity comparison: Hygrometer and isotope analyzer

During EUREC4A and ATOMIC the P3 made two *in situ* measurements of humidity, onewith the normal chilled-mirror hygrometer, and one with a cavity ring down spectrometer. The spectrometer's main purpose was to measure isotopes of water vapor but it's good for the total humidty as well.

For this example we need to plot some `xarray` datasets from the catalog.

```
import xarray as xr
import numpy as np

import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book"])
import colorcet as cc
%matplotlib inline

import eurec4a
cat = eurec4a.get_intake_catalog()
```

We'll create a combined data with relative humidity from the aircraft hygrometer ("fl" means "flight level") and the water vapor isotope analyzer ("iso") from a single flight. Drop (remove) times when the aircraft is on the ground.

```
fl = cat.P3.flight_level["P3-0119"].to_dask()
pi = cat.P3.isotope_analyzer.water_vapor_1hz["P3-0119"].to_dask()
rhs = xr.Dataset({"press" :fl["press"],
                  "alt"   :fl["alt"],
                  "rh_p3" :fl["RH"],
                  "rh_iso":pi["rh_iso"]})
rhs = rhs.where(rhs.alt > 80., drop = True)
```

Adriana Bailey from NCAR, who was responsible for the isotope analyzer, finds that while the two instruments agree most of the time, the hygrometer is subject to both overshooting (e.g. when the measured signal surpasses the expected value following a rapid rise in environmental water vapor concentration) and ringing (i.e. rapid oscillations around the expected value) during rapid and large changes in water vapor concentration.

Here are two figures to illustrate the problem and shows why you'd want to use the water vapor measurements from the isotope analyzer when they're available. The top panel combines data from two profiles and shows large excursions in the water vapor measured by the hygrometer. The lower panel shows that most measurements agree well but it's clear that it's better to use the the isotope analyzer measurements of humidity during the experiment.

```
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex = True, figsize = (8.3, 16.6))
#
# One time window
#
profiles = rhs.sel(time = slice(np.datetime64("2020-01-19T15:36:00"),
                                np.datetime64("2020-01-19T16:07:40")))
marker = "."
ax1.scatter(profiles["rh_p3" ], profiles["press"],
            color=cc.glasbey[3], marker = marker, label = "Hygrometer")
ax1.scatter(profiles["rh_iso"], profiles["press"],
            color="0.5",         marker = marker, label = "Isotope analyzer")
#
# A second time window, shown as squares of roughly the same size
#
profiles = rhs.sel(time = slice(np.datetime64("2020-01-19T20:31:12"),
                                np.datetime64("2020-01-19T20:41:16")))
marker = "s"
ax1.scatter(profiles["rh_p3" ], profiles["press"],
            color=cc.glasbey[3], marker = marker, s = .2 * plt.rcParams['lines.
 ↪markersize']**2)
ax1.scatter(profiles["rh_iso"], profiles["press"],
            color="0.5",         marker = marker, s = .2 * plt.rcParams['lines.
 ↪markersize']**2)


ax1.invert_yaxis()
ax1.legend(markerscale = 2)
ax1.set_xlim(0, 100)
ax1.set_ylabel("pressure (hPa)")

ax1.annotate('Ringing',      ( 5,890), fontsize="large")
ax1.annotate('Overshooting',(45,700), fontsize="large")
#
# Picarro vs. hygrometer RH - they mostly agree
#
ax2.scatter(rhs["rh_p3"],rhs["rh_iso"], s=3)
# 1:1 line
ax2.plot([0,102], [0,102], 'k-', color = 'black')
ax2.set_xlim(0, 102)
```

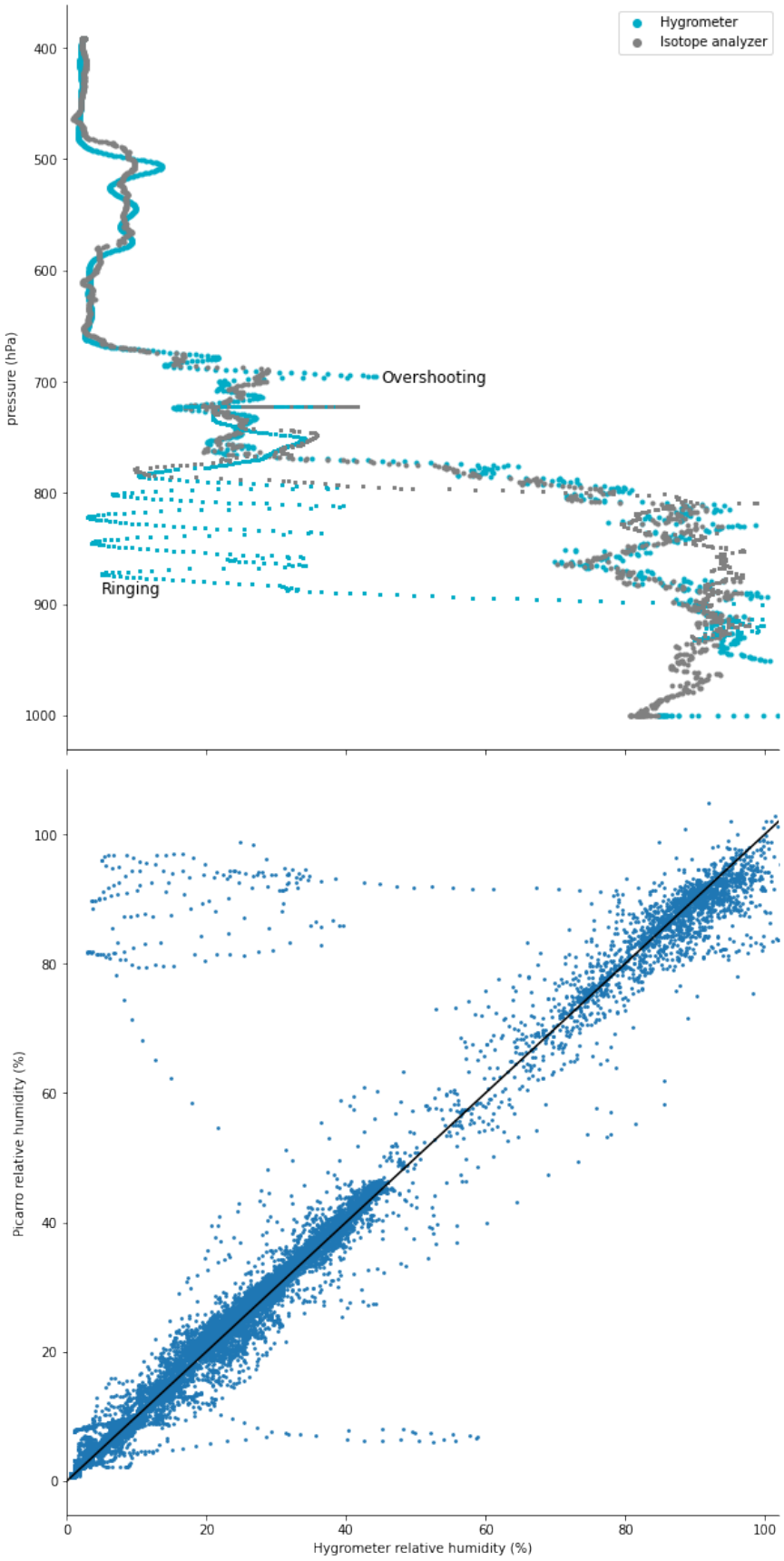
(continues on next page)

(continued from previous page)

```
ax2.set_xlabel("Hygrometer relative humidity (%)")
ax2.set_ylabel("Picarro relative humidity (%)")
```

```
<ipython-input-3-ca4e695269ff>:36: UserWarning: color is redundantly defined by the
 ↪'color' keyword argument and the fmt string "k-" (-> color='k'). The keyword␣
 ↪argument will take precedence.
  ax2.plot([0,102], [0,102], 'k-', color = 'black')
```

```
Text(0, 0.5, 'Picarro relative humidity (%)')
```


Hygrometer
Isotope analyzer
pressure (hPa)
400
500
600
700
800
900
1000
Overshooting
Ringing
Picarro relative humidity (%)
0
20
40
60
80
100
Hygrometer relative humidity (%)

## 2.3 W-band radar example

During EUREC4A and ATOMIC NOAA deployed W-band (94 GHz) radars on both the P-3 aircraft and the ship Ron Brown. The airborne radar was operated with 220 30-meter range gates with a dwell time of 0.5 seconds. The minimum detectable reflectivity of -36 dBZ at a range of 1 km although accurate estimates of Doppler properties require about -30 dBZ at 1 km.

The data are available through the EUREC4A intake catalog.

```
import datetime

import matplotlib.pyplot as plt
import colorcet as cc
%matplotlib inline

import eurec4a
cat = eurec4a.get_intake_catalog()
```

We'll select an hour's worth of observations from a single flight day, and mask out any observations with signal-to-noise ratio less than -10 dB.

```
time_slice = slice(datetime.datetime(2020, 1, 19, hour=18),
                   datetime.datetime(2020, 1, 19, hour=19))

cloud_params = cat.P3.remote_sensing['P3-0119'].to_dask().sel(time=time_slice)

w_band         = cat.P3.w_band_radar['P3-0119'].to_dask().sel(time=time_slice,␣
 ↪height=slice(0,3))
w_band         = w_band.where(w_band.snr > -10)
```

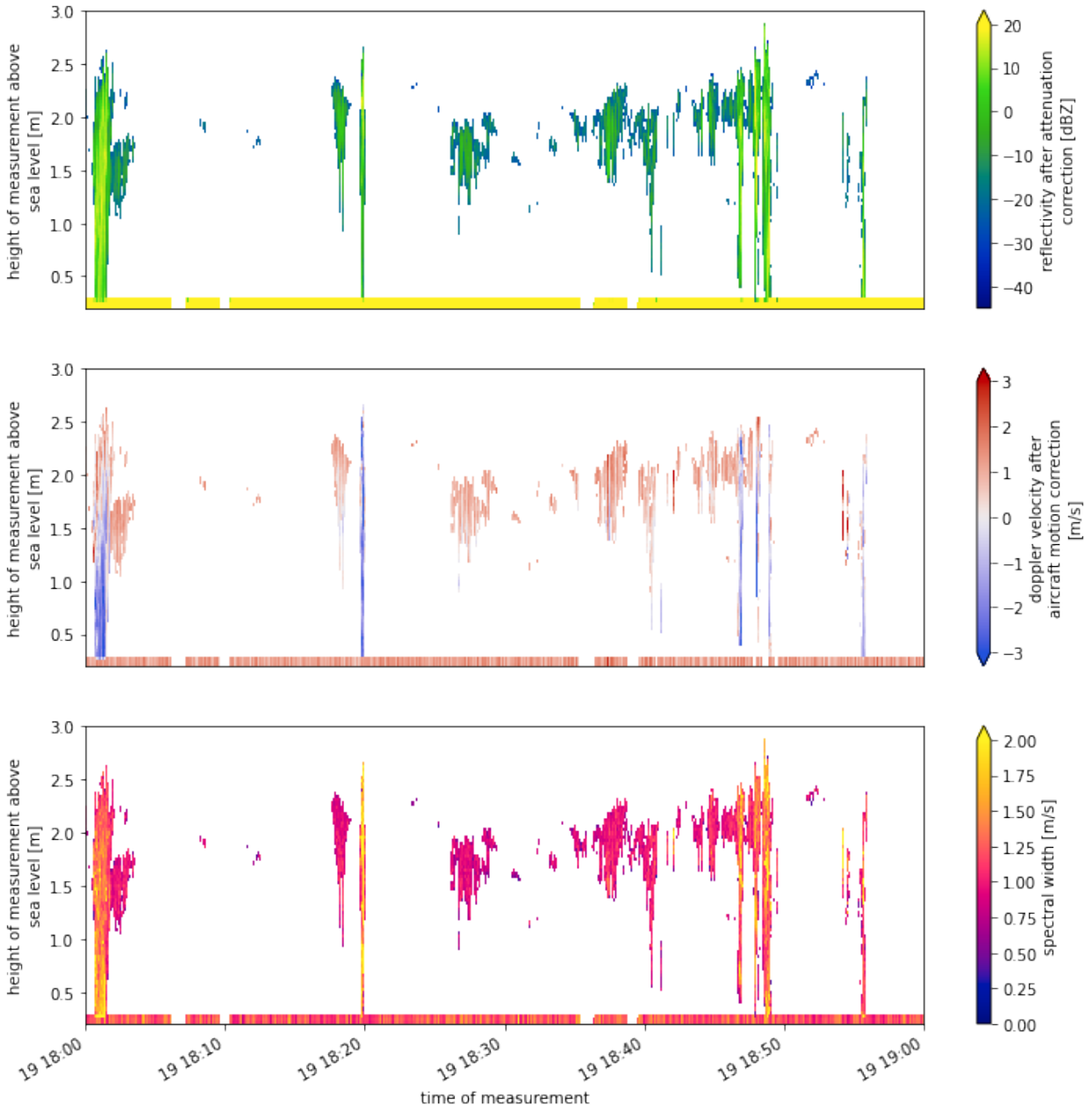
The three main quantities measured by the radar are the reflectivity, the Doppler velocity, and the spectral width.

```
fig = plt.figure(figsize = (12,10.2))

axes = fig.subplots(3, 1, sharex=True)
w_band.corrected_reflectivity.plot(x="time", y="height",
                                   ax = axes[0],
                                   vmin = -45, vmax = 20,
                                   cmap = cc.m_bgy)
w_band.corrected_doppler_velocity.plot(x="time", y="height",
                                       ax = axes[1],
                                       vmin = -3, vmax = 3,
                                       cmap = cc.m_coolwarm)

w_band.spectral_width.plot(x="time", y="height",
                           ax = axes[2],
                           vmin = 0, vmax = 2,
                           cmap = cc.m_bmy)

for ax in axes[0:2]:
    ax.tick_params(
    axis='x',          # changes apply to the x-axis
    which='both',      # both major and minor ticks are affected
    bottom=False,      # ticks along the bottom edge are off
    top=False,         # ticks along the top edge are off
    labelbottom=False) # labels along the bottom edge are off
    ax.xaxis.set_visible(False)
```

(continues on next page)

(continued from previous page)

```
fig.subplots_adjust(bottom = 0)
```



## 2.4 Ocean temperatures: AXBTs and SWIFT buoys

During EUREC4A/ATOMIC the P-3 deployed 165 Airborne eXpendable BathyThermographs (AXBTs) to measure profiles of ocean temperature. (These are kind of the oceanic equivalent of dropsondes but they don't measure salinity.) Often these were dropped around other ocean temperature measurements - for example the autonomous Surface Wave Instrument Floats with Tracking (SWIFT) buoys deployed from the Ron Brown by Elizabeth Thompson of NOAA and her colleagues. The SWIFT deployment is described in [QTC+21].

Let's take a look at some of the AXBT measurements and how they compare to the SWIFTs.

```
import xarray as xr
import numpy as np
import datetime

import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book"])
%matplotlib inline

import eurec4a
cat = eurec4a.get_intake_catalog()
```

Mapping takes quite some setup. Maybe we'll encapsulate this later but for now we repeat code in each notebook.

What days did the P-3 fly on? We can find out via the flight segmentation files.

```
# On what days did the P-3 fly? These are UTC date
all_flight_segments = eurec4a.get_flight_segments()
flight_dates = np.unique([np.datetime64(flight["takeoff"]).astype("datetime64[D]")
                          for flight in all_flight_segments["P3"].values()])
```

Now set up colors to code each flight date during the experiment. One could choose a categorical palette so the colors were as different from each other as possible. Here we'll choose from a continuous set that spans the experiment so days that are close in time are also close in color.

The P-3 only deployed AXBTs on some flights, and the SWIFT buoys were only deployed on a subset of those dates.

```
axbts = cat.P3.AXBT.Level_3.to_dask()
swifts = [cat[s].all.to_dask() for s in list(cat) if "SWIFT" in s]
axbt_dates = np.intersect1d(np.unique(axbts.time.astype("datetime64[D]").values),
                            flight_dates)

swift_candidates = np.unique(np.concatenate([swift.time.astype('datetime64[D]').values
                                             for swift in swifts]))
# Dates with potential SWIFT/P-3 overlap
swift_dates = np.intersect1d(swift_candidates, axbt_dates)
```

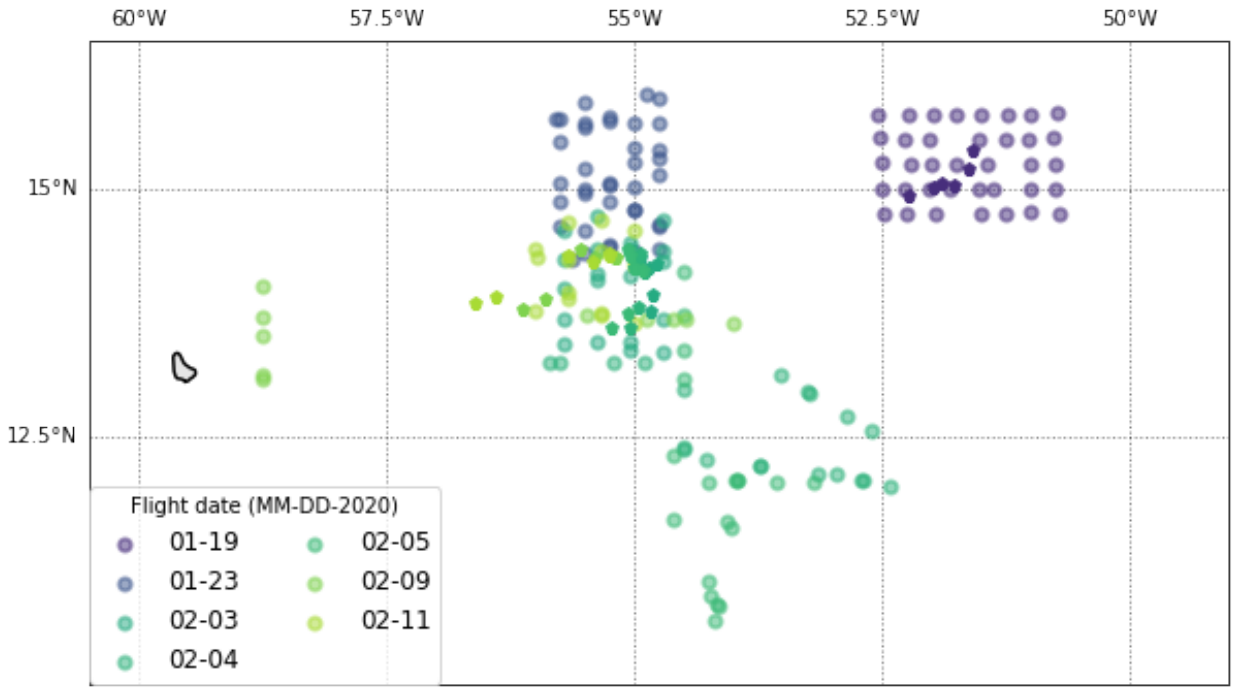
For plotting purposes it'll be handy to define a one-day time window and to convert between date/time formats

```
one_day = np.timedelta64(1, "D")

def to_datetime(dt64):
    epoch = np.datetime64("1970-01-01")
    second = np.timedelta64(1, "s")
    return datetime.datetime.utcfromtimestamp((dt64 - epoch) / second)
```

Now we can make a map that shows where the AXBTs were deployed and where the SWIFTs were on days there the two platforms overlapped

```
fig = plt.figure(figsize = (8.3, 9.4))
ax  = set_up_map(plt)
add_gridlines(ax)

#
# AXBT locations
#
for d in axbt_dates:
    flight = axbts.sel(time=slice(d, d + one_day))
    ax.scatter(flight.lon, flight.lat,
               lw=2, alpha=0.5, color=color_of_day(d),
               transform=ccrs.PlateCarree(), zorder=7,
               label=f"{to_datetime(d):%m-%d}")

#
# SWIFT locations on selected dates (where there's overlap)
#
for d in swift_dates:
    flight = axbts.sel(time=slice(d, d + one_day))
    for swift in swifts:
        drift = swift.sel(time = flight.time.mean(), method = "nearest")
        ax.scatter(drift.lon, drift.lat,
                   alpha=1, color=color_of_day(d),
                   transform=ccrs.PlateCarree(), zorder=7, marker = "p")

plt.legend(ncol=2,loc=(0.0,0.0),fontsize=12,framealpha=0.8,markerscale=1,
           title="Flight date (MM-DD-2020)")
```

```
<matplotlib.legend.Legend at 0x12bfd1760>
```

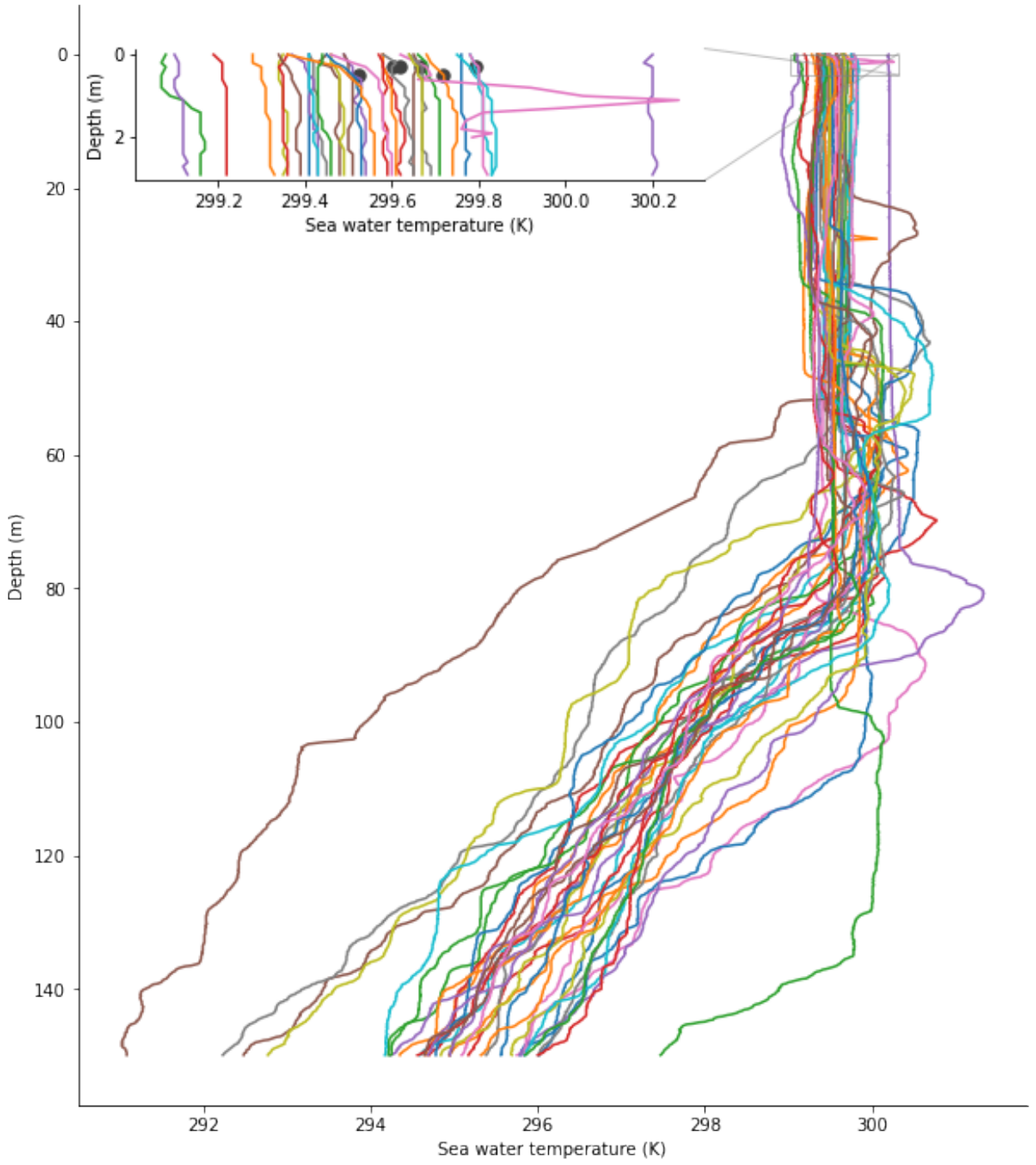



On 19 Jan and 3 Feb the AXBTs bracket the SWIFTs; on 23 Jan the SWIFTs are at the southern end of the AXBT pattern.

The next plot will focus on 19 Jan. Let's look at the profile of ocean temperature in the first 150 m from the AXBTs and compare the near-surface temperatures to the SWIFTs they are surrounding.

```
fig, ax = plt.subplots(figsize=[8.3, 9.4])
d = np.datetime64("2020-01-19")
axbt_1day   = axbts.sel(time=slice(d, d + one_day))
# Swift data at mean of AXBT times
swifts_1day = [s.sel(time = axbt_1day.time.mean(), method = "nearest") for s in␣
↪swifts]

axbt_1day.temperature.where(axbt_1day.depth < 150).plot.line(y="depth",
                                                             add_legend=False,␣
↪yincrease=False)
ax.set_xlabel("Sea water temperature (K)")
ax.set_ylabel("Depth (m)")

#
# Inset plot! https://matplotlib.org/3.1.1/gallery/subplots_axes_and_figures/zoom_
↪inset_axes.html
#
axin = ax.inset_axes([0.06, 0.84, 0.6, 0.12])
axin.scatter([s.sea_water_temperature.values + 273.15 for s in swifts_1day],
             # SWIFTs 16 and 17 report water temperature at 0.5 m depth; SWIFTs 23-25␣
↪report at 0.3 m
             # See the variable long_name or  Tables 8 and 9 of Quinn et al.
             [0.5 if '0.5' in s.sea_water_temperature.long_name else 0.3 for s in␣
↪swifts_1day],
             color="0.25",
             s = 1.5 * plt.rcParams['lines.markersize'] ** 2)
axbt_1day.temperature.where(axbt_1day.depth < 3).plot.line(y="depth",
                                                           add_legend=False,
                                                           yincrease=False, ax = axin)
axin.set_xlabel("Sea water temperature (K)")
axin.set_ylabel("Depth (m)")
ax.indicate_inset_zoom(axin)
```

```
(<matplotlib.patches.Rectangle at 0x133472310>,
 (<matplotlib.patches.ConnectionPatch at 0x132ac63a0>,
  <matplotlib.patches.ConnectionPatch at 0x132ac6880>,
  <matplotlib.patches.ConnectionPatch at 0x132ac6b20>,
  <matplotlib.patches.ConnectionPatch at 0x132ac6dc0>))
```


Depth (m)
0
20
40
60
80
100
120
140
292
294
296
298
300
Sea water temperature (K)
Depth (m)
0
2
299.2
299.4
299.6
299.8
300.0
300.2
Sea water temperature (K)

## 2.5 WSRA example: surface wave state

During EUREC4A/ATOMIC the P-3 flew with a Wide Swath Radar Altimeter (WSRA), a digital beam-forming radar altimeter operating at 16 GHz in the Ku band. It generates 80 narrow beams spread over 30 deg to produce a topographic map of the sea surface waves and their backscattered power. These measurements allow for continuous reporting of directional ocean wave spectra and quantities derived from this including significant wave height, sea surface mean square slope, and the height, wavelength, and direction of propagation of primary and secondary wave fields. WSRA measurements are processed by the private company ProSensing, which designed and built the instrument.

The WSRA also produces rainfall rate estimates from path-integrated attenuation but we won't look at those here.

The data are available through the EUREC4A intake catalog.

```
import xarray as xr
import numpy as np

import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book"])
import colorcet as cc
%matplotlib inline

import eurec4a
cat = eurec4a.get_intake_catalog()
```

Mapping takes quite some setup. Maybe we'll encapsulate this later but for now we repeat code in each notebook.

We'll select an hour's worth of observations from a single flight day. WSRA data are stored as "trajectories" - discrete times with associated positions and observations.

```
wsra_example = cat.P3.wsra["P3-0119"].to_dask().sel(trajectory=slice(0,293))
```

Now it's interesting to see how the wave slope (top panel) and the wave height (bottom) vary spatially on a given day.

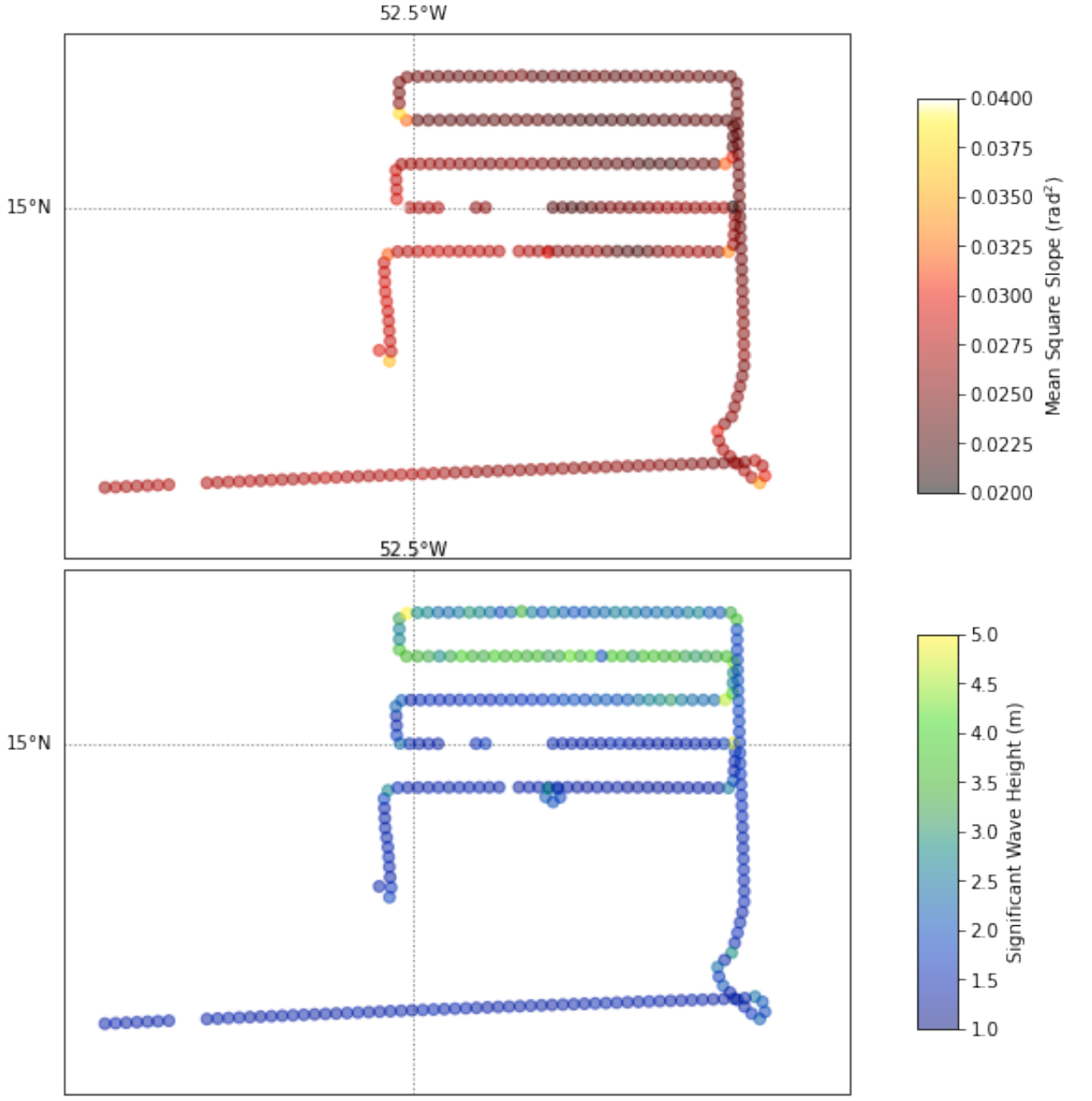
```
fig, (ax1, ax2) = plt.subplots(nrows=2, sharex = True, figsize = (12,8),
                               subplot_kw={'projection': ccrs.PlateCarree()})

#
# Mean square slope
#
ax_to_map(ax1, lon_e=-50, lon_w=-54.5, lat_s = 13, lat_n = 16)
add_gridlines(ax1)
pts = ax1.scatter(wsra_example.longitude,wsra_example.latitude,
           c=wsra_example.sea_surface_mean_square_slope_median,
           vmin = 0.02, vmax=0.04,
           cmap=cc.cm.fire,
           alpha=0.5,
           transform=ccrs.PlateCarree(),zorder=7)
fig.colorbar(pts, ax=ax1, shrink=0.75, aspect=10, label="Mean Square Slope (rad$^2$)")
#
# Significant wave height
#
ax_to_map(ax2, lon_e=-50, lon_w=-54.5, lat_s = 13, lat_n = 16)
add_gridlines(ax2)
pts = ax2.scatter(wsra_example.longitude,wsra_example.latitude,
           c=wsra_example.sea_surface_wave_significant_height,
           vmin = 1, vmax=5,
           cmap=cc.cm.bgy,
           alpha=0.5,
```

(continues on next page)

(continued from previous page)

```
            transform=ccrs.PlateCarree(),zorder=7)
fig.colorbar(pts, ax=ax2, shrink=0.75, aspect=10, label="Significant Wave Height (m)")
```

```
<matplotlib.colorbar.Colorbar at 0x13d9536a0>
```



CHAPTER

THREE

# METEOR

The RV Meteor's objective during the campaign was to measure along the same cross section of the HALO circle during the whole campaign to derive statistics of clouds inside this area. For this, the ship was steaming on a north-south transect between ~ 12° N and ~ 14° 30' N. It was equipped with a suite of in situ and remote sensing systems. For further details please refer to the official eurec4a website and the cruise report [MKR+20].


## 3.1 Reading cloud radar data

The high resolution data has about 500 MB per file, which when read in over a remote source can lead to long wait times. To reduce the wait times the data can be read in lazily using dask. Intake will this do this by default. Let's obtain the EUREC4A intake catalog:

```
import eurec4a
cat = eurec4a.get_intake_catalog()
```

### 3.1.1 Available products

The `LIMRAD94` cloud radar offers multiple products which can be accessed using names and additional parameters. Let's see which products and parameters are available for the cloud radar. For the parameters, we are also interested in their valid range:

```
for key, source in cat.Meteor.LIMRAD94.items():
    desc = source.describe()
    user_parameters = desc.get("user_parameters", [])
    if len(user_parameters) > 0:
        params = " (" + ", ".join(p["name"] for p in user_parameters) + ")"
    else:
```

(continues on next page)

(continued from previous page)

```
        params = ""
    print(f"{key}{params}: {desc['description']}")
    for parameter in user_parameters:
        print(f"    {parameter['name']}: {parameter['min']} ... {parameter['max']}␣
↪default: {parameter['default']}")
    print()
```

```
low_res (date): daily 30m 30s averaged radar reflectivity
    date: 2020-01-17 00:00:00 ... 2020-02-29 00:00:00 default: 2020-02-01 00:00:00

high_res (date, version): daily heave corrected original resolution cloudradar data
    date: 2020-01-17 00:00:00 ... 2020-02-29 00:00:00 default: 2020-02-01 00:00:00
    version: 1.0 ... 1.1 default: 1.1
```

### 3.1.2 Radar reflectivity

We'll have a look at the `high_res` data in version 1.1. We'll keep the default date for simplicity:

```
ds = cat.Meteor.LIMRAD94.high_res(version=1.1).to_dask()
ds
```

```
<xarray.Dataset>
Dimensions:                    (chirp: 3, range: 367, time: 44447)
Coordinates:
  * time                       (time) datetime64[ns] 2020-02-01T00:00:06.7000...
  * range                      (range) float32 313.0 335.4 ... 1.296e+04
Dimensions without coordinates: chirp
Data variables: (12/23)
    frequency                  float32 ...
    Numfft                     float32 ...
    altitude                   float32 ...
    NumSpectraAveraged         (chirp) float32 dask.array<chunksize=(3,), meta=np.
↪ndarray>
    latitude                   (time) float32 dask.array<chunksize=(1000,), meta=np.
↪ndarray>
    longitude                  (time) float32 dask.array<chunksize=(1000,), meta=np.
↪ndarray>
    ...                         ...
    width                      (time, range) float32 dask.array<chunksize=(1000, 367),
↪ meta=np.ndarray>
    ldr                        (time, range) float32 dask.array<chunksize=(1000, 367),
↪ meta=np.ndarray>
    kurt                       (time, range) float32 dask.array<chunksize=(1000, 367),
↪ meta=np.ndarray>
    Skew                       (time, range) float32 dask.array<chunksize=(1000, 367),
↪ meta=np.ndarray>
    heave_cor                  (time, range) float32 dask.array<chunksize=(1000, 367),
↪ meta=np.ndarray>
    heave_cor_bins             (time, range) float64 dask.array<chunksize=(1000, 367),
↪ meta=np.ndarray>
Attributes: (12/20)
    Conventions:    CF-1.8
    title:          LIMRAD94 (SLDR) Doppler Cloud Radar, calibrated file
    campaign_id:    EUREC4A
```

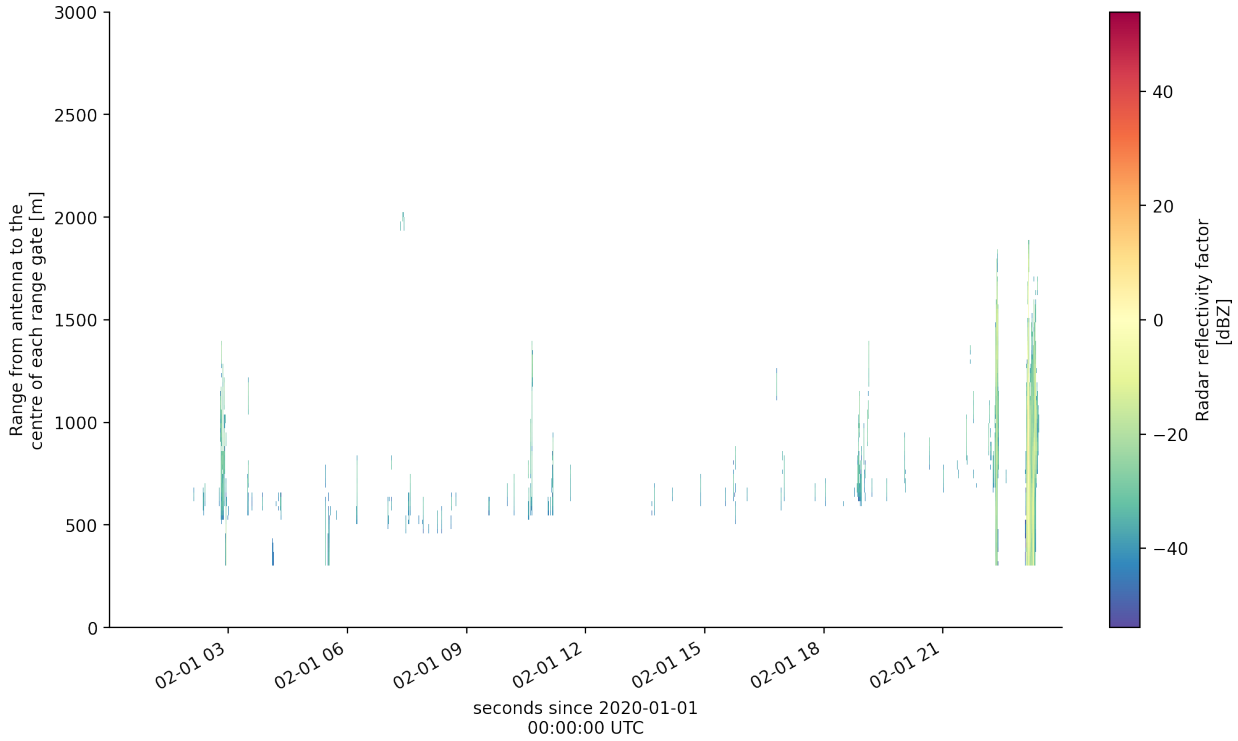
(continues on next page)

(continued from previous page)

```
    platform_id:    Meteor
    instrument_id:  LIMRAD94
    version_id:     v1.1
    ...             ...
    description:    Concatenated data files of LIMRAD 94GHz - FMCW Radar, fil...
    history:        Created Thu Feb 18 23:16:55 2021
    _FillValue:     -999.0
    day:            1
    month:          2
    year:           2020
```

Explore the dataset and choose the variable you want to work with. The variables are loaded lazily, i.e. only when their content is really required to complete the operation. An example which forces the data to load is plotting, in this case only the radar reflectivity will be loaded.

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")

ds.Zh.plot(x='time', cmap="Spectral_r")  # plot the variable with time as the x axis
plt.ylim(0, 3000);
```



CHAPTER

FOUR

# MERIAN

The Maria S. Merian is one of the research vessel deployed in the Atlantic Ocean during the EUREC4A campaign. The RV was equipped with stabilized Wband and micro rain radars, a water vapor raman lidar, a wind lidar, and a cloud kite for in situ measurements. Radiosondes were launched from the vessel during the entire campaign. All instruments and PIs are listed on the official eurec4a webpage.

During EUREC4A Maria S. Merian span a large region of ocean between 60 and 51 degrees W and 14 and 6 degrees N.

## 4.1 Cloud radar data collected on MS Merian

### 4.1.1 General information

From here you can access the W band radar data collected on the RV MSMerian. The radar is a frequency modulated continuous-wave (FMCW) 94 GHz dual polarization radar (manufactured by RPG GmbH) and the data are published on AERIS. The data presented here are corrected for ship motions. For more details on the algorithm applied for correction, please check the paper in preparation in ESSD.

It measured continously during the campaign, with a time resolution of 3 seconds and vertical range resolution varying with height between 7.5, 9.2 and 30 m. Here we provide you with some tools to plot and visualize the data.A **quicklook browser** is helping you to pick the case you are more interested in. If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset author that you can find here.

### Getting the data catalog

The python code below allows you to browse through the days avaiable

```
from datetime import datetime
import matplotlib.pyplot as plt
plt.style.use(["./mplstyle/book", "./mplstyle/wide"])
import numpy as np
import eurec4a
cat = eurec4a.get_intake_catalog()
```

### Getting the data from the MS-Merian ship

To visualize which datasets are available for the MS-Merian ship, type:

```
list(cat['MS-Merian'])
```

```
['MRR_PRO', 'track', 'FMCW94_RPG']
```

### Check available days of Wband radar data

To check which days are available from the Wband radar dataset, type:

```
for key, source in cat['MS-Merian']['FMCW94_RPG'].items():
    desc = source.describe()
    user_parameters = desc.get("user_parameters", [])
    if len(user_parameters) > 0:
        params = " (" + ", ".join(p["name"] for p in user_parameters) + ")"
    else:
        params = ""
    print(f"{key}{params}: {desc['description']}")
    for parameter in user_parameters:
        print(f"    {parameter['name']}: {parameter['min']} ... {parameter['max']}␣
↪default: {parameter['default']}")
```

```
motion_corrected (date): daily ship motion corrected Wband radar data
    date: 2020-01-19 00:00:00 ... 2020-02-19 14:00:00 default: 2020-01-19 20:00:00
```

### Selecting one day for plot/visualization

To work on a specific day, please provide the date as input in the form "yyyy-mm-dd hh:mm" as a field to date in the code below.

---

**Note:** The datetime type accepts multiple values: Python datetime, ISO8601 string, Unix timestamp int, "now" and "today".

---

```
date = '2020-01-27 13:00' # <--- provide the date here
ds = cat['MS-Merian']['FMCW94_RPG'].motion_corrected(date=date).to_dask()
ds
```

```
<xarray.Dataset>
Dimensions:                 (height: 550, time: 26719)
Coordinates:
  * time                    (time) datetime64[ns] 2020-01-27T00:00:01 ... 202...
  * height                  (height) float32 104.4 111.8 ... 9.949e+03 9.983e+03
    lat                     (time) float32 ...
    lon                     (time) float32 ...
Data variables: (12/13)
    rain_rate               (time) float32 ...
    relative_humidity       (time) float32 ...
    air_temperature         (time) float32 ...
    air_pressure            (time) float32 ...
    wind_speed              (time) float32 ...
    wind_direction          (time) float32 ...
    ...                      ...
    brightness_temperature  (time) float32 ...
    instrument              |S64 ...
    mean_doppler_velocity   (time, height) float32 ...
    radar_reflectivity      (time, height) float32 ...
    spectral_width          (time, height) float32 ...
    skewness                (time, height) float32 ...
Attributes: (12/30)
    CREATED_BY:         Claudia Acquistapace
    CREATED_ON:         2021-07-01 11:35:01.608208
    ORCID-AUTHOR:       Claudia Acquistapace: 0000-0002-1144-4753
    DOI:                10.25326/235 (https://doi.org/10.25326/235)
    FILL_VALUE:         NaN
    PI_NAME:            Claudia Acquistapace
    ...                 ...
    COMMENT:
    Conventions:        CF-1.8
    title:              daily w-band radar Doppler moments and surface weather...
    institution:        University of Cologne - Germany
    history:            source: wband data postprocessed\nprocessing: ship mot...
    featureType:        trajectoryProfile
```

### Plot some radar quantities

To create time/height plots for radar moments, pick a variable name having dimension `(time, height)` from the list of data variables and type the name in the code below. The example here is for mean Doppler velocity (corrected for ship motions). You can decide to either select the entire day or a given hour by providing `time_min` and `time_max`. To provide time_min and time_max, modify the string 'yyyy-mm-ddThh:mm:ss' Example: to select the entire day :

```
time_min = np.datetime64('2020-01-27T00:00:00')
time_max = np.datetime64('2020-01-27T23:59:59')
```

to select between 13:00 and 15:00 UTC:

```
time_min = np.datetime64('2020-01-27T13:00:00')
time_max = np.datetime64('2020-01-27T15:00:00')
```

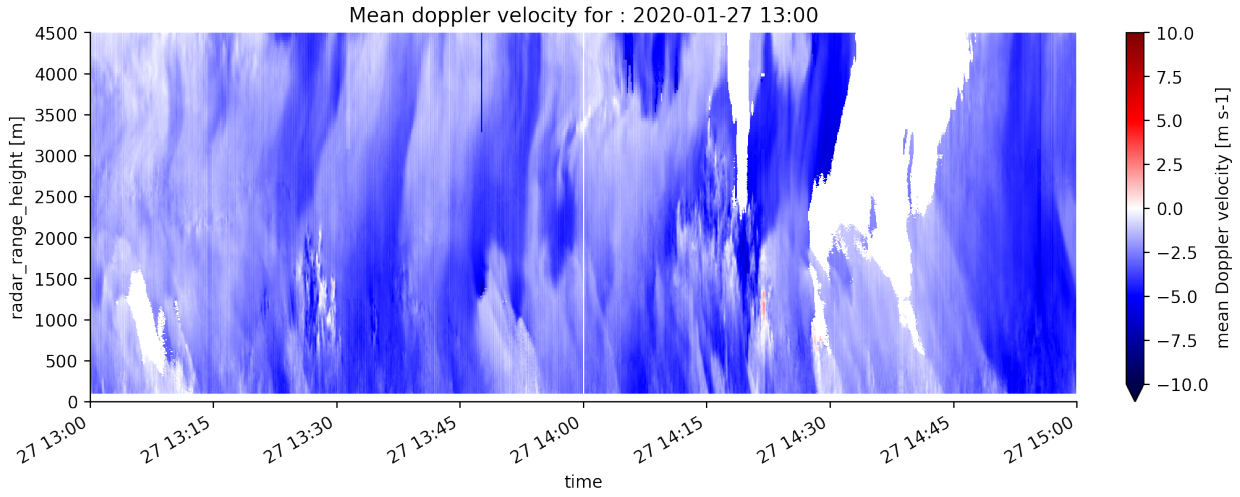
```
# set min and max time values for plotting along the x-axis
time_min = np.datetime64('2020-01-27T13:00:00') # insert the string value␣
↪corresponding to t_min
time_max = np.datetime64('2020-01-27T15:00:00') # insert the string value␣
↪corresponding to t_max
```

(continues on next page)

(continued from previous page)

```
# selecting subset of data
ds_sliced = ds.sel(time=slice(time_min, time_max))
```

```
fig, ax = plt.subplots()
# set here the variable from ds to plot, its color map and its min and max values
ds_sliced.mean_doppler_velocity.plot(x='time', y='height', cmap="seismic", vmin=-10.,␣
↪vmax=10.)
ax.set_title("Mean doppler velocity for : "+date)
ax.set_xlim(time_min, time_max)
ax.set_ylim(0, 4500);
```




### Check corresponding merian position in the selected hour

We use the function `track2layer` defined in the *interactive HALO tracks* chapter, and we provide as input the data for the time interval selected during the day

```
import ipyleaflet

# function to associate features to a given track
def track2layer(track, color="green", name=""):
    return ipyleaflet.Polyline(
        locations=np.stack([track.lat.values, track.lon.values], axis=1).tolist(),
        color=color,
        fill=True,
        weight=4,
        name=name
    )

# definition of the base map
m = ipyleaflet.Map(
    basemap=ipyleaflet.basemaps.Esri.NatGeoWorldMap,
    center=(11.3, -57), zoom=6)
m.add_layer(track2layer(ds_sliced, 'red', 'merian position'))  # adding layer of␣
↪merian position
m.add_control(ipyleaflet.ScaleControl(position='bottomleft'))
```

(continues on next page)

(continued from previous page)

```
m.add_control(ipyleaflet.LayersControl(position='topright'))
m.add_control(ipyleaflet.FullScreenControl())
display(m)
```

```
Map(center=[11.3, -57], controls=(ZoomControl(options=['position', 'zoom_in_text',
 ↪'zoom_in_title', 'zoom_out_…
```

CHAPTER

FIVE

# MULTI-PLATFORM DATASETS

Some datasets are synthesized from multiple platforms.

## 5.1 Dropsondes dataset JOANNE

The following script exemplifies the access and usage of dropsonde data measured during EUREC4A - ATOMIC.

More information on the dataset can be found at https://github.com/Geet-George/JOANNE/tree/master/joanne/Level_3#level—3. If you have questions or if you would like to use the data for a publication, please don't hesitate to get in contact with the dataset authors as stated in the dataset attributes `contact` and `author`.

### 5.1.1 Get data

- To load the data we first load the EUREC4A meta data catalogue. More information on the catalog can be found here.

```
import datetime
import numpy as np
import eurec4a
cat = eurec4a.get_intake_catalog()
```

- We can funrther specify the platform, instrument, if applicable dataset level or variable name, and pass it on to dask.

**Note:** Have a look at the attributes of the xarray dataset `ds` for all relevant information on the dataset, such as author, contact, or citation infromation.

```
ds = cat.dropsondes.JOANNE.level3.to_dask()
ds
```

```
<xarray.Dataset>
Dimensions:          (alt: 1001, nv: 2, sounding: 1064)
Coordinates:
  * alt              (alt) int64 0 10 20 30 40 50 ... 9960 9970 9980 9990 10000
    lat              (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
↪ndarray>
    launch_time      (sounding) datetime64[ns] dask.array<chunksize=(1064,), meta=np.
↪ndarray>
    lon              (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
↪ndarray>
```

(continues on next page)

(continued from previous page)

```
  * sounding         (sounding) int64 0 1 2 3 4 5 ... 1059 1060 1061 1062 1063
Dimensions without coordinates: nv
Data variables: (12/19)
    N_gps            (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    N_ptu            (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    PW               (sounding) float32 dask.array<chunksize=(1064,), meta=np.ndarray>
    alt_bnds         (alt, nv) float64 dask.array<chunksize=(1001, 2), meta=np.
 ↪ndarray>
    flight_height    (sounding) float32 dask.array<chunksize=(1064,), meta=np.ndarray>
    flight_lat       (sounding) float32 dask.array<chunksize=(1064,), meta=np.ndarray>
    ...               ...
    ta               (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    theta            (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    u                (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    v                (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    wdir             (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
    wspd             (sounding, alt) float32 dask.array<chunksize=(266, 501), meta=np.
 ↪ndarray>
Attributes: (12/13)
    ASPEN-version:           BatchAspen v3.4.3
    AVAPS-Software-version:  Version 4.1.2
    Conventions:             CF-1.8
    JOANNE-version:          0.7.0+2.g4a878b3.dirty
    author:                  Geet George
    author_email:            geet.george@mpimet.mpg.de
    ...                      ...
    creation_time:           2020-08-06 09:58:16.927582 UTC
    featureType:             trajectory
    instrument_id:           Vaisala RD-41
    product_id:              Level-3
    project_id:              JOANNE
    title:                   EUREC4A JOANNE Level-3
```

### 5.1.2 Load HALO flight phase information

All HALO flights were split up into flight phases or segments to allow for a precise selection in time and space of a circle or calibration pattern. For more information have a look at the respective github repository.

```
meta = eurec4a.get_flight_segments()
```

```
segments = [{**s,
             "platform_id": platform_id,
             "flight_id": flight_id
            }
            for platform_id, flights in meta.items()
            for flight_id, flight in flights.items()
            for s in flight["segments"]
           ]
```

```
segments_by_segment_id = {s["segment_id"]: s for s in segments}
```

```
segments_ordered_by_start_time = list(sorted(segments, key=lambda s: s["start"]))
```

We select all `dropsondes` with the quality flag `GOOD` from the first circle on February 5.

```
first_circle_Feb05 = [s["segment_id"]
                      for s in segments_ordered_by_start_time
                      if "circle" in s["kinds"]
                      and s["start"].date() == datetime.date(2020,2,5)
                      and s["platform_id"] == "HALO"
                     ][0]
first_circle_Feb05
```

```
'HALO-0205_c1'
```

```
dropsonde_ids = segments_by_segment_id[first_circle_Feb05]["dropsondes"]["GOOD"]
dropsonde_ids
```

```
['HALO-0205_s01',
 'HALO-0205_s02',
 'HALO-0205_s03',
 'HALO-0205_s04',
 'HALO-0205_s05',
 'HALO-0205_s06',
 'HALO-0205_s07',
 'HALO-0205_s08',
 'HALO-0205_s09',
 'HALO-0205_s10',
 'HALO-0205_s12']
```

We transfer the information from our flight segment selection to the dropsondes data in the xarray dataset.

```
from functools import reduce
mask_sondes_first_circle_Feb05 = reduce(lambda a, b: a | b, [ds.sonde_id==d
                                                             for d in dropsonde_ids])
ds_sondes_first_circle_Feb05 = ds.isel(sounding=mask_sondes_first_circle_Feb05)
```

### 5.1.3 Plots

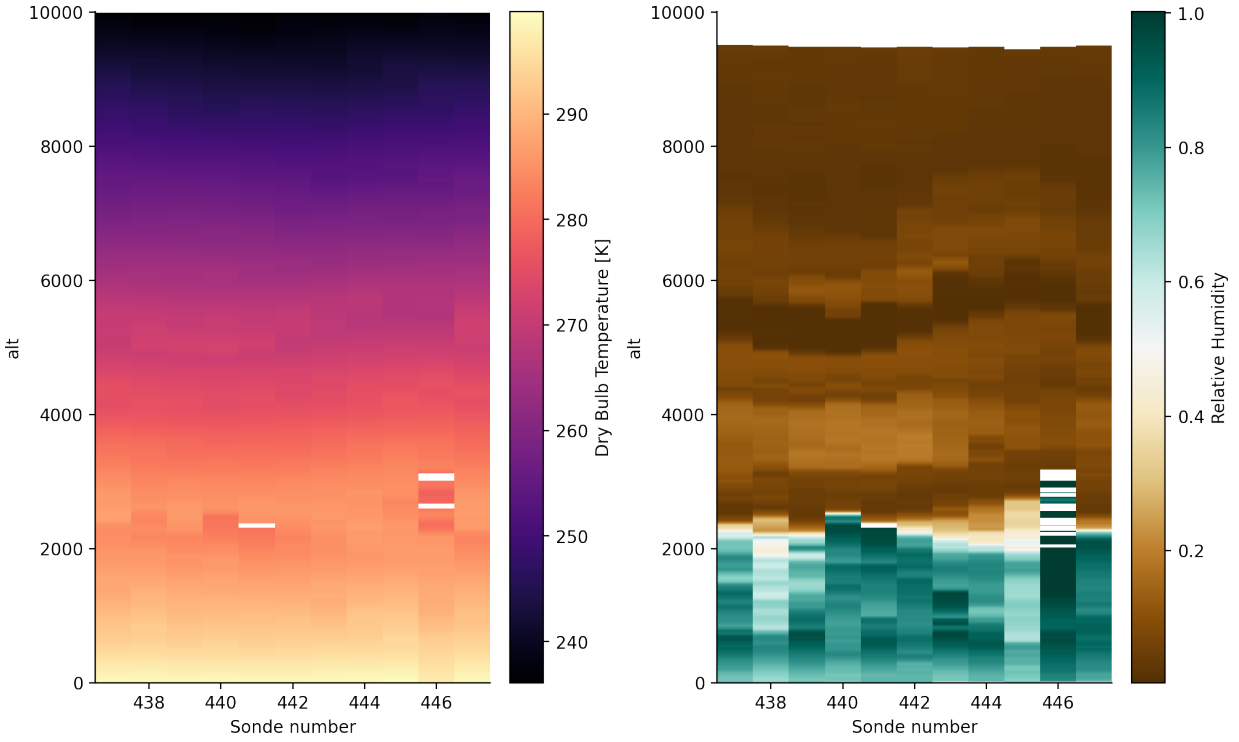
You can get a list of available variables in the dataset from `ds.variables.keys()`

**Note:** fetching the data and displaying it might take a few seconds

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use("./mplstyle/book")
```

### Temperature and relative humidity as stored in the xarray dataset

```
fig, (ax0, ax1) = plt.subplots(1, 2)
ds_sondes_first_circle_Feb05.ta.transpose("alt", "sounding").plot(ax=ax0, cmap="magma
↪")
ds_sondes_first_circle_Feb05.rh.transpose("alt", "sounding").plot(ax=ax1, cmap="BrBG")
None
```




### Temperature and relative humidity profiles.

The temperature profiles are colored according to their launch time, while relative humidity profiles are colored according to their integrated water vapour in the measured column, i.e. precipitable water.

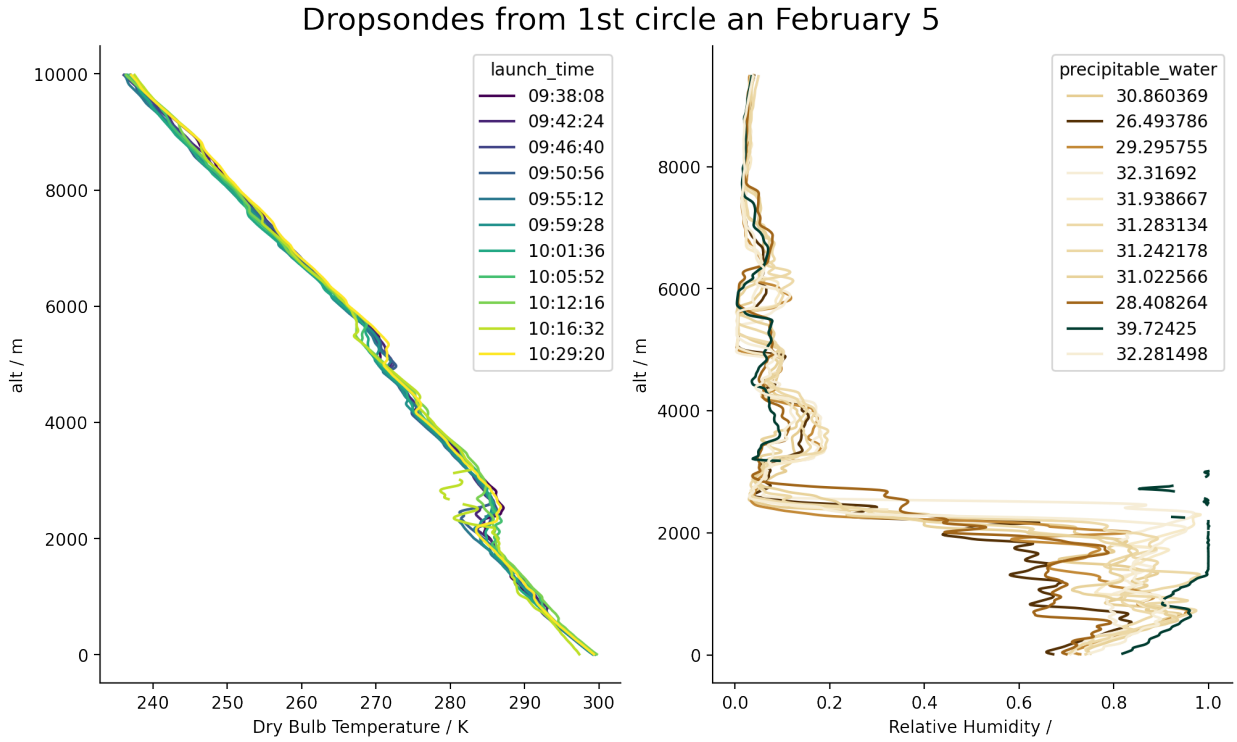
```
def dt64_to_dt(dt64):
    epoch = np.datetime64('1970-01-01T00:00:00')
    second = np.timedelta64(1, 's')
    return datetime.datetime.utcfromtimestamp(int((dt64 - epoch) / second))
```

```
fig, (ax0, ax1) = plt.subplots(1, 2)

y = ds_sondes_first_circle_Feb05.alt

x0 = ds_sondes_first_circle_Feb05.ta.transpose("alt", "sounding")
ax0.set_prop_cycle(color=plt.cm.viridis(np.linspace(0, 1, len(dropsonde_ids))))
ax0.plot(x0, y.data[:, np.newaxis])
ax0.set_xlabel(f"{x0.long_name} / {x0.units}")
ax0.set_ylabel(f"{y.name} / m")
ax0.legend([dt64_to_dt(d).strftime("%H:%M:%S")
            for d in ds_sondes_first_circle_Feb05.launch_time],
```

(continues on next page)

(continued from previous page)

```
            title=x0.launch_time.name)

x1 = ds_sondes_first_circle_Feb05.rh.transpose("alt", "sounding")
c = ds_sondes_first_circle_Feb05.PW

ax1.set_prop_cycle(color=plt.cm.BrBG([int(i) for i in (c - c.min())#cividis_r
                                      / (c - c.min()).max() * 255]))
p = ax1.plot(x1, y.data[:, np.newaxis])
ax1.set_xlabel(f"{x1.long_name} / {x1.units}")
ax1.set_ylabel(f"{y.name} / m")
ax1.legend(ds_sondes_first_circle_Feb05.PW.values,
            title=ds_sondes_first_circle_Feb05.PW.standard_name)

fig.suptitle('Dropsondes from 1st circle an February 5', fontsize=18)
None
```




### wind speed variations throughout February 5

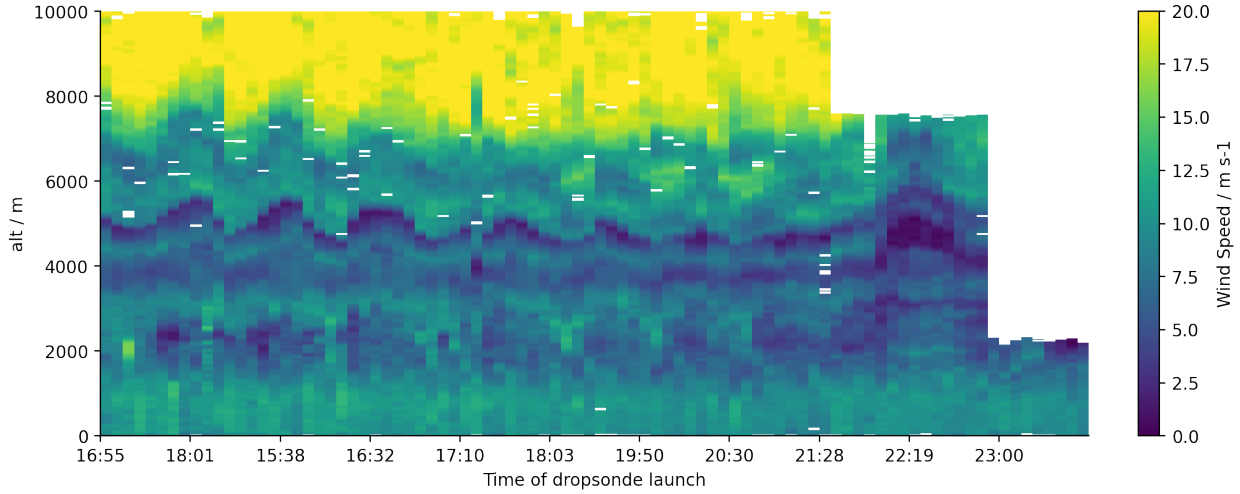
```
mask_sondes_Feb05 = ds.launch_time.astype("<M8[D]") == np.datetime64("2020-02-05")
ds_sondes_Feb05 = ds.isel(sounding=mask_sondes_Feb05)
```

```
with plt.style.context("mplstyle/wide"):
    fig, ax = plt.subplots()
    p = ax.pcolormesh(ds_sondes_Feb05.wspd.transpose("alt", "sounding"), vmin=0,␣
↪vmax=20)
    plt.colorbar(p, ax=ax, label=f"{ds_sondes_Feb05.wspd.long_name} / {ds_sondes_
↪Feb05.wspd.units}")
    xticks = np.arange(0, ds_sondes_Feb05.sounding.size, int(ds_sondes_Feb05.sounding.
↪size/10))
```

(continues on next page)

(continued from previous page)

```
    ax.set_xticks(xticks)
    ax.set_xticklabels([dt64_to_dt(t).strftime("%H:%M") for t in ds.launch_time.
↪values[xticks]])
    ax.set_xlabel(f"{ds_sondes_Feb05.launch_time.long_name}")
    yticks = np.arange(0, ds_sondes_Feb05.alt.size, int(ds_sondes_Feb05.alt.size/5))
    ax.set_yticks(yticks)
    ax.set_yticklabels(ds_sondes_Feb05.alt.values[yticks])
    ax.set_ylabel(f"{ds_sondes_Feb05.alt.name} / m")
    None
```



CHAPTER

SIX

# TOOLBOX

Further dataset operations, such as including multiple datasets or using the flight phase separation information.

## 6.1 How to work with flight phase segmentation files

```
import datetime
import eurec4a
```

```
meta = eurec4a.get_flight_segments()
meta.keys()
```

```
dict_keys(['HALO', 'P3'])
```

### 6.1.1 1. map `segment_id`s to `segment`s

This can be useful for further queries based on specific segment properties. In addition to the original properties in each `segment`, the `platform_id` and `flight_id` are also stored.

```
segments = [{**s,
             "platform_id": platform_id,
             "flight_id": flight_id
            }
            for platform_id, flights in meta.items()
            for flight_id, flight in flights.items()
            for s in flight["segments"]
           ]
```

```
segments_by_segment_id = {s["segment_id"]: s for s in segments}
```

### 6.1.2 2. list `flight_ids`

```
flight_ids = [flight_id
              for platform_id, flights in meta.items()
              for flight_id, flight in flights.items()
              ]
flight_ids
```

```
['HALO-0119',
 'HALO-0122',
 'HALO-0124',
 'HALO-0126',
 'HALO-0128',
 'HALO-0130',
 'HALO-0131',
 'HALO-0202',
 'HALO-0205',
 'HALO-0207',
 'HALO-0209',
 'HALO-0211',
 'HALO-0213',
 'HALO-0215',
 'HALO-0218',
 'P3-0117',
 'P3-0119',
 'P3-0123',
 'P3-0124',
 'P3-0131',
 'P3-0203',
 'P3-0204',
 'P3-0205',
 'P3-0209',
 'P3-0210',
 'P3-0211']
```

List `flight_id` for a specified day, here February 5

```
flight_id = [flight_id
             for platform_id, flights in meta.items()
             for flight_id, flight in flights.items()
             if flight["date"] == datetime.date(2020,2,5)
             ]
flight_id
```

```
['HALO-0205', 'P3-0205']
```

### 6.1.3 3. list flight segmentation `kinds`

A segment is an object which includes at minimum a `segment_id`, `name`, `start` and `end` time.

```
kinds = set(k for s in segments for k in s["kinds"])
kinds
```

```
{'axbt',
 'baccardi_calibration',
 'circle',
 'circle_break',
 'circling',
 'cloud',
 'clover_leg',
 'clover_turn',
 'lidar_leg',
 'profile',
 'radar_calibration_tilted',
 'radar_calibration_wiggle',
 'straight_leg',
 'super_curtain',
 'transit'}
```

### 6.1.4 4. List of common properties in all `segments`

```
set.intersection(*(set(s.keys()) for s in segments))
```

```
{'dropsondes',
 'end',
 'flight_id',
 'irregularities',
 'kinds',
 'name',
 'platform_id',
 'segment_id',
 'start'}
```

```
segment_ids_by_kind = {kind: [segment["segment_id"]
                              for segment in segments
                              if kind in segment["kinds"]]
    for kind in kinds
}
```

### 6.1.5 5. Further random examples:

- total number of all circles flown during EUREC4A / ATOMIC

```
len(segment_ids_by_kind["circle"])
```

```
89
```

- How much time did HALO and P3 spend circling?

```
circling_time = sum([s["end"] - s["start"]
                     for s in segments
                     if "circling" in s["kinds"]
                    ], datetime.timedelta())
circling_time
```

```
datetime.timedelta(days=3, seconds=29188)
```

- get `segment_id` for all circles on a given day sorted by the start time, here February 5

```
segments_ordered_by_start_time = list(sorted(segments, key=lambda s: s["start"]))
```

```
circles_Feb05 = [s["segment_id"]
                 for s in segments_ordered_by_start_time
                 if "circle" in s["kinds"]
                 and s["start"].date() == datetime.date(2020,2,5)
                 and s["platform_id"] == "HALO"
                ]
circles_Feb05
```

```
['HALO-0205_c1',
 'HALO-0205_c2',
 'HALO-0205_c3',
 'HALO-0205_c4',
 'HALO-0205_c5',
 'HALO-0205_c6']
```

- select all `dropsondes` with the quality flag `GOOD` from the first circle on February 5.

```
circles_Feb05[0]
```

```
'HALO-0205_c1'
```

```
segments_by_segment_id[circles_Feb05[0]]["dropsondes"]["GOOD"]
```

```
['HALO-0205_s01',
 'HALO-0205_s02',
 'HALO-0205_s03',
 'HALO-0205_s04',
 'HALO-0205_s05',
 'HALO-0205_s06',
 'HALO-0205_s07',
 'HALO-0205_s08',
 'HALO-0205_s09',
 'HALO-0205_s10',
 'HALO-0205_s12']
```

## 6.2 Show the intake catalog

The eurec4a intake catalog is maintained on github at eurec4a/eurec4a-intake. The structure of the files however does not represent the structure of the catalog. In order to get a quick overview about its contents, here's a little script which prints out the current catalog tree.

```
import eurec4a
```

```
cat = eurec4a.get_intake_catalog()
```

```
def tree(cat, level=0):
    prefix = " " * (3*level)
    try:
        for child in list(cat):
            parameters = [p["name"] for p in cat[child].describe().get("user_
 ↪parameters", [])]
            if len(parameters) > 0:
                parameter_str = " (" + ", ".join(parameters) + ")"
            else:
                parameter_str = ""
            print(prefix + str(child) + parameter_str)
            tree(cat[child], level+1)
    except:
        pass
```

```
tree(cat)
```

```
radiosondes
```

```
   atalante_meteomodem
   atalante_vaisala
   bco
   meteor
   ms_merian
   ronbrown
barbados
```

```
   bco
```

```
      CORAL_LIDAR (version, date, dt, content_type)
Atalante
```

```
   track
ATR
```

```
   track
BOREAL
```

```
   track
CU-RAAVEN
```

```
   track
Caravela
```

```
   track
HALO
```

```
   BAHAMAS
```

```
      QL
```

```
         HALO-0119
         HALO-0122
         HALO-0124
         HALO-0126
         HALO-0128
         HALO-0130
         HALO-0131
         HALO-0202
         HALO-0205
         HALO-0207
         HALO-0209
         HALO-0211
         HALO-0213
         HALO-0215
         HALO-0218
      PositionAttitude
```

```
         HALO-0115
         HALO-0118
         HALO-0119
         HALO-0122
         HALO-0124
         HALO-0126
         HALO-0128
         HALO-0130
         HALO-0131
         HALO-0202
         HALO-0205
         HALO-0207
         HALO-0209
         HALO-0211
         HALO-0213
         HALO-0215
         HALO-0218
   specMACS
```

```
      cloudmaskSWIR
```

```
         HALO-0119
         HALO-0122
         HALO-0124
         HALO-0126
         HALO-0128
         HALO-0130
         HALO-0131
         HALO-0202
         HALO-0205
         HALO-0207
```

(continues on next page)

(continued from previous page)

```
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
    experimental
```

```
        points
```

```
           HALO-0205_sl1
           HALO-0205_sl2
           HALO-0205_sl3
           HALO-0205_sl4
        centroids
```

```
           HALO-0205_sl1
           HALO-0205_sl2
           HALO-0205_sl3
           HALO-0205_sl4
        mesh
```

```
           HALO-0205_sl1
           HALO-0205_sl2
           HALO-0205_sl3
           HALO-0205_sl4
  UNIFIED
```

```
    dropsondes
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
    HAMPradar
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
```

(continues on next page)

(continued from previous page)

```
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
    HAMPradiometer
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
    BAHAMAS
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
    HAMPradiometer_cloudmask
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
```

(continues on next page)

(continued from previous page)

```
        HALO-0213
        HALO-0215
        HALO-0218
    HAMPradar_cloudmask
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
    HAMPradiometer_retrievals
```

```
        HALO-0119
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
        HALO-0218
  SMART
```

```
    spectral_irradiances
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
  VELOX
```

```
    cloudmask
```

```
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
KT19
```

```
    cloudmask
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
WALES
```

```
    cloudparameter
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
    wv
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
```

(continues on next page)

(continued from previous page)

```
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
    adepg
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
    bsri
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
    bsrg
```

```
        HALO-0122
        HALO-0124
        HALO-0126
        HALO-0128
        HALO-0130
        HALO-0131
        HALO-0202
        HALO-0205
        HALO-0207
        HALO-0209
        HALO-0211
        HALO-0213
        HALO-0215
  BACARDI
```

```
      irradiances
```

```
         HALO-0119
         HALO-0122
         HALO-0124
         HALO-0126
         HALO-0128
         HALO-0130
         HALO-0131
         HALO-0202
         HALO-0205
         HALO-0207
         HALO-0209
         HALO-0211
         HALO-0213
         HALO-0215
         HALO-0218
   track
Humpback
```

```
   track
IFM03
```

```
   track
IFM09
```

```
   track
IFM12
```

```
   track
Kracken
```

```
   track
Melonhead
```

```
   track
mini-MPCK
```

```
   track
MPCK-plus
```

```
   track
MS-Merian
```

```
   MRR_PRO
```

```
      motion_corrected (date)
   track
   FMCW94_RPG
```

```
      motion_corrected (date)
Omura
```

```
   track
P3
   AXBT
```

```
      Level_3
   flight_level
```

```
      P3-0117
      P3-0119
      P3-0123
      P3-0124
      P3-0131
      P3-0203
      P3-0204
      P3-0205
      P3-0209
      P3-0210
      P3-0211
   isotope_analyzer
```

```
      water_vapor_1hz
```

```
         P3-0117
         P3-0119
         P3-0123
         P3-0124
         P3-0131
         P3-0203
         P3-0204
         P3-0205
         P3-0209
         P3-0210
         P3-0211
      water_vapor_5hz
```

```
         P3-0117
         P3-0119
         P3-0123
         P3-0124
         P3-0131
         P3-0203
         P3-0204
         P3-0205
         P3-0209
         P3-0210
         P3-0211
   remote_sensing
      P3-0117
      P3-0119
      P3-0123
      P3-0124
      P3-0131
      P3-0203
      P3-0204
      P3-0205
```

(continues on next page)

(continued from previous page)

```
      P3-0209
      P3-0210
      P3-0211
   track
   w_band_radar
```

```
      P3-0117
      P3-0119
      P3-0123
      P3-0124
      P3-0131
      P3-0203
      P3-0204
      P3-0205
      P3-0209
      P3-0210
      P3-0211
   wsra
      P3-0117
      P3-0119
      P3-0123
      P3-0124
      P3-0131
      P3-0203
      P3-0204
      P3-0205
      P3-0209
      P3-0210
      P3-0211
QuadCopter
```

```
   track
RonBrown
```

```
   track
SD-1026
```

```
   track
   1min
   5min
SD-1060
```

```
   track
   1min
   5min
SD-1061
```

```
   track
   1min
   5min
SD-1063
```

```
   track
SD-1064
```

```
   track
Skywalker07
```

```
   track
Skywalker10
```

```
   track
Skywalker12
```

```
   track
SVP-B-4101696
```

```
   track
SVP-B-4101697
```

```
   track
SVP-B-4101698
```

```
   track
SVP-B-4101699
```

```
   track
SVP-B-4101780
```

```
   track
SVP-BRST-4402505
```

```
   track
SVP-BRST-4402506
```

```
   track
SVP-BRST-4402507
```

```
   track
SVP-BRST-4402508
```

```
   track
SVP-BRST-6203717
```

```
   track
SVP-BS-4101757
```

```
   track
SVP-BS-4101758
```

```
   track
SVP-BSW-3101569
```

```
   track
SVP-BSW-3101570
```

```
   track
SVP-BSW-3101571
```

```
   track
SVP-BSW-3101572
```

```
   track
SVP-BSW-3101573
```

```
   track
SVP-BSW-3101574
```

```
   track
SVP-BSW-3101575
```

```
   track
SVP-BSW-3101576
```

```
   track
SVP-BSW-3101577
```

```
   track
SVP-BSW-3101578
```

```
   track
SWIFT16
```

```
   track
   all
SWIFT17
```

```
   track
   all
SWIFT22
```

```
   track
   all
SWIFT23
```

```
   track
   all
SWIFT24
```

```
   track
   all
SWIFT25
```

```
   track
   all
TO
```

```
   track
WG245
```

```
   track
WG247
```

```
   track
dropsondes
```

```
   JOANNE
```

```
      level3
satellites
```

```
   GOES16
```

```
      latlongrid (resolution, channel, date)
radiative_profiles
```

```
   clear_sky (version)
Meteor
```

```
   LIMRAD94
```

```
      low_res (date)
      high_res (date, version)
   track
```

There's also a graphical user interface (GUI) implemented in intake. The GUI additionally requires the `panel` python package and it interactively queries the catalog, so it doesn't work nicely in a book. This is why the following lines of code are commented out, but they can be used in an interactive notebook.

```
#import intake
#intake.gui.add(cat)
#intake.gui.panel
```

## 6.3 running How to EUREC4A locally

There are multiple options to run the Code examples from this book locally. In any case, it will involve the following steps:

- install Python (we assume hat this is already done, have a look at python.org if you want to get it)
- obtain the code from the book
- install all required dependencies
- run the code

You can decide between the *quick an dirty* method and the method *using `git`*, which will also set you up to contribute to the book.

### 6.3.1 quick and dirty

If you just like to run the code of a single notebook and don't care to much about the details, the quickest option might be to download the chapter you are viewing as an ipython notebook (`.ipynb`) via the download button () on the top right of the page. If you don't see the `.ipynb` option here, that's because the source of the page can not be interpreted as a notebook and thus is not available for direct execution.

If you would just run the downloaded code, the chance is high that some required libraries are not yet installed on your system. You can either do try and error to find out which libraries are required for the chapter you downloaded, or you can simply installed all requirements for the entire book by running the following command on your command line:

Using pip

```
pip install jupyter
pip install -r https://raw.githubusercontent.com/eurec4a/how_to_eurec4a/master/
↪requirements.txt
```

This won't work with any notebooks that use `cartopy` to make maps, `pip` does not manage their dependencies well.

Using conda

```
wget https://raw.githubusercontent.com/eurec4a/how_to_eurec4a/master/requirements.txt
conda create -f requirements.txt
conda activate how_to_eurec4a
```

This creates a conda environment called `how_to_eurec4a` which contains all dependencies including `cartopy`

Afterwards, you can start a local notebook server (either `jupyter notebook` or `jupyter lab`) and run and modify the chapter locally.

---

**Note:** Handling requirements in this project is not entirely straightforward, as the requirements strongly depend on which datasets are used. We use intake catalogs to describe how a dataset can be accessed which simplifies a lot of things. But as a consequence the set of required libraries is not only determined by the code in this repository, but also by the entries in the catalog.

---

### 6.3.2 via git

If you like to do it more properly, you can also clone the repository via git. Depending on if you have SSH public key authentication set up or not, you can do this via SSH or HTTPS:

#### SSH

```
git clone git@github.com:eurec4a/how_to_eurec4a.git
```

### HTTPS

```
git clone https://github.com/eurec4a/how_to_eurec4a
```

This will create a local copy of the entire book repository in a newly created local folder `how_to_eurec4a`. Please change into this directory.

---

**Maybe use a virtual environment**

If you use pip you might want to use a virtual environment for the book if you like to keep the required libraries in a confined place, but it is entirely optional and up to your preferences. There are many options to do so and virtualenv is one of them. Using virtualenv, you could create and activate an environment like:

```
virtualenv venv
. venv/bin/activate
```

and the continue normally.

---

You'll have to install the dependencies as above, but as you already have all the files on your machine, you can also install it directly via:

Using pip

```
pip install jupyter
pip install -r requirements.txt
```

This won't work with any notebooks that use `cartopy` to make maps, `pip` does not manage their dependencies well.

Using conda

```
conda create -f requirements.txt
conda activate how_to_eurec4a
```

This creates a conda environment called `how_to_eurec4a` which contains all dependencies including `cartopy`

Depending on your needs, you can continue using *interactive notebooks* or *compile the entire book*.

---

**About MyST notebooks.**

Internally, the book does not use the `.ipynb` file format as it is not well-suited for version control. Instead, we use Markedly Structured Text or MyST which is a variant of Markdown, optimized to contain executable code cells as in notebooks. The extension of MyST files is `.md`. In contrast to notebooks, these files **do not** contain generated cell outputs, so you'll actually have to run the files in order to see anything.

`jupytext` is used to convert between MyST and `ipynb` formats. It is installed through the `requirements.txt` and should register itself with `jupyter`, so that you can open the files as if they where `ipynb` files. If that does not work, please have a look at the installation instructions of `jupytext`.

---

### interactive

`jupyter` itself is not installed by the requirements file. If you're using `pip` you might want to install it as well:

```
pip install jupyter
```

Once everything is set up, you can either start your notebook server:

… either using classical notebooks

```
jupyter notebook
```

… or using jupyter lab

```
jupyter lab
```

### compile the book

You can also execute `jupyter-book` directly via:

```
jb build how_to_eurec4a
```

which itself will run all code cells and output the results as HTML pages in a newly created folder. This variant is especially useful if you like to work directly on the MyST files (see note below) using a text editor and should be done every time before you submit new changes to the book.

### adding new articles

Articles are generated from markdown files within the `how_to_eurec4a` folder of the git repository. The following instructions assume that you are working in that directory.

### text articles

Text articles can be created by adding standard markdown files using your favourite text editor.

### executable notebook

If you want to add a new notebook, you can either start out from an existing **MyST** Markdown file by copying and modifying it, or you can create a new one from the jupyter notebook menu using `File` > `New Text Notebook` > `MyST Markdown`. If you already have an existing ipython notebook and want to convert it to **MyST**, you can do this by running

```
jupytext --to myst your_notebook.ipynb
```

This will create a new markdown file named `your_notebook.md`. After conversion, the `ipynb` file is not needed anymore and it should not be committed into the repository.

#### entry in the table of contents

After preparing your article or notebook, you'll have to add it into the table of contents, such that it will actually show up in the compiled book. You can do this by modifying `_toc.yml`, where you can add the articles file name without suffix.

#### compile to PDF

If you've got a LaTeX environment available (e.g. TeXLive), you can also compile the book into a single PDF file. To do so, just run:

```
jb build how_to_eurec4a --builder pdflatex
```

## 6.4 The issue with netCDF data types

**TL; DR**

If you want to be on the safe side, use only `SHORT`, `INT`, `FLOAT`, `DOUBLE`, `STRING` or array of `CHAR` as data types. In particular, **don't** use single byte integers, any unsigned integer and no 64 bit integer.

The data format netCDF which is used in many places throughout our community has evolved over time. During this evolution, the basic data types representable within the netCDF data format have been expanded. Currently we are at netCDF version 4 with HDF5 as a storage backend, which offers a lot more flexibility compared to previous versions of netCDF including many more data types. One **could** assume that this is all fine and we can use these data types freely, but here we are unlucky and there is more to keep in mind.

In order to paint the full picture, we'll have to take a little detour and think about what *netCDF actually is* or even more what we think (or probably should think) about when talking about netCDF. As we'll see, this is not something which can be answered in a straight forward manner, so it is fair to ask *if we should care at all*. Finally we'll have a look at *an experiment* which shows plenty of ways in which different data types may fail around netCDF and which types seem to be fine.

### 6.4.1 What is netCDF?

NetCDF is more than only the "netCDF4 on HDF5" storage format. This is not only a bold statement, but is actually asked and discussed by the netCDF developers (What is netCDF?, Jan 2020). This chapter is not exactly about the same topic in the referenced issue, but still has similar roots. The relevant part for this discussion is that there is at least netCDF, the **data model** and netCDF, the **persistence format**. It is also recognized that there are multiple persistence formats (netCDF3, netCDF4/HDF5, zarr) for the same data model and there are translation layers such as OPeNDAP which can be used as a transport mechanism between the computer on which data is stored and the computer on which data is analyzed. We further look at the relevance of the netCDF data model and netCDF persistence format for the EUREC4A datasets.

### The data model(s)

There are actually a few different ideas of data models around, which share some general ideas. Maybe the most illustrative way to show this general idea is an extended version of xarray's logo, but the underlying ideas are of course applicable independently of xarray or even Python.

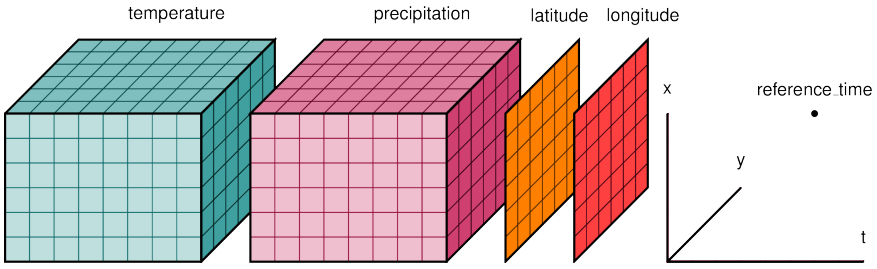



Fig. 6.1: Dataset diagram (From the xarray documentation)

A dataset is a collection of *variables* and *dimensions*. Variables are multi-dimensional arrays and they can share dimensions (i.e. to express that `temperature` and `precipitation` are defined at the same locations, the three axes of the corresponding arrays should be identified with the same dimension label, that is e.g. `x`, `y` and `z`). Some variables may be promoted into a *coordinate*, which does not require to store it differently, but it changes the interpretation of the data. In the example above, `latitude` and `longitude` would likely be identified as coordinates. One would typically use *coordinate* values to index into some data or to label the ticks on a plot axis, while a normal *variable* would usually be used to define a line or a color within a plot. There may be *variables* which should be *coordinates* only for a specific use case and vice versa. It is useful to distinguish between *variables* and *coordinates* within the metadata of a dataset, but the representation of the actual data may be the same for both, *variables* and *coordinates*. In addition to the existence of *coordinates*, there are more pieces of valuable additional information (think of units, coordinate reference systems, author information etc…). Thus we usually want those extra bits to datasets and variables, which can be done by the use of *attributes*.

It turns out that many datasets in the EUREC4A context can be represented very well in this model. I assume that in many cases when we talk about that a dataset being available in netCDF, what we really care about is that the dataset is organized along this general structure and thus can be accesses *as if it were netCDF*.

### The storage and transport formats

#### netCDF

**Note:** NetCDF defines two internal "data models", thus they could go into the section above. However, those models are also loosely tied to their backend storage formats and thus, they might fit better in the current paragraph.

In order to put the high level idea from above into a data structure which can be stored on disk, netCDF defines two low level data models where the Classic data model came first and the Enhanced data model evolved from it. Both models care about *variables*, *dimensions* and *attributes* as introduced above.

The **classic** model knows about `CHAR`, `BYTE`, `SHORT`, `INT`, `FLOAT` and `DOUBLE` data types which are all **signed**. There may be at most one unlimited (i.e. resizeable) dimension per variable.

The **enhanced** model is based on the classic model and adds `INT64` and `STRING` to the mix as well as **unsigned** variants of all integral types. Further additions of the enhanced model are groups (which are like a dataset within a dataset) and user defined types. There may be any number of unlimited dimensions per variable.

Those two data models are closely tied to the netCDF3 and netCDF4 storage formats.

- netCDF3 is based on the *classic* data model and defines a storage format which is custom to netCDF.
- netCDF4 is based on the *enhanced* data model and uses HDF5 as its backend storage format. HDF5 enables further enhancements like internal data compression.

**Note:** HDF5 features `external` and `virtual` datasets which can be compatible to netCDF4 but can not be written using netCDF4 API.

Both storage formats store all data in a single file, chunking of the dataset is done internally. If the dataset should be accessed from a remote location, usually the entire dataset must be fetched before accessing the data.

In Jan 2019 HTTP byte-range enabled partial reads from remote datasets have been added to netCDF-c, but as stated in the pull request, it is not meant for production use:

> Note that this is not intended as a true production capability because, as is known, this kind of access to can be quite slow. In addition, the byte-range IO drivers do not currently do any sort of optimization or caching.

Thus, if a dataset should be accessed efficiently from a remote location, something different must be added, which is what OPeNDAP is for.

### OPeNDAP

**Note:** Datacenters used by EUREC4A provide data via OPeNDAP include Aeris, NOAA PSL, NOAA NCEI, MPIM RAMADDA and the specMACS macsServer. The How to EUREC4A book makes use of this service to access the various datasets.

OPeNDAP is a network protocol to access scientific data via network. In particular, it uses HTTP as a transport mechanism and defines a way to request subsets of a dataset which is formed along the data model as described above. OPeNDAP is thus the go-to method if an existing netCDF dataset should be made available remotely. In the context of EUREC4A, most datacenters provide access to uploaded datasets via OPeNDAP.

OPeNDAP is specified in two versions, namely version 2 and version 4. However, DAP4 is still a draft since 2004 and the only widely supported version of OPeNDAP is version 2.

The data model of OPeNDAP v2 is slightly different from the data model in netCDF. Regarding data types, OPeNDAP v2 defines `Byte`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Float32`, `Float64`, `String`, `URL`.

As opposed to the `BYTE` in netCDF Classic, the `Byte` in OPeNDAP is **unsigned**. This makes byte types of netCDF Classic and OPeNDAP incompatible, but as noted in the *summary*, there exists a hack which tries to circumvent that. One could assume that `UBYTE` of netCDF Enhanced would fit to this `Byte`, but there are issues as well which are confirmed in *the experiment*.

Furthermore, there is no `CHAR`, so whenever a netCDF dataset containing text as a sequence of `CHAR` is encountered by an OPeNDAP server, this will be converted to an OPeNDAP `String`. A consequence of this is that `CHAR`s and `STRING`s can not be distinguished when transferred over OPeNDAP and thus may be converted into each other after one cycle through netCDF → OPeNDAP → netCDF. This behaviour is not (yet) studied in this text.

#### zarr

**to do**

zarr should be discussed in this place.

### 6.4.2 Why should we care?

Each of the formats has distinct advantages and disadvantages:

- **netCDF** is a compact format, the whole dataset can be stored in a single file which can be transferred from one place to another and everything is contained. This is great for storing data and to send around smaller datasets, but retrieving only a subset of a dataset from a remote server is not possible. Libraries which can open netCDF files usually do not handle HTTP, so the dataset needs to be stored in a local directory to use it.
- **OPeNDAP** is a network protocol based on HTTP which has been made exactly for the purpose of requesting subsets of a dataset from a remote server. As it is only a network protocol, it can not be used to store the data, so every dataset which should be provided via OPeNDAP must be stored in another format (like netCDF) or computed on the fly. Support for reading OPeNDAP is build into netCDF-c.
- **zarr** is a storage format which is spread out into several files in a defined directory structure. This makes it a little harder to pass around, but this chunked structure makes remote access exceptionally simple and fast. Chelle Gentemann has prepared an impressive demonstration on this topic.

**Note:** zarr can be used as a single file by using a zip file as its directory structure.

So, depending on the use case, different formats are optimal and none of them supports everything:

| format | for storage | sending around | for remote access | widely supported |
|---|---|---|---|---|
| netCDF | ✓ | ✓ | ✗, works via OPeNDAP | ✓ |
| OPeNDAP | ✗ | ✗ | ✓, can share netCDF | ✓ |
| zarr | ✓ | moderate | ✓, with high performance | ✗, netCDF-c is working on it |

Thus, if we want to have datasets which can be used locally as well as remotely (some might call it “in the cloud”), we should take care that our datasets are convertible between those formats so that we can adapt to the specific use case.

### 6.4.3 An experiment

Currently, the main use case for the Aeris data server is to publish netCDF files and retrieve them back, either via direct download or via OPeNDAP. This experiment is designed to test under which circumstances data which has been created as netCDF can be retrieved correctly via OPeNDAP. In order to minimize additional problems due to the mix of incompatible libraries, all dataset decoding is done using the same netCDF-c library (independent of the access to the dataset, directly or via OPeNDAP).

The individual test cases show differences between data types, the use of negative numbers, the use of values used as numeric values, and the use of values interpreted as flags. For each data type, there is a drop down menu which shows the individual sub cases. If any one of the sub cases failed, the data type is marked as erroneous.

In each case the test follows the same steps:

- The original dataset is dumped via `ncdump`
- The OPeNDAP dataset attribute structure (`.das`) is retrieved via `curl` to look at the raw response of the server
- The dataset as received via OPeNDAP is dumped via `ncdump`
- A little script checks if it detects any significant error between the original and the OPeNDAP dataset

#### Results

---

**BYTE**

#### + numbers

original dataset:

```
$ ncdump test_NC_BYTE.nc
netcdf test_NC_BYTE {
dimensions:
        test = 5 ;
variables:
        byte test(test) ;
                test:_FillValue = 3b ;
                test:valid_range = 0b, 127b ;
data:

 test = 0, 1, 2, _, 127 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE.nc.das
Attributes {
    test {
        String _Unsigned "false";
        Byte _FillValue 3;
        Byte valid_range 0, 127;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE.nc
netcdf test_NC_BYTE {
dimensions:
        test = 5 ;
variables:
        byte test(test) ;
                test:_Unsigned = "false" ;
                test:_FillValue = 3b ;
                test:valid_range = 0b, 127b ;
data:

 test = 0, 1, 2, _, 127 ;
}
```

✅

### +/- numbers

original dataset:

```
$ ncdump test_NC_BYTE_neg.nc
netcdf test_NC_BYTE_neg {
dimensions:
        test = 7 ;
variables:
        byte test(test) ;
                test:_FillValue = 3b ;
                test:valid_range = -128b, 127b ;
data:

 test = -128, -1, 0, 1, 2, _, 127 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE_neg.nc.das
Attributes {
    test {
        String _Unsigned "false";
        Byte _FillValue 3;
        Int16 valid_range -128, 127;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE_neg.nc
netcdf test_NC_BYTE_neg {
dimensions:
        test = 7 ;
variables:
        byte test(test) ;
                test:_Unsigned = "false" ;
```

(continues on next page)

(continued from previous page)

```
                test:_FillValue = 3b ;
                test:valid_range = -128s, 127s ;
data:

 test = -128, -1, 0, 1, 2, _, 127 ;
}
```

**Error:** data type of attribute `valid_range` of variable `test` differs

### + flags

original dataset:

```
$ ncdump test_NC_BYTE_flag.nc
netcdf test_NC_BYTE_flag {
dimensions:
        test = 5 ;
variables:
        byte test(test) ;
                test:flag_values = 0b, 1b, 2b, 3b, 127b ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 127 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE_flag.nc.das
Attributes {
    test {
        String _Unsigned "false";
        Byte flag_values 0, 1, 2, 3, 127;
        String flag_meanings "a b c d e";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE_flag.nc
netcdf test_NC_BYTE_flag {
dimensions:
        test = 5 ;
variables:
        byte test(test) ;
                test:_Unsigned = "false" ;
                test:flag_values = 0b, 1b, 2b, 3b, 127b ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 127 ;
}
```

✅

### +/- flags

original dataset:

```
$ ncdump test_NC_BYTE_flag_neg.nc
netcdf test_NC_BYTE_flag_neg {
dimensions:
        test = 7 ;
variables:
        byte test(test) ;
                test:flag_values = -128b, -1b, 0b, 1b, 2b, 3b, 127b ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -128, -1, 0, 1, 2, 3, 127 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE_flag_neg.nc.das
Attributes {
    test {
        String _Unsigned "false";
        Int16 flag_values -128, -1, 0, 1, 2, 3, 127;
        String flag_meanings "a b c d e f g";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_BYTE_flag_neg.nc
netcdf test_NC_BYTE_flag_neg {
dimensions:
        test = 7 ;
variables:
        byte test(test) ;
                test:_Unsigned = "false" ;
                test:flag_values = -128s, -1s, 0s, 1s, 2s, 3s, 127s ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -128, -1, 0, 1, 2, 3, 127 ;
}
```

**Error:** data type of attribute `flag_values` of variable `test` differs

---

**UBYTE**

### + numbers

original dataset:

```
$ ncdump test_NC_UBYTE.nc
netcdf test_NC_UBYTE {
dimensions:
        test = 5 ;
variables:
        ubyte test(test) ;
                test:_FillValue = 3UB ;
                test:valid_range = 0UB, 255UB ;
data:

 test = 0, 1, 2, _, 255 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UBYTE.nc.das
Attributes {
    test {
        String _Unsigned "true";
        Byte _FillValue 3;
        Int16 valid_range 0, -1;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UBYTE.nc
netcdf test_NC_UBYTE {
dimensions:
        test = 5 ;
variables:
        byte test(test) ;
                test:_Unsigned = "true" ;
                test:_FillValue = 3b ;
                test:valid_range = 0s, -1s ;
data:

 test = 0, 1, 2, _, -1 ;
}
```

**Error:** values of variable `test` are not equal

### + flags

original dataset:

```
$ ncdump test_NC_UBYTE_flag.nc
netcdf test_NC_UBYTE_flag {
dimensions:
        test = 5 ;
variables:
        ubyte test(test) ;
                test:flag_values = 0UB, 1UB, 2UB, 3UB, 255UB ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 255 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UBYTE_flag.nc.das
Attributes {
    test {
        String _Unsigned "true";
        Int16 flag_values 0, 1, 2, 3, -1;
        String flag_meanings "a b c d e";
        Int16 _FillValue -1;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UBYTE_flag.nc
```

**Error:** /usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UBYTE_flag.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UBYTE_flag.nc: NetCDF: Not a valid data type or _FillValue type mismatch

**Error:** [Errno -45] NetCDF: Not a valid data type or _FillValue type mismatch: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UBYTE_flag.nc'

---

**SHORT**

### + numbers

original dataset:

```
$ ncdump test_NC_SHORT.nc
netcdf test_NC_SHORT {
dimensions:
        test = 5 ;
variables:
        short test(test) ;
                test:_FillValue = 3s ;
                test:valid_range = 0s, 32767s ;
data:

 test = 0, 1, 2, _, 32767 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT.nc.das
Attributes {
    test {
        Int16 _FillValue 3;
        Int16 valid_range 0, 32767;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT.nc
netcdf test_NC_SHORT {
dimensions:
        test = 5 ;
variables:
        short test(test) ;
                test:_FillValue = 3s ;
                test:valid_range = 0s, 32767s ;
data:

 test = 0, 1, 2, _, 32767 ;
}
```

✓

### +/- numbers

original dataset:

```
$ ncdump test_NC_SHORT_neg.nc
netcdf test_NC_SHORT_neg {
dimensions:
        test = 7 ;
variables:
        short test(test) ;
```

(continues on next page)

(continued from previous page)

```
                test:_FillValue = 3s ;
                test:valid_range = -32768s, 32767s ;
data:

 test = -32768, -1, 0, 1, 2, _, 32767 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT_neg.nc.das
Attributes {
    test {
        Int16 _FillValue 3;
        Int16 valid_range -32768, 32767;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT_neg.nc
netcdf test_NC_SHORT_neg {
dimensions:
        test = 7 ;
variables:
        short test(test) ;
                test:_FillValue = 3s ;
                test:valid_range = -32768s, 32767s ;
data:

 test = -32768, -1, 0, 1, 2, _, 32767 ;
}
```

✅

### + flags

original dataset:

```
$ ncdump test_NC_SHORT_flag.nc
netcdf test_NC_SHORT_flag {
dimensions:
        test = 5 ;
variables:
        short test(test) ;
                test:flag_values = 0s, 1s, 2s, 3s, 32767s ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 32767 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT_flag.nc.das
Attributes {
    test {
        Int16 flag_values 0, 1, 2, 3, 32767;
        String flag_meanings "a b c d e";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT_flag.nc
netcdf test_NC_SHORT_flag {
dimensions:
        test = 5 ;
variables:
        short test(test) ;
                test:flag_values = 0s, 1s, 2s, 3s, 32767s ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 32767 ;
}
```

✓

### +/- flags

original dataset:

```
$ ncdump test_NC_SHORT_flag_neg.nc
netcdf test_NC_SHORT_flag_neg {
dimensions:
        test = 7 ;
variables:
        short test(test) ;
                test:flag_values = -32768s, -1s, 0s, 1s, 2s, 3s, 32767s ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -32768, -1, 0, 1, 2, 3, 32767 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT_flag_neg.nc.das
Attributes {
    test {
        Int16 flag_values -32768, -1, 0, 1, 2, 3, 32767;
        String flag_meanings "a b c d e f g";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_SHORT_flag_neg.nc
netcdf test_NC_SHORT_flag_neg {
dimensions:
        test = 7 ;
variables:
        short test(test) ;
                test:flag_values = -32768s, -1s, 0s, 1s, 2s, 3s, 32767s ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -32768, -1, 0, 1, 2, 3, 32767 ;
}
```

✅

---

**USHORT**

#### + numbers

original dataset:

```
$ ncdump test_NC_USHORT.nc
netcdf test_NC_USHORT {
dimensions:
        test = 5 ;
variables:
        ushort test(test) ;
                test:_FillValue = 3US ;
                test:valid_range = 0US, 65535US ;
data:

 test = 0, 1, 2, _, 65535 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_USHORT.nc.das
Attributes {
    test {
        Int16 _FillValue 3;
        Int16 valid_range 0, -1;
        String _Unsigned "true";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_USHORT.nc
```

**Error:** /usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_USHORT.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_USHORT.nc: NetCDF: Not a valid data type or _FillValue type mismatch

**Error:** [Errno -45] NetCDF: Not a valid data type or _FillValue type mismatch: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_USHORT.nc'

### + flags

original dataset:

```
$ ncdump test_NC_USHORT_flag.nc
netcdf test_NC_USHORT_flag {
dimensions:
        test = 5 ;
variables:
        ushort test(test) ;
                test:flag_values = 0US, 1US, 2US, 3US, 65535US ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, _ ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_USHORT_flag.nc.das
Attributes {
    test {
        Int16 flag_values 0, 1, 2, 3, -1;
        String flag_meanings "a b c d e";
        Int16 _FillValue -1;
        String _Unsigned "true";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_USHORT_flag.nc
```

**Error:** /usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_USHORT_flag.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_USHORT_flag.nc: NetCDF: Not a valid data type or _FillValue type mismatch

**Error:** [Errno -45] NetCDF: Not a valid data type or _FillValue type mismatch: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_USHORT_flag.nc'

**INT**

### + numbers

original dataset:

```
$ ncdump test_NC_INT.nc
netcdf test_NC_INT {
dimensions:
        test = 5 ;
variables:
        int test(test) ;
                test:_FillValue = 3 ;
                test:valid_range = 0, 2147483647 ;
data:

 test = 0, 1, 2, _, 2147483647 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT.nc.das
Attributes {
    test {
        Int32 _FillValue 3;
        Int32 valid_range 0, 2147483647;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT.nc
netcdf test_NC_INT {
dimensions:
        test = 5 ;
variables:
        int test(test) ;
                test:_FillValue = 3 ;
                test:valid_range = 0, 2147483647 ;
data:

 test = 0, 1, 2, _, 2147483647 ;
}
```

✓

### +/- numbers

original dataset:

```
$ ncdump test_NC_INT_neg.nc
netcdf test_NC_INT_neg {
dimensions:
        test = 7 ;
variables:
        int test(test) ;
                test:_FillValue = 3 ;
                test:valid_range = -2147483648, 2147483647 ;
data:

 test = -2147483648, -1, 0, 1, 2, _, 2147483647 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT_neg.nc.das
Attributes {
    test {
        Int32 _FillValue 3;
        Int32 valid_range -2147483648, 2147483647;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT_neg.nc
netcdf test_NC_INT_neg {
dimensions:
        test = 7 ;
variables:
        int test(test) ;
                test:_FillValue = 3 ;
                test:valid_range = -2147483648, 2147483647 ;
data:

 test = -2147483648, -1, 0, 1, 2, _, 2147483647 ;
}
```

✅

### + flags

original dataset:

```
$ ncdump test_NC_INT_flag.nc
netcdf test_NC_INT_flag {
dimensions:
        test = 5 ;
variables:
        int test(test) ;
```

(continues on next page)

(continued from previous page)

```
                test:flag_values = 0, 1, 2, 3, 2147483647 ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 2147483647 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT_flag.nc.das
Attributes {
    test {
        Int32 flag_values 0, 1, 2, 3, 2147483647;
        String flag_meanings "a b c d e";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT_flag.nc
netcdf test_NC_INT_flag {
dimensions:
        test = 5 ;
variables:
        int test(test) ;
                test:flag_values = 0, 1, 2, 3, 2147483647 ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 2147483647 ;
}
```

✅

### +/- flags

original dataset:

```
$ ncdump test_NC_INT_flag_neg.nc
netcdf test_NC_INT_flag_neg {
dimensions:
        test = 7 ;
variables:
        int test(test) ;
                test:flag_values = -2147483648, -1, 0, 1, 2, 3, 2147483647 ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -2147483648, -1, 0, 1, 2, 3, 2147483647 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT_flag_neg.nc.das
Attributes {
    test {
        Int32 flag_values -2147483648, -1, 0, 1, 2, 3, 2147483647;
        String flag_meanings "a b c d e f g";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT_flag_neg.nc
netcdf test_NC_INT_flag_neg {
dimensions:
        test = 7 ;
variables:
        int test(test) ;
                test:flag_values = -2147483648, -1, 0, 1, 2, 3, 2147483647 ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -2147483648, -1, 0, 1, 2, 3, 2147483647 ;
}
```

✅

---

**UINT**

#### + numbers

original dataset:

```
$ ncdump test_NC_UINT.nc
netcdf test_NC_UINT {
dimensions:
        test = 5 ;
variables:
        uint test(test) ;
                test:_FillValue = 3U ;
                test:valid_range = 0U, 4294967295U ;
data:

 test = 0, 1, 2, _, 4294967295 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT.nc.das
Attributes {
    test {
        Int32 _FillValue 3;
        Int32 valid_range 0, -1;
```

(continues on next page)

(continued from previous page)

```
        String _Unsigned "true";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT.nc
```

**Error:** /usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT.nc: NetCDF: Not a valid data type or _FillValue type mismatch

**Error:** [Errno -45] NetCDF: Not a valid data type or _FillValue type mismatch: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT.nc'

### + flags

original dataset:

```
$ ncdump test_NC_UINT_flag.nc
netcdf test_NC_UINT_flag {
dimensions:
        test = 5 ;
variables:
        uint test(test) ;
                test:flag_values = 0U, 1U, 2U, 3U, 4294967295U ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, _ ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT_flag.nc.das
Attributes {
    test {
        Int32 flag_values 0, 1, 2, 3, -1;
        String flag_meanings "a b c d e";
        Int32 _FillValue -1;
        String _Unsigned "true";
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT_flag.nc
```

**Error:** /usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT_flag.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT_flag.nc: NetCDF: Not a valid data type or _FillValue type mismatch

**Error:** [Errno -45] NetCDF: Not a valid data type or _FillValue type mismatch: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT_flag.nc'

---

**INT64**

### + numbers

original dataset:

```
$ ncdump test_NC_INT64.nc
netcdf test_NC_INT64 {
dimensions:
        test = 5 ;
variables:
        int64 test(test) ;
                test:_FillValue = 3LL ;
                test:valid_range = 0LL, 9223372036854775807LL ;
data:

 test = 0, 1, 2, _, 9223372036854775807 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64.nc.das
Error {
    code = 403;
    message = "NcDDS Variable data type = long";
};
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64.nc
```

**Error:** oc_open: server error retrieving url: code=403 message="NcDDS Variable data type = long"/usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64.nc: NetCDF: Access failure

**Error:** [Errno -77] NetCDF: Access failure: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64.nc'

### +/- numbers

original dataset:

```
$ ncdump test_NC_INT64_neg.nc
netcdf test_NC_INT64_neg {
dimensions:
        test = 7 ;
variables:
        int64 test(test) ;
                test:_FillValue = 3LL ;
                test:valid_range = -9223372036854775808LL, 9223372036854775807LL ;
data:

 test = -9223372036854775808, -1, 0, 1, 2, _, 9223372036854775807 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64_neg.nc.das
Error {
    code = 403;
    message = "NcDDS Variable data type = long";
};
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64_neg.nc
```

**Error:** oc_open: server error retrieving url: code=403 message="NcDDS Variable data type = long"/usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_neg.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_neg.nc: NetCDF: Access failure

**Error:** [Errno -77] NetCDF: Access failure: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_neg.nc'

### + flags

original dataset:

```
$ ncdump test_NC_INT64_flag.nc
netcdf test_NC_INT64_flag {
dimensions:
        test = 5 ;
variables:
        int64 test(test) ;
                test:flag_values = 0LL, 1LL, 2LL, 3LL, 9223372036854775807LL ;
                test:flag_meanings = "a b c d e" ;
data:
```

(continues on next page)

(continued from previous page)

```
 test = 0, 1, 2, 3, 9223372036854775807 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64_flag.nc.das
Error {
    code = 403;
    message = "NcDDS Variable data type = long";
};
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64_flag.nc
```

**Error:** oc_open: server error retrieving url: code=403 message="NcDDS Variable data type = long"/usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_flag.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_flag.nc: NetCDF: Access failure

**Error:** [Errno -77] NetCDF: Access failure: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_flag.nc'

### +/- flags

original dataset:

```
$ ncdump test_NC_INT64_flag_neg.nc
netcdf test_NC_INT64_flag_neg {
dimensions:
        test = 7 ;
variables:
        int64 test(test) ;
                test:flag_values = -9223372036854775808LL, -1LL, 0LL, 1LL, 2LL, 3LL,␣
 ↪9223372036854775807LL ;
                test:flag_meanings = "a b c d e f g" ;
data:

 test = -9223372036854775808, -1, 0, 1, 2, 3, 9223372036854775807 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64_flag_neg.nc.das
Error {
    code = 403;
```

(continues on next page)

(continued from previous page)

```
    message = "NcDDS Variable data type = long";
};
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_INT64_flag_neg.nc
```

**Error:** oc_open: server error retrieving url: code=403 message="NcDDS Variable data type = long"/usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_flag_neg.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_flag_neg.nc: NetCDF: Access failure

**Error:** [Errno -77] NetCDF: Access failure: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_INT64_flag_neg.nc'

---

**UINT64**

### + numbers

original dataset:

```
$ ncdump test_NC_UINT64.nc
netcdf test_NC_UINT64 {
dimensions:
        test = 5 ;
variables:
        uint64 test(test) ;
                test:_FillValue = 3ULL ;
                test:valid_range = 0ULL, 0ULL ;
data:

 test = 0, 1, 2, _, 18446744073709551615 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT64.nc.das
Error {
    code = 403;
    message = "NcDDS Variable data type = long";
};
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT64.nc
```

**Error:** oc_open: server error retrieving url: code=403 message="NcDDS Variable data type = long"/usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT64.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT64.nc: NetCDF: Access failure

**Error:** [Errno -77] NetCDF: Access failure: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT64.nc'

### + flags

original dataset:

```
$ ncdump test_NC_UINT64_flag.nc
netcdf test_NC_UINT64_flag {
dimensions:
        test = 5 ;
variables:
        uint64 test(test) ;
                test:flag_values = 0ULL, 1ULL, 2ULL, 3ULL, 0ULL ;
                test:flag_meanings = "a b c d e" ;
data:

 test = 0, 1, 2, 3, 18446744073709551615 ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT64_flag.nc.das
Error {
    code = 403;
    message = "NcDDS Variable data type = long";
};
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_UINT64_flag.nc
```

**Error:** oc_open: server error retrieving url: code=403 message="NcDDS Variable data type = long"/usr/local/bin/ncdump: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT64_flag.nc: https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT64_flag.nc: NetCDF: Access failure

**Error:** [Errno -77] NetCDF: Access failure: b'https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_testfiles/test_NC_UINT64_flag.nc'

**FLOAT**

### + numbers

original dataset:

```
$ ncdump test_NC_FLOAT.nc
netcdf test_NC_FLOAT {
dimensions:
        test = 7 ;
variables:
        float test(test) ;
                test:_FillValue = NaNf ;
                test:valid_range = 0.f, Infinityf ;
data:

 test = 0, 1, 2, 3, _, 3.402823e+38, Infinityf ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_FLOAT.nc.das
Attributes {
    test {
        Float32 _FillValue NaN;
        Float32 valid_range 0.0, Infinity;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_FLOAT.nc
netcdf test_NC_FLOAT {
dimensions:
        test = 7 ;
variables:
        float test(test) ;
                test:_FillValue = NaNf ;
                test:valid_range = 0.f, Infinityf ;
data:

 test = 0, 1, 2, 3, _, 3.402823e+38, Infinityf ;
}
```

✅

### +/- numbers

original dataset:

```
$ ncdump test_NC_FLOAT_neg.nc
netcdf test_NC_FLOAT_neg {
dimensions:
        test = 10 ;
variables:
        float test(test) ;
                test:_FillValue = NaNf ;
                test:valid_range = -Infinityf, Infinityf ;
data:

 test = -Infinityf, -3.402823e+38, -1, 0, 1, 2, 3, _, 3.402823e+38, Infinityf ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_FLOAT_neg.nc.das
Attributes {
    test {
        Float32 _FillValue NaN;
        Float32 valid_range -Infinity, Infinity;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_FLOAT_neg.nc
netcdf test_NC_FLOAT_neg {
dimensions:
        test = 10 ;
variables:
        float test(test) ;
                test:_FillValue = NaNf ;
                test:valid_range = -Infinityf, Infinityf ;
data:

 test = -Infinityf, -3.402823e+38, -1, 0, 1, 2, 3, _, 3.402823e+38, Infinityf ;
}
```

✅

---

**DOUBLE**

### + numbers

original dataset:

```
$ ncdump test_NC_DOUBLE.nc
netcdf test_NC_DOUBLE {
dimensions:
        test = 7 ;
variables:
        double test(test) ;
                test:_FillValue = NaN ;
                test:valid_range = 0., Infinity ;
data:

 test = 0, 1, 2, 3, _, 1.79769313486232e+308, Infinity ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_DOUBLE.nc.das
Attributes {
    test {
        Float64 _FillValue NaN;
        Float64 valid_range 0.0, Infinity;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_DOUBLE.nc
netcdf test_NC_DOUBLE {
dimensions:
        test = 7 ;
variables:
        double test(test) ;
                test:_FillValue = NaN ;
                test:valid_range = 0., Infinity ;
data:

 test = 0, 1, 2, 3, _, 1.79769313486232e+308, Infinity ;
}
```

✓

### +/- numbers

original dataset:

```
$ ncdump test_NC_DOUBLE_neg.nc
netcdf test_NC_DOUBLE_neg {
dimensions:
        test = 10 ;
variables:
        double test(test) ;
```

(continues on next page)

(continued from previous page)

```
                test:_FillValue = NaN ;
                test:valid_range = -Infinity, Infinity ;
data:

 test = -Infinity, -1.79769313486232e+308, -1, 0, 1, 2, 3, _,
    1.79769313486232e+308, Infinity ;
}
```

data attribute structure via OPeNDAP:

```
$ curl https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_DOUBLE_neg.nc.das
Attributes {
    test {
        Float64 _FillValue NaN;
        Float64 valid_range -Infinity, Infinity;
    }
}
```

dataset as interpreted by netCDF via OPeNDAP:

```
$ ncdump https://observations.ipsl.fr/thredds/dodsC/EUREC4A/PRODUCTS/testdata/netcdf_
 ↪testfiles/test_NC_DOUBLE_neg.nc
netcdf test_NC_DOUBLE_neg {
dimensions:
        test = 10 ;
variables:
        double test(test) ;
                test:_FillValue = NaN ;
                test:valid_range = -Infinity, Infinity ;
data:

 test = -Infinity, -1.79769313486232e+308, -1, 0, 1, 2, 3, _,
    1.79769313486232e+308, Infinity ;
}
```

✅

### Summary

The result of this experiment is underwhelming but shows a clear picture. A large set of data types can not be used reliably in a combination of netCDF and OPeNDAP. The failure modes however differ significantly between the various types:

- 64 bit integers simply can not be encoded in XDR which is the binary encoding of OPeNDAP. Thus the server just fails and the user receives an error.
- unsigned short and unsigned int could be represented by OPeNDAP in principle, however the Server introduces spurious `_FillValue`s which additionally are of the wrong (signed) integral type such that the netCDF client library refuses to read the data. This is most likely a Bug in the specific server.
- unsigned bytes could in principle be represented by OPeNDAP, but the values received by the client are wrong. This might be an attempt by the client to somehow handle the erroneously delivered `valid_range`. The big problem here is that this can result in an **error without a message**.
- signed bytes can work sometimes even if they are **not representable** by OPeNDAP. This is due to a **hack** which has been introduced into netCDF-c which is based on the use of the additional `_Unsigned` attribute which is

created automatically by the server. The hack however only applies to the data values and not to the attributes. As a consequence, the data type of the attributes may be changed **depending on the values** stored in the attributes. In particular, **signed** bytes seem to work **only if they are positive**. The behaviour is however really weird, so maybe one should not count on it.

As a **consequence**, the only numeric data types which should be used in any dataset are `SHORT`, `INT`, `FLOAT` and `DOUBLE`. `STRING` and `CHAR` (when used as text) seem to be ok, but they have not been investigated in this setting yet.

### 6.4.4 Did we learn something?

Well, it depends. In principle, these results are also present, but well hidden inside the NetCDF Users Guide: The Best Practices state:

- Do not use char type variables for numeric data, use byte type variables instead.

[…]

NetCDF-3 does not have unsigned integer primitive types.

- To be completely safe with unknown readers, widen the data type, or use floating point.

This excludes the use of `CHAR` for numeric applications and effectively also `BYTE`, `UBYTE`.

And in the data model section we can find:

For maximum interoparability with existing code, new data should be created with the The Classic Model.

This excludes the use of 64 bit integers and unsigned types.

Thus, the take-home message of this article is not new, everything could have been found in the NetCDF Users Guide. However at least I was not fully aware of all of these implications and did only find the references into the Users Guide while preparing this article. I think if we want to provide datasets which are usable by everyone, the insights of this article are important to share.

CHAPTER

SEVEN

# REFERENCES

CHAPTER

# EIGHT

# SCRIPT EXECUTION STATISTICS

| Document | Modified | Method | Run Time (s) | Status |
|---|---|---|---|---|
| bacardi | 2021-09-15 18:40 | force | 16.96 | ✅ |
| cloudmasks | 2021-10-08 14:04 | force | 182.59 | ✅ |
| dropsondes | 2021-09-15 18:43 | force | 8.88 | ✅ |
| flight-phase-operations | 2021-09-15 18:43 | force | 2.33 | ✅ |
| flight_tracks_leaflet | 2021-09-15 18:44 | force | 61.2 | ✅ |
| hamp_comparison | 2021-09-15 18:44 | force | 29.6 | ✅ |
| merian_cloudradar | 2021-09-15 18:45 | force | 7.01 | ✅ |
| meteor_cloudradar | 2021-09-15 18:45 | force | 25.09 | ✅ |
| p3_AXBTs | 2021-10-19 11:53 | force | 171.44 | ✅ |
| p3_flight_tracks | 2021-10-08 15:36 | force | 75.66 | ✅ |
| p3_humidity_comparison | 2021-10-08 15:36 | force | 16.51 | ✅ |
| p3_wband_radar | 2021-10-08 15:37 | force | 23.07 | ✅ |
| p3_wsra | 2021-10-08 15:37 | force | 8.04 | ✅ |
| show_intake_catalog | 2021-09-15 18:54 | force | 44.09 | ✅ |
| smart | 2021-09-15 18:54 | force | 12.69 | ✅ |
| specmacs | 2021-09-15 18:54 | force | 12.07 | ✅ |
| unified | 2021-09-15 18:55 | force | 26.1 | ✅ |
| velox | 2021-09-15 18:55 | force | 13.83 | ✅ |
| wales | 2021-09-15 18:55 | force | 7.05 | ✅ |

# BIBLIOGRAPHY


[BSA+17] Sandrine Bony, Bjorn Stevens, Felix Ament, Sebastien Bigorre, Patrick Chazette, Susanne Crewell, Julien Delanoë, Kerry Emanuel, David Farrell, Cyrille Flamant, Silke Gross, Lutz Hirsch, Johannes Karstensen, Bernhard Mayer, Louise Nuijens, James H Ruppert, Irina Sandu, Pier Siebesma, Sabrina Speich, Frédéric Szczap, Julien Totems, Raphaela Vogel, Manfred Wendisch, and Martin Wirth. EUREC4A: A Field Campaign to Elucidate the Couplings Between Clouds, Convection and Circulation. *Surv. Geophys.*, 38(6):1529–1568, 2017. doi:10.1007/s10712-017-9428-0.

[Kon21] Heike Konow. Eurec4a's HALO. *tbd*, 2021.

[MKR+20] Wiebke Mohr, Stefan Kinne, Callum Rollo, Abiel Kidane, Oliver Schlenczek, Marcel Schröder, Robert Grosz, Sanola Sandiford, Andreas Raeke, Przemyslaw Makuch, John Gollop, Almuth Neuberger, Geiske de Groot, Marcel Meyer, Sebastian Los, Michal Chilinski, Alma Anna Ubele, Darek Baranowski, Yanmichel Morfa-Avalos, Jan von Arx, Ludwig Worbes, Jakub Nowak, Kevin Helfer, Heike Kalesse, Antonio Ibáñez-Landeta, Elizabeth Siddle, Katharina Baier, Wojciech Szkolka, Imke Schirmacher, and Johannes Röttenbacher. Eurec4a campaign, cruise no. m161, 17 jan 2020 - 03 mar 2020, bridgetown \(barbados\) - ponta delgada \(portugal\), meteor-berichte. 2020. doi:10.2312/cr_m161.

[PFB+21] R. Pincus, C. W. Fairall, A. Bailey, H. Chen, P. Y. Chuang, G. de Boer, G. Feingold, D. Henze, Q. T. Kalen, J. Kazil, M. Leandro, A. Lundry, K. Moran, D. A. Naeher, D. Noone, A. J. Patel, S. Pezoa, I. PopStefanija, E. J. Thompson, J. Warnecke, and P. Zuidema. Observations from the noaa p-3 aircraft during atomic. *Earth Syst. Sci. Data Discuss.*, 2021:1–25, 2021. doi:10.5194/essd-2021-11.

[QTC+21] P. K. Quinn, E. J. Thompson, D. J. Coffman, S. Baidar, L. Bariteau, T. S. Bates, S. Bigorre, A. Brewer, G. de Boer, S. P. de Szoeke, K. Drushka, G. R. Foltz, J. Intrieri, S. Iyer, C. W. Fairall, C. J. Gaston, F. Jansen, J. E. Johnson, O. O. Krüger, R. D. Marchbanks, K. P. Moran, D. Noone, S. Pezoa, R. Pincus, A. J. Plueddemann, M. L. Pöhlker, U. Pöschl, E. Quinones Melendez, H. M. Royer, M. Szczodrak, J. Thomson, L. M. Upchurch, C. Zhang, D. Zhang, and P. Zuidema. Measurements from the rv \textit Ronald H. Brown and related platforms as part of the atlantic tradewind ocean-atmosphere mesoscale interaction campaign (atomic). *Earth Syst. Sci. Data*, 13(4):1759–1790, 2021. doi:10.5194/essd-13-1759-2021.

[SBF+21] B. Stevens, S. Bony, D. Farrell, F. Ament, A. Blyth, C. Fairall, J. Karstensen, P. K. Quinn, S. Speich, C. Acquistapace, F. Aemisegger, A. L. Albright, H. Bellenger, E. Bodenschatz, K.-A. Caesar, R. Chewitt-Lucas, G. de Boer, J. Delanoë, L. Denby, F. Ewald, B. Fildier, M. Forde, G. George, S. Gross, M. Hagen, A. Hausold, K. J. Heywood, L. Hirsch, M. Jacob, F. Jansen, S. Kinne, D. Klocke, T. Kölling, H. Konow, M. Lothon, W. Mohr, A. K. Naumann, L. Nuijens, L. Olivier, R. Pincus, M. Pöhlker, G. Reverdin, G. Roberts, S. Schnitt, H. Schulz, A. P. Siebesma, C. C. Stephan, P. Sullivan, L. Touzé-Peiffer, J. Vial, R. Vogel, P. Zuidema, N. Alexander, L. Alves, S. Arixi, H. Asmath, G. Bagheri, K. Baier, A. Bailey, D. Baranowski, A. Baron, S. Barrau, P. A. Barrett, F. Batier, A. Behrendt, A. Bendinger, F. Beucher, S. Bigorre, E. Blades, P. Blossey, O. Bock, S. Böing, P. Bosser, D. Bourras, P. Bouruet-Aubertot, K. Bower, P. Branellec, H. Branger, M. Brennek, A. Brewer, P.-E. Brilouet, B. Brügmann, S. A. Buehler, E. Burke, R. Burton, R. Calmer, J.-C. Canonici, X. Carton, G. Cato Jr., J. A. Charles, P. Chazette, Y. Chen, M. T. Chilinski, T. Choularton, P. Chuang, S. Clarke, H. Coe, C. Cornet, P. Coutris, F. Couvreux, S. Crewell, T. Cronin, Z. Cui, Y. Cuypers, A. Da-

ley, G. M. Damerell, T. Dauhut, H. Deneke, J.-P. Desbios, S. Dörner, S. Donner, V. Douet, K. Drushka, M. Dütsch, A. Ehrlich, K. Emanuel, A. Emmanouilidis, J.-C. Etienne, S. Etienne-Leblanc, G. Faure, G. Feingold, L. Ferrero, A. Fix, C. Flamant, P. J. Flatau, G. R. Foltz, L. Forster, I. Furtuna, A. Gadian, J. Galewsky, M. Gallagher, P. Gallimore, C. Gaston, C. Gentemann, N. Geyskens, A. Giez, J. Gollop, I. Gouirand, C. Gourbeyre, D. de Graaf, G. E. de Groot, R. Grosz, J. Güttler, M. Gutleben, K. Hall, G. Harris, K. C. Helfer, D. Henze, C. Herbert, B. Holanda, A. Ibanez-Landeta, J. Intrieri, S. Iyer, F. Julien, H. Kalesse, J. Kazil, A. Kellman, A. T. Kidane, U. Kirchner, M. Klingebiel, M. Körner, L. A. Kremper, J. Kretzschmar, O. Krüger, W. Kumala, A. Kurz, P. L'Hégaret, M. Labaste, T. Lachlan-Cope, A. Laing, P. Landschützer, T. Lang, D. Lange, I. Lange, C. Laplace, G. Lavik, R. Laxenaire, C. Le Bihan, M. Leandro, N. Lefevre, M. Lena, D. Lenschow, Q. Li, G. Lloyd, S. Los, N. Losi, O. Lovell, C. Luneau, P. Makuch, S. Malinowski, G. Manta, E. Marinou, N. Marsden, S. Masson, N. Maury, B. Mayer, M. Mayers-Als, C. Mazel, W. McGeary, J. C. McWilliams, M. Mech, M. Mehlmann, A. N. Meroni, T. Mieslinger, A. Minikin, P. Minnett, G. Möller, Y. Morfa Avalos, C. Muller, I. Musat, A. Napoli, A. Neuberger, C. Noisel, D. Noone, F. Nordsiek, J. L. Nowak, L. Oswald, D. J. Parker, C. Peck, R. Person, M. Philippi, A. Plueddemann, C. Pöhlker, V. Pörtge, U. Pöschl, L. Pologne, M. Posyniak, M. Prange, E. Quiñones Meléndez, J. Radtke, K. Ramage, J. Reimann, L. Renault, K. Reus, A. Reyes, J. Ribbe, M. Ringel, M. Ritschel, C. B. Rocha, N. Rochetin, J. Röttenbacher, C. Rollo, H. Royer, P. Sadoulet, L. Saffin, S. Sandiford, I. Sandu, M. Schäfer, V. Schemann, I. Schirmacher, O. Schlenczek, J. Schmidt, M. Schröder, A. Schwarzenboeck, A. Sealy, C. J. Senff, I. Serikov, S. Shohan, E. Siddle, A. Smirnov, F. Späth, B. Spooner, M. K. Stolla, W. Szkółka, S. P. de Szoeke, S. Tarot, E. Tetoni, E. Thompson, J. Thomson, L. Tomassini, J. Totems, A. A. Ubele, L. Villiger, J. von Arx, T. Wagner, A. Walther, B. Webber, M. Wendisch, S. Whitehall, A. Wiltshire, A. A. Wing, M. Wirth, J. Wiskandt, K. Wolf, L. Worbes, E. Wright, V. Wulfmeyer, S. Young, C. Zhang, D. Zhang, F. Ziemen, T. Zinner, and M. Zöger. EUREC4A. *Earth Syst. Sci. Data Discuss.*, 2021:1–78, January 2021. doi:10.5194/essd-2021-18.

[SAB+19] Bjorn Stevens, Felix Ament, Sandrine Bony, Susanne Crewell, Florian Ewald, Silke Gross, Akio Hansen, Lutz Hirsch, Marek Jacob, Tobias Kölling, Heike Konow, Bernhard Mayer, Manfred Wendisch, Martin Wirth, Kevin Wolf, Stephan Bakan, Matthias Bauer-Pfundstein, Matthias Brueck, Julien Delanoë, André Ehrlich, David Farrell, Marvin Forde, Felix Gödde, Hans Grob, Martin Hagen, Evelyn Jäkel, Friedhelm Jansen, Christian Klepp, Marcus Klingebiel, Mario Mech, Gerhard Peters, Markus Rapp, Allison A. Wing, and Tobias Zinner. A high-altitude long-range aircraft configured as a cloud observatory: the narval expeditions. *Bull Amer. Met. Soc.*, 100(6):1061–1077, 2019. doi:10.1175/BAMS-D-18-0198.1.

[WMSH01] Manfred Wendisch, Dörthe Müller, Dieter Schell, and Jost Heintzenberg. An airborne spectral albedometer with active horizontal stabilization. *J. Atmos. Ocean Technol.*, 18(11):1856–1866, 2001. doi:10.1175/1520-0426(2001)018<1856:AASAWA>2.0.CO;2.