CLIMaCCF Documentation

Release V1.0

Deutsches Zentrum für Luft und Raumfahrt (DLR)
Universidad Carlos III de Madrid (UC3M)
Hamburg University of Technology (TUHH)
Delft University of Technology (TUD)

August 10, 2022

CONTENTS

INTRODUCTION

Overview: The Python Library CLIMaCCF is a software package developed by UC3M and DLR. The main idea of CLIMaCCF is to provide an open-source, easy-to-use, and flexible software tool that efficiently calculates spatially and temporally resolved climate impact of aviation emissions by using algorithmic climate change functions (aCCFs). The individual aCCFs of water vapour, NO_x-induced ozone and methane, and contrail-cirrus and also merged aCCFs that combine the individual aCCFs can be calculated.

License: CLIMaCCF is released under GNU Lesser General Public License v3.0 (LGPLv3). Citing the Software Documentation Paper (Dietmüller et al. 2022 [1]) together with CLIMaCCF software DOI (https://doi.org/10.5281/zenodo.6977272) and version number will serve to document the scientific impact of the software. You should consider this an obligation if you have taken advantage of CLIMaCCF.

Citation info: Dietmüller, S. Matthes, S., Dahlmann, K., Yamashita, H., Simorgh, A., Soler, M. , Linke, F., Lührs, B., Meuser, M. M., Weder, C., Grewe, V., Yin, F., Castino, F. (2022): A python library for computing individual and merged non-CO_2 algorithmic climate change functions: CLIMaCCF V1.0, GMDD.

User support: Support of all general technical questions on CLIMaCCF, i.e., installation, application, and development, will be provided by Abolfazl Simorgh (abolfazl.simorgh@uc3m.es), Simone Dietmüller (Simone.Dietmueller@dlr.de), and Hiroshi Yamashita (Hiroshi.Yamashita@dlr.de).

Core developer team: Abolfazl Simorgh (UC3M), Manuel Soler (UC3M), Simone Dietmüller (DLR), Hiroshi Yamashita (DLR), Sigrun Matthes (DLR).

GETTING STARTED

This section briefly presents the necessary information required to get started with CLIMaCCF.

2.1 Installation

The installation is the first step to working with CLIMaCCF. In the following, the steps required to install the library are provided.

0. It is highly recommended to create a virtual environment (e.g., env_climaccf) with Python version 3.8 (or 3.9):

```
conda create -n env_climaccf python==3.8
conda activate env_climaccf
pip install setuptools~=49.6.0
pip install pint~=0.19.2
```

1. Clone or download the repository. The CLIMaCCF source code is available on a public GitHub repository: https://github.com/dlr-pa/climaccf.git. The easiest way to obtain it is to clone the repository using git: git clone https://github.com/dlr-pa/climaccf.git.

2. Locate yourself in the CLIMaCCF (library folder) path, and run the following line, using terminal (MacOS and Linux) or cmd (Windows), which will install all dependencies:

```
python setup.py install
```

3. The installation package contains a set of sample data and an example script for testing purposes. To run it at the library folder, enter the following command:

```
python setup.py pytest
```

4. The library runs successfully if env_processed.nc is generated at the library folder/test/sample_data/. One can visualize the file using a visualization tool.

2.2 Configuration

The scope of CLIMaCCF is to provide individual and merged aCCFs as spatially and temporally resolved information considering meteorology from the actual synoptical situation, the aircraft type, the selected physical climate metric, and the selected version of prototype algorithms in individual aCCFs [1]. Consequently, some user-preferred settings need to be defined. The easiest (and user-friendliest, less error-prone) way is to use a configuration file. In CLIMaCCF, the configuration settings are included in a YAML file named *config-user.yml*. YAML is a human-friendly markup language and is commonly used for configuration files. In the following, a sample YAML user configuration file (also located in the CLIMaCCF folder) is provided:

```
#############################################
# User's configuration file for the CLIMaCCF #
#############################################


#** Configuration of the calculation of algorithmic climate change functions␣
↪(aCCFs) **#

# If true, efficacies are considered in the aCCF calculation
efficacy: true
        # Options: true, false
efficacy-option: lee_2021
        # Options one:  'lee_2021' (efficacies according to Lee et al. (2021))
        # Options two:   user-defined efficacies:
        #    CH4: xx
        #    CO2: xx
        #    Cont.: xx
        #    H2O: xx
        #    O3: xx


# Specifies the version of the prototype aCCF
aCCF-V: V1.1
        # currently 2 options for aCCFs: 'V1.0': Yin et al. (2022), 'V1.1':␣
↪Matthes et al. (2022)

# User-defined scaling factors of the above selected aCCF version. Not␣
↪recommended to be changed from default value 1, unless modification of the␣
↪aCCFs is wanted (e.g. sensitivity studies)
aCCF-scalingF:
  CH4: 1
  CO2: 1
  Cont.: 1
  H2O: 1
  O3: 1


# Specifies the climate indicator. Currently, Average Temperature Response (ATR)␣
↪has been implemented
climate_indicator: ATR
        # Options: 'ATR'
```

(continues on next page)

```
# Specifies the emission scenario of the climate metric. Currently, pulse␣
↪emission and increasing future emission scenario (business as usual) included
emission_scenario: future_scenario
        # Options: 'pulse' and 'future_scenario'


# Specifies the time horizon (in years) over which the selected climate indicator␣
↪is calculated
TimeHorizon: 20
        # Options: 20, 50, 100


# Determination of persistent contrail formation areas (PCFA), needed to␣
↪calculate aCCF of (day/night) contrails.
PCFA: PCFA-ISSR
      # Options: 'PCFA-ISSR' (PCFA defined by ice-supersaturated regions with␣
↪threshold for relative humidity over ice and temperature), 'PCFA-SAC' (Contrail␣
↪formation with Schmidt-Appleman criterion SAC (Appleman, 1953) &
      # contrail persistence, if ambient air is ice supersaturated)


# Parameters for calculating ice-supersaturated regions (ISSR)
PCFA-ISSR:
  # Specifies the threshold of relative humidity over ice in order to identify␣
↪ice supersaturated regions. Note that for persistent contrails relative␣
↪humidity over ice has to be greater 100%. However to take into account␣
↪subgridscale variability in humidity field of input data, the threshold of␣
↪relative humidity (over ice) has to be adopted for the selected resolution of␣
↪data product (for more details see Dietmueller et al. 2022)
  rhi_threshold: 0.9
      # Options: user defined threshold value < 1. Threshold depends on the used␣
↪data set, e.g., in case of the reanalysis data product ERA5 with high␣
↪resolution (HRES) it is 0.9
  temp_threshold: 235


# Parameters for calculating Schmidt-Appleman criterion (SAC). These parameters␣
↪vary for different aircraft types.
PCFA-SAC:
  # water vapour emission's index in [kg(H2O)/kg(fuel)]
  EI_H2O: 1.25
  # Fuel specific energy in [J/kg]
  Q: 43000000.0
  # Engine's overall efficiency
  eta: 0.3



#** Technical specifications of aircraft/engine dependent parameters **#
# Specifies the values of NOx emission index (NOx_EI) and flown distance per kg␣
↪burnt fuel (F_km)
```

```
NOx_EI&F_km: TTV
      # Options: 'TTV' for typical transatlantic fleet mean values (NOx_EI, F_km)␣
→from literature (Penner et al. 1999, Graver and Rutherford 2018) and 'ac_
→dependent' for altitude and aircraft/engine dependent values (NOx_EI, F_km).␣
→Note that if Confg['NOx_EI&F_km'] = 'TTV', the following confg['ac_type'] is␣
→ignored.

# If Confg['NOx_EI&F_km'] = 'ac_dependent', aircraft class (i.e. regional, single-
→aisle, wide-body) needs to be selected. For these aircraft classes aggregated␣
→fleet-level values of NOx_EI and F_km are provided (for more details see␣
→Dietmueller et al. 2022).
ac_type: wide-body
      # Options: 'regional', 'single-aisle', 'wide-body'

#** Specifies the saved output file **#

# If true, the primary mode ozone (PMO) effect is included to the CH4 aCCF and␣
→the total NOx aCCF
PMO: true
      # Options: true, false

# If true, the total NOx aCCF is calculated (i.e. aCCF-NOx = aCCF-CH4 + aCCF-O3)
NOx_aCCF: false
      # Options: true, false

# If true, all individual aCCFs are converted to the same unit of K/kg(fuel) and␣
→saved in the output file.
unit_K/kg(fuel): false
      # Options: true, false

# If true, merged non-CO2 aCCF is calculated
merged: true
      # Options: true, false

# If true, climate hotspots (regions that are very sensitive to aviation␣
→emissions) are calculated (for more details see Dietmueller et al. 2022)
Chotspots: false
      # Options: true, false

# If constant, climate hotspots are calculated based on the user-specified␣
→threshold,
# if dynamic, the thresholds for identifying climate hotspots are determined␣
→dynamically by calculating the percentile value of the merged aCCF over a␣
→certain geographical region (for details, see Dietmueller et al. 2022).
Chotspots_calc_method: dynamic
      # Options: constant, dynamic

# Specifies the constant threshold for calculating climate hotspots (if Chotspots_
→calc_method: constant).
```

```
Chotspots_calc_method_cons: 1e-13

# Specifies the percentage (e.g. 95%) of the percentile value as well as the␣
↪geographical region for which the percentile of the merged aCCF is calculated.␣
↪Thus the percentile defines the dynamical threshold for climate hotspots (if␣
↪Chotspots_calc_method: dynamic). Note that percentiles are saved in the output␣
↪file
Chotspots_calc_method_dynm:
     hotspots_percentile: 95
          # Options: percentage < 100
     latitude: false
          # Options: (lat_min, lat_max), false
     longitude: false
          # Options: (lon_min, lon_max), false

# If true, it assigns binary values to climate hotspots (0: areas with climate␣
↪impacts below a specified threshold. 1: areas with climate impacts above a␣
↪specified threshold)
# If false, it assigns 0 for areas with climate impacts below the specified␣
↪threshold and provides values of merged aCCFs for areas with climate impacts␣
↪above the threshold.
hotspots_binary: true
     # Options: true, false

# If true, meteorological input variables, needed to calculate aCCFs, are saved␣
↪in the netCDF output file in same resolution as the aCCFs
MET_variables: false
     # Options: true, false

# If true, polygons containing climate hotspots will be saved in the GeoJson file
geojson: true
     # Options: true, false

# Specifies the color of polygons
color: copper
     # Options: colors of cmap, e.g., copper, jet, Reds

# Specifies the horizontal resolution
horizontal_resolution: 0.5
     # Options: lower resolutions in degrees

# Specifies geographical region
lat_bound: false
     # Options: (lat_min, lat_max), false
lon_bound: false
     # Options: (lon_min, lon_max), false

# Specifies the output format
save_format: netCDF
```

2.2. Configuration

```
    # Options: netCDF (netcdf, nc) and PICKLE (pickle, Pickle)

#** Specifies output for statistical analysis, if ensemble prediction system␣
↪(EPS) data products are used **#

# The following two options (confg['mean'], confg['std']) are ignored if the␣
↪input data are deterministic

# If true, mean values of aCCFs and meteorological variables are saved in the␣
↪output file
mean: false
      # Options: true, false

# If true, standard deviation of aCCFs and meteorological variables are saved in␣
↪the output file
std: false
      # Options: true, false
```

One can load the configurations in the main script using:

```
with open("config-user.yml", "r") as ymlfile: confg = yaml.load(ymlfile)
```

Now, the configuration settings are included in a dictionary called *confg*. One can directly define configuration settings in a dictionary. Notice that default values for the settings have been defined within the library database; thus, defining dictionary *confg* is optional and, if included, overwrites the default ones.

2.3 Input

To calculate aCCFs within CLIMaCCF, meteorological input parameters are required. These input parameters are listed in Table 1, together with their physical unit. The current implementation of the Library is compatible with the standard of the European Centre for Medium-Range Weather Forecasts (ECMWF) data (for both reanalysis and forecast data products) (https://www.ecmwf.int). In the case of taking ECMWF input data, the respective short names and parameter IDs are given in Table 1. The user has to provide two datasets: one for input data provided at each pressure level and one for input data provided on one single pressure level (e.g., surface layer or top of atmosphere (TOA)). Within CLIMaCCF, the directories of these two datasets are defined in climaccf_run_main.py:

```
input_dir = {}
# Input data provided at pressure levels such as temperature, geopotential and␣
↪relative humidity:
input_dir['path_pl']  = dir_pressure_variables

# Input data provided at one single pressure level such as top net thermal␣
↪radiation at the TOA:
input_dir['path_sur'] =  dir_surface_variables
```

Table 1: Meteorological input parameters needed to calculate aCCFs within CLIMaCCF. Respective ECMWF short names, units, and parameter IDs are provided.

Parameter	Short name	Units	ECMWF parameter ID
Pressure	pres	$[K.m^2/Kg.s]$	54
Potential vorticity	pv	$[K.m^2/Kg.s]$	60
Geopotential	z	$[m^2/s^2]$	129
Temperature	t	$[K]$	130
Relative humidity	r	[%]	157
Top net thermal radiation	ttr	$[J/m^2]$	179
TOA incident solar radiation	tisr	$[J/m^2]$	212

In addition to the locations of input data, the directory of the CLIMaCCF needs to be specified within input_dir:

```
# Directory of CLIMaCCF:
input_dir ['path_lib'] = climaccf_dir
```

Finally, the directory where all outputs will be written has to be provided by the user:

```
# Destination directory where all output will be written:
output_dir = dir_results
```

2.4 Running & output

After defining configurations and input and output directories, CLIMaCCF is prepared to calculate individual and merged aCCFs. To start working, we import the library:

```
import climaccf
from climaccf.main_processing import ClimateImpact
```

First, the input meteorological variables will be processed. This processing step is mainly related to 1) extracting variables of input data, 2) calculating required variables from alternative ones in case of missing variables (for details, see Table 5 of Dietmüller et al. 2022 [1]), 3) unifying the naming and dimension of variables, and 4) changing the resolution and geographical area of the output. The horizontal resolution and the geographical region of the output can be selected in the user configuration file (config-user.yml). Notice that the horizontal resolution cannot be higher than the resolution of the meteorological input data, and the decrease in resolution is a factor i of natural numbers. For instance, if the resolution of meteorological input data is $0.25° \times 0.25°$, the resolution can be reduced to $i \cdot 0.25° \times i \cdot 0.25°$, for $i \in \mathrm{N}$.

```
CI = ClimateImpact(input_dir, output_dir, **confg)
```

Second, after processing the weather data, aCCFs are calculated, taking into account the user-defined configuration settings in *config-user.yml*.

```
CI.calculate_accfs(**confg)
```

Third, an output file (either in netCDF or PICKLE file formats) will be generated. The output file contains different variables depending on the selected user configurations. For instance, the output file contains both individual and merged aCCFs if, in *config-user.yml*, one selects **merged: true**. The dimension of output variables for the Ensemble Prediction System (EPS) data products is *time, member, pressure level, latitude,* and *longitude* (i.e., 5D array), and for the deterministic ones, *time, pressure level, latitude,* and *longitude* (i.e., 4D array). The generated netCDF file (if selected) is compatible with well-known visualization tools such as ferret, NCO, and Panoply. In addition to the netCDF (or PICKLE), the user can choose the GeoJSON format for storing polygons of climate sensitive regions (i.e., climate hotspots). If one selects: **merged: true, Chotspots: true**, some GeoJson files (number: pressure levels * number of time) will be generated in the specified output directory.

MODULES

3.1 Processing of meteorological input data

climaccf.extract_data.**extract_coordinates**(*ds*, *ex_variables*, *ds_sur=None*)

> Extracts coordinates (axes) of the input dataset defined with different possible names.

> > **Parameters**
> > > **ds** (Dataset) – Dataset opened with xarray.

> > **Returns ex_var_name**
> > > List of available coordinates.

> > **Return type**
> > > list

> > **Returns variables**
> > > Assigns bool to the axes (e.g., if ensemble members are not available, it sets False).

> > **Return type**
> > > dict

climaccf.extract_data.**extract_data_variables**(*ds*, *ds_sr=None*, *verbose=False*)

> Extracts available required variables of the input dataset defined with different possible names.

> > **Parameters**
> > > - **ds** (Dataset) – Dataset opened with xarray.
> > > - **ds_sr** (Dataset) – Dataset containing surface parameters opened with xarray.
> > > - **verbose** (bool) – Used to show more information.

> > **Returns ex_var_name**
> > > Available required weather variables.

> > **Return type**
> > > list

> > **Returns variables**
> > > Assigns bool to the required weather variables.

> > **Return type**
> > > dict

climaccf.extract_data.**logic_cal_accfs**(*variables*)

> Creates a dictionary containing logical values showing the possibility to calculate each aCCF.
>
> > **Parameters**
> > > **variables** (dict) – Variables available in the given dataset.
> >
> > **Returns**
> > > dictionary containing logical values showing the possibility to calculate each aCCF.
> >
> > **Return type**
> > > dict

climaccf.extend_dim.**extend_dimensions**(*inf_coord, ds, ds_sur, ex_variables*)

> Unifies the dimension of all types of given data as either 4- or 5-dimensional arrays, depending on the existence of ensemble members. If the data has only two fields: latitude and longitude, this function adds time and level fields, (e.g., for the deterministic data products: (latitude:360, longitude:720) -> (time:1, pressure level:1, latitude:360, longitude:720)).
>
> > **Parameters**
> > > - **inf_coord** (dict) – Information on original coordinates.
> > > - **ds** (Dataset) – Dataset opened with xarray containing variables on pressure levels.
> > > - **ds_sur** (Dataset) – Dataset containing surface parameters opened with xarray.
> > > - **ex_variables** (dict) – New coordinates
> >
> > **Returns ds_pl**
> > > New dataset of pressure level variables including the added coordinates
> >
> > **Return type**
> > > dataset
> >
> > **Returns ds_surf**
> > > New dataset of surface parameters including the added coordinates
> >
> > **Return type**
> > > dataset

climaccf.processing_surf_vars.**extend_olr_pl_4d**(*sur_var, pl_var, index, fore_step*)

> Calculates outgoing longwave radiation (OLR) [W/m^2] at TOA from the parameter top net thermal radiation (ttr) [J/m^2], and extends (duplicating) it to all pressure levels for consistency of dimensions. For a specific time, OLR is calculated in 3D (i.e., level, latitude, longitude).
>
> > **Parameters**
> > > - **sur_var** (Dataset) – Dataset containing surface parameters opened with xarray.
> > > - **pl_var** (Dataset) – Dataset containing pressure level parameters opened with xarray.
> > > - **index** (int) – Index of the time.

- **fore_step** (int) – Forecast step in hours.

> **Returns arr**
> OLR in 3D (i.e., level, latitude, longitude).

> **Return type**
> array

climaccf.processing_surf_vars.**extend_olr_pl_5d**(*sur_var*, *pl_var*, *index*, *fore_step*)

Calculates outgoing longwave radiation (OLR) [W/m^2] at TOA from the parameter top net thermal radiation (ttr) [J/m^2], and extends (duplicating) it to all pressure levels for consistency of dimensions. For a specific time, OLR is calculated in 4D (i.e., number, level, latitude, longitude).

> **Parameters**
>
> - **sur_var** (Dataset) – Dataset containing surface parameters opened with xarray.
>
> - **pl_var** (Dataset) – Dataset containing pressure level parameters opened with xarray.
>
> - **index** (int) – Index of the time that exists in the dataset of pressure level parameters at this step.
>
> - **fore_step** (int) – Forecast step in hours.

> **Returns arr**
> OLR in 4D (i.e., number, level, latitude, longitude).

> **Return type**
> array

climaccf.processing_surf_vars.**get_olr**(*sur_var*, *pl_var*, *number=True*, *fore_step=None*)

Calculates outgoing longwave radiation (OLR) [W/m^2] at TOA from the parameter top net thermal radiation (ttr) [J/m^2]. OLR is calculated in 5D or 4D depending on the existence of ensemble members.

> **Parameters**
>
> - **sur_var** (Dataset) – Dataset containing surface parameters opened with xarray.
>
> - **pl_var** (int) – Dataset containing pressure level parameters opened with xarray.
>
> - **number** (bool) – Determines whether the weather data contains ensemble members or not.
>
> - **fore_step** – Forecast step in hours.

> **Returns arr**
> OLR.

> **Return type**
> numpy.ndarray

climaccf.processing_surf_vars.**get_olr_4d**(*sur_var*, *pl_var*, *thr*, *fore_step=None*)

Calculates outgoing longwave radiation (OLR) [W/m^2] at TOA from the parameter top

---

net thermal radiation (ttr) [J/m^2]. OLR is calculated in 4D (i.e, time, level, latitude, longitude).

> **Parameters**
> - **sur_var** (`Dataset`) – Dataset containing surface parameters opened with xarray.
> - **pl_var** (`int`) – Dataset containing pressure level parameters opened with xarray.
> - **thr** (`dict`) – Thresholds to automatically determine forecast steps.
> - **fore_step** – Forecast step in hours.
>
> **Returns arr**
> OLR in 4D (i.e., time, level, latitude, longitude).
>
> **Return type**
> numpy.ndarray

climaccf.processing_surf_vars.**get_olr_5d**(*sur_var*, *pl_var*, *thr*, *fore_step=None*)

> Calculates outgoing longwave radiation (OLR) [W/m^2] at TOA from the parameter top net thermal radiation (ttr) [J/m^2]. OLR is calculated in 5D (i.e, time, number, level, latitude, longitude).
>
> **Parameters**
> - **sur_var** (`Dataset`) – Dataset containing surface parameters opened with xarray.
> - **pl_var** (`int`) – Dataset containing pressure level parameters opened with xarray.
> - **thr** (`dict`) – Thresholds to automatically determine forecast steps.
> - **fore_step** – Forecast step in hours.
>
> **Returns arr**
> OLR in 5D (i.e., time, number, level, latitude, longitude).
>
> **Return type**
> numpy.ndarray

3.2 Calculation of meteorological input data from alternative variables

climaccf.calc_altrv_vars.**get_pvu**(*ds*)

> Calculates potential vorticity [in PVU] from meteorological variables pressure, temperature and x and y component of the wind using MetPy (https://www.unidata.ucar.edu/software/metpy/).
>
> **Parameters**
> **ds** (`Dataset`) – Dataset opened with xarray.
>
> **Returns PVU**
> potential vorticity [in PVU]

> **Return type**
> numpy.ndarray

climaccf.calc_altrv_vars.**get_rh_ice**(*ds*)

> Calculates relative humidity over ice from relative humidity over water
>
> > **Parameters**
> > **ds** (Dataset) – Dataset opened with xarray.
> >
> > **Returns rh_ice**
> > relative humidity over ice [in %]
> >
> > **Return type**
> > numpy.ndarray

climaccf.calc_altrv_vars.**get_rh_sd**(*ds*)

> Calculates the relative humidity over ice/water from specific humidity
>
> > **Parameters**
> > **ds** (Dataset) – Dataset opened with xarray.
> >
> > **Returns rh_sd**
> > relative humidity over water/ice [%]
> >
> > **Return type**
> > numpy.ndarray

3.3 Weather store

class climaccf.weather_store.**WeatherStore**(*weather_data, weather_data_sur=None,*
flipud='auto', **weather_config)

> Prepare the data required to calculate aCCFs and store them in a xarray dataset.
>
> **__init__**(*weather_data, weather_data_sur=None, flipud='auto', **weather_config*)
>
> > Processes the weather data.
> >
> > > **Parameters**
> > > - **weather_data** – Dataset opened with xarray containing variables on different pressure levels.
> > > - **weather_data_sur** – Dataset opened with xarray containing variables on single pressure level (i.e., outgoing longwave radiation in this case).
>
> **get_xarray**()
>
> > Creates a new xarray dataset containing processed weather variables.
> >
> > > **Returns ds**
> > > xarray dataset containing user-selected variables (e.g., merged aCCFs, mean aCCFs, Climate hotspots).
> > >
> > > **Return type**
> > > dataset

reduce_domain(*bounds, verbose=False*)

    Reduces horizontal domain and time.

        Parameters

            bounds – ranges defined as tuple (e.g., lat_bound=(35, 60.0)).

        Return type

            dict

3.4 Persistent contrail formation

climaccf.contrail.**get_cont_form_thr**(*ds, member, SAC_config*)

    Calculates the threshold temperature and threshold of relative humidity over water required for contrail formation (Schmidt-Applemann-Citerion, Applemann 1953). A good approximation of the Schmidt-Appleman Criterion is given in Schumann 1996.

        Parameters

            • **ds** (Dataset) – Dataset opened with xarray.

            • **member** (bool) – Detemines the presense of ensemble forecasts in the given dataset.

        Returns SAC_config

            Configurations containing required parameters to calculate Schmidt-Applemann-Citerion.

        Return type

            dict

        Returns T_Crit

            Threshold temperature for Schmidt-Appleman

        Return type

            numpy.ndarray

climaccf.contrail.**get_pcfa**(*ds, member, confg*)

    Calculates the persistent contrail formation areas (PCFA) with two options: 1) PCFA defined by ice-supersaturated regions with threshold for relative humidity over ice and temperature and 2) Contrail formation with Schmidt-Appleman criterion SAC (Appleman, 1953) & contrail persistence, if ambient air is ice supersaturated. Areas of persistent contrail formation are needed to calculate aCCF of (day/night) contrails.

        Parameters

            • **ds** (Dataset) – Dataset opened with xarray.

            • **member** (dict) – Detemines the presense of ensemble members in the given dataset.

            • **confg** – Configurations containing the selected option to calculate PCFA and required parameters for each option.

        Returns pcfa

            Persistent contrail formation areas (PCFA).

Return type
    numpy.ndarray

climaccf.contrail.**get_relative_hum**(*ds*, *member*, *intrp=True*)

Relative humiditiy over ice and water provided by ECMWF dataset. In ECMWF relative humidity is defined with respect to saturation of the mixed phase: i.e. with respect to saturation over ice below -23 °C and with respect to saturation over water above °C. In the regime in between a quadratic interpolation is applied.

> **Parameters**
>
> - **ds** (Dataset) – Dataset opened with xarray.
>
> - **member** (bool) – Detemines the presense of ensemble forecasts in the given dataset.
>
> **Returns ri**
>   Relative humidity over ice.
>
> **Return type**
>   numpy.ndarray
>
> **Returns rw**
>   Relative humidity over water.
>
> **Return type**
>   numpy.ndarray

climaccf.contrail.**get_rw_from_specific_hum**(*ds*, *member*)

Calculates relative humidity over water from specific humidity.

> **Parameters**
>
> - **ds** (Dataset) – Dataset opened with xarray.
>
> - **member** (bool) – Detemines the presense of ensemble forecasts in the given dataset.
>
> **Returns r_w**
>   Relative humidity over water.
>
> **Return type**
>   numpy.ndarray

3.5 Calculation of prototype aCCFs

class climaccf.accf.**GeTaCCFs**(*wd_inf*)

Calculation of algorithmic climate change functions (aCCFs).

__init__(*wd_inf*)

Prepares the data required to calculate aCCFs and store them in self.

> **Parameters**
>   **wd_inf** (Class) – Contains processed weather data with all information.

---

accf_ch4()

Calculates the aCCF of methane according to Yin et al. 2022 [2] (aCCF-V1.0) and Matthes et al. 2022 [3] (aCCF-V1.1): aCCF values are given in average temperature response as over next 20 years, assuming pulse emission (P-ATR20-methane [K/kg(NO_2)]). To calculate the aCCF of methane, meteorological variables geopotential and incoming solar radiation are required.

> **Returns accf**
> Algorithmic climate change function of methane.

> **Return type**
> numpy.ndarray

accf_dcontrail()

Calculates the aCCF of day-time contrails according to Yin et al. 2022 [2] (aCCF-V1.0) and Matthes et al. 2022 [3] (aCCF-V1.1): aCCF values are given in average temperature response as over next 20 years, assuming pulse emissions (P-ATR20-contrails [K/km]). To calculate the aCCF of day-time contrails, meteorological variables temperature and relative humidity over ice are required. Notice that, relative humidity over ice is required for the determination of persistent contrail formation areas.

> **Returns accf**
> Algorithmic climate change function of day-time contrails.

> **Return type**
> numpy.ndarray

accf_h2o()

Calculates the aCCF of water vapour according to Yin et al. 2022 [2] (aCCF-V1.0) and Matthes et al. 2022 [3] (aCCF-V1.1): aCCF values are given in average temperature response as over next 20 years, assuming pulse emission (P-ATR20-water-vapour [K/kg(fuel)]). To calculate the aCCF of water vapour, meteorological variable potential vorticity is required.

> **Returns accf**
> Algorithmic climate change function of water vapour.

> **Return type**
> numpy.ndarray

accf_ncontrail()

Calculates the aCCF of night-time contrails according to Yin et al. 2022 [2] (aCCF-V1.0) and Matthes et al. 2022 [3] (aCCF-V1.1): aCCF values are given in average temperature response as over next 20 years, assuming pulse emissions (P-ATR20-contrails [K/km]). To calculate the aCCF of night-time contrails, meteorological variables temperature and relative humidity over ice are required. Notice that, relative humidity over ice is required for the determination of persistent contrail formation areas.

> **Returns accf**
> Algorithmic climate change function of nighttime contrails.

> **Return type**
> numpy.ndarray

accf_o3()

Calculates the aCCF of ozone according to Yin et al. 2022 [2] (aCCF-V1.0) and Matthes et al. 2022 [3] (aCCF-V1.1): aCCF values are given in average temperature response as over next 20 years, assuming pulse emission (P-ATR20-ozone [K/kg(NO_2)]). To calculate the aCCF of ozone, meteorological variables temperature and geopotential are required.

> **Returns accf**
>> Algorithmic climate change function of Ozone.
>
> **Return type**
>> numpy.ndarray

get_accfs(***problem_config*)

Calculates individual aCCFs, the merged aCCF and climate hotspots based on the defined configurations, parameters and etc.

get_std(*var*, *normalize=False*)

Calculates the standard deviation of the inputted variables over the ensemble members.

> **Parameters**
>> - **var** – variable.
>>
>> - **normalize** – If True, it calculates standard deviation over the normalized variable. If False, standard deviation is taken from the original variable.
>
> **Return type**
>> numpy.ndarray
>
> **Return type**
>> bool
>
> **Returns x_std**
>> standard deviation of the variable.
>
> **Return type**
>> numpy.ndarray

get_xarray()

Creates an xarray dataset containing user-selected variables.

> **Returns ds**
>> xarray dataset containing user-selected variables (e.g., merged aCCFs, mean aCCFs, Climate hotspots).
>
> **Return type**
>> dataset
>
> **Returns enc**
>> encoding
>
> **Return type**
>> dict

`climaccf.accf.`**`convert_accf`**(*name, value, confg*)

Converts aCCFs based on the selected configurations (i.e., efficacy, climate indicator, emission scenarios and time horizons).

> **Parameters**
>
> - **name** – Name of the species (e.g., 'CH4').
>
> - **value** – Value of the species to be converted (P-ATR20 without efficacy factor).
>
> - **confg** – User-defined configurations for conversions.
>
> **Return type**
> string
>
> **Return type**
> numpy.ndarray
>
> **Return type**
> dict
>
> **Returns value**
> Converted aCCF.
>
> **Return type**
> numpy.ndarray

`climaccf.accf.`**`get_Fin`**(*ds, lat*)

Calculates TOA incoming solar radiation.

> **Parameters**
>
> - **ds** – dataset to extract the number of day.
>
> - **lat** – latitude.
>
> **Return type**
> Dataset
>
> **Return type**
> numpy.ndarray
>
> **Returns Fin**
> Incoming solar radiation.
>
> **Return type**
> numpy.ndarray

TESTING CLIMACCF

Here we provide an example configuration script together with some provided ERA5 sample data (retrieved from the Copernicus Climate Data Store: https://cds.climate.copernicus.eu/, European Reanalysis 5, 2020)). In order to test CLIMaCCF on your system and to test if the output is generated correctly, we recommend running CLIMaCCF using the example provided in the following.

First of all, define the configurations in a YAML file format (e.g., config-user.yml) as:

```yaml
# Configuration of the calculation of algorithmic climate change functions
↪(aCCFs) #

efficacy: true
efficacy-option: lee_2021
aCCF-V: V1.1
aCCF-scalingF:
    CH4: 1
    CO2: 1
    Cont.: 1
    H2O: 1
    O3: 1
climate_indicator: ATR
emission_scenario: future_scenario
TimeHorizon: 20
PCFA: PCFA-ISSR
PCFA-ISSR:
    rhi_threshold: 0.9
temp_threshold: 235
PCFA-SAC:
    EI_H2O: 1.25
    Q: 43000000.0
    eta: 0.3


# Technical specifiactions of aircraft/engine dependent parameters #

NOx_EI&F_km: TTV
ac_type: wide-body

# Specifies the saved output file #
```

```
PMO: true
NOx_aCCF: false
unit_K/kg(fuel): false
merged: true
Chotspots: false
Chotspots_calc_method: dynamic
Chotspots_calc_method_cons: 1e-13
Chotspots_calc_method_dynm:
    hotspots_percentile: 95
    latitude: false
    longitude: false
hotspots_binary: true
MET_variables: false
geojson: true
color: copper
horizontal_resolution: 0.5
lat_bound: false
lon_bound: false
save_format: netCDF


# Specifies output for statistical analysis, if ensemble prediction system (EPS)␣
↪data products are used #


mean: false
std: false
```

Then, by running the following script:

```python
import climaccf
from climaccf.main_processing import ClimateImpact

path_here = 'climaccf/'
test_path = path_here + '/test/sample_data/'
input_dir = {'path_pl': test_path + 'pressure_lev_june2018_res0.5.nc', 'path_sur
↪': test_path + 'surface_june2018_res0.5.nc', 'path_lib': path_here}
output_dir = test_path + 'env_processed.nc'

""" %%%%%%%%%%%%%%% LOAD CONFIGURATIONS %%%%%%%%%%%%%%% """

with open("config-user.yml", "r") as ymlfile:
    confg = yaml.safe_load(ymlfile)

""" %%%%%%%%%%%%%%%%% MAIN %%%%%%%%%%%%%%%%% """

CI = ClimateImpact(input_dir, output_dir, **confg)
CI.calculate_accfs(**confg)
```

the output netCDF file is generated in: *climaccf/test/sample_data/env_processed.nc*. In the following, a script is provided to visualize the output.

```python
from cartopy.mpl.geoaxes import GeoAxes
import cartopy.crs as ccrs
from cartopy.mpl.geoaxes import GeoAxes
from cartopy.mpl.ticker import LongitudeFormatter, LatitudeFormatter
import matplotlib.pyplot as plt
import matplotlib as mpl
from mpl_toolkits.axes_grid1 import AxesGrid
import numpy as np
import xarray as xr

plt.rc('font',**{'family':'serif','serif':['cmr10']})
plt.rc('text', usetex=True)
font = {'family' : 'normal',
        'size'   : 13}


path = 'climaccf/test/sample_data/env_processed.nc'
ds = xr.open_dataset(path, engine='h5netcdf')
lats = ds['latitude'].values
lons = ds['longitude'].values
lons1,lats1 = np.meshgrid(lons,lats)

cc_lon = np.flipud(lons1)[::1, ::1]
cc_lat = np.flipud(lats1)[::1, ::1]



time = np.datetime64('2018-06-01T06')
pressure_level = 250
time_idx = np.where (ds.time.values == time)[0][0]
pl_idx   = np.where (ds.level.values == pressure_level) [0][0]
aCCF_merged  = np.flipud(ds['aCCF_merged'].values[time_idx, pl_idx, :, :])[::1,␣
↪::1]

def main():
    projection = ccrs.PlateCarree()
    axes_class = (GeoAxes,
                  dict(map_projection=projection))


    fig = plt.figure(figsize=(5,5))
    axgr = AxesGrid(fig, 111, axes_class=axes_class,
                    nrows_ncols=(1,1),
                    axes_pad=1.0,
                    share_all = True,
                    cbar_location='right',
                    cbar_mode='each',
                    cbar_pad=0.2,
                    cbar_size='3%',
                    label_mode='')  # note the empty label_mode

    for i, ax in enumerate(axgr):
```

(continues on next page)

```
        xticks = [-20, -5, 10, 25, 40, 55]
        yticks = [0,10,20, 30, 40,  50,  60, 70, 80]
        ax.coastlines()
        ax.set_xticks(xticks, crs=projection)
        ax.set_yticks(yticks, crs=projection)
        lon_formatter = LongitudeFormatter(zero_direction_label=True)
        lat_formatter = LatitudeFormatter()
        ax.xaxis.set_major_formatter(lon_formatter)
        ax.yaxis.set_major_formatter(lat_formatter)
        ax.set_title(time)
        p = ax.contourf(cc_lon, cc_lat, aCCF_merged,
                        transform=projection,
                        cmap='YlOrRd')

        axgr.cbar_axes[i].colorbar(p)
        cax = axgr.cbar_axes[i]
        axis = cax.axis[cax.orientation]
        axis.label.set_text('aCCF-merged [K/kg(fuel)]')

    plt.show()

main()
```

For instance, using the script, one should get the following figure for the merged aCCFs at
250hPa on June 01, 2018 at 06:00 (UTC):
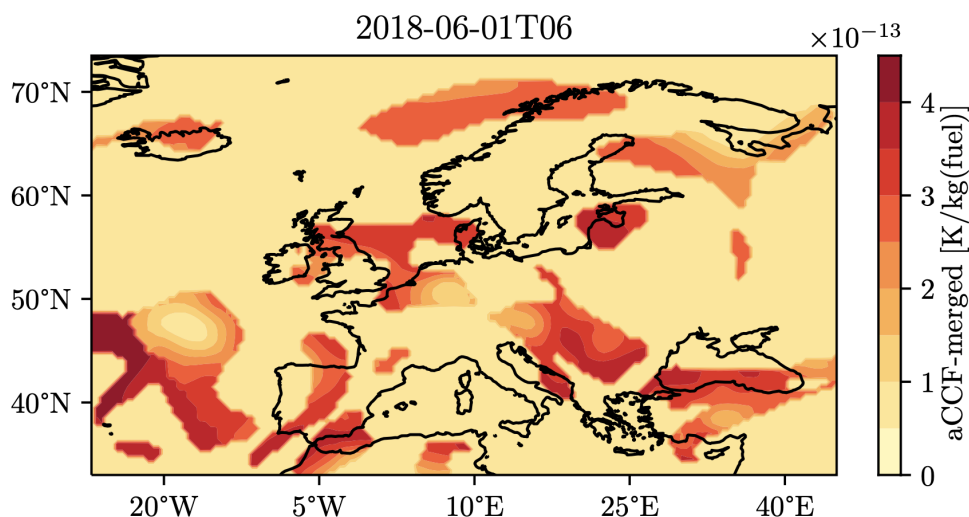
Figure S0: Example for merged non-CO_2 aCCF at pressure level 250 hPa over Europe

ACKNOWLEDGMENTS

[1] Simone Dietmüller, Sigrun Matthes, Katrin Dahlmann, Hiroshi Yamashita, Abolfazl Simorgh, Manuel Soler, Florian Linke, Benjamin Lührs, Maximiliam M. Meuser, Christian Weder, Volker Grewe, Feijia Yin and Federica Castino. A python library for computing individual and merged non-CO_2 algorithmic climate change functions. *GMDD*, 2022.

[2] Feijia Yin, Volker Grewe, Federica Castino, Pratik Rao, Sigrun Matthes, Hiroshi Yamashita, Katrin Dahlmann, Christine Frömming, Simone Dietmüller, Emma Peter, Patrick Klingaman, Keith Shine, Benjamin Lührs, and Florian Linke. Predicting the climate impact of aviation for en-route emissions: the algorithmic climate change function sub model ACCF 1.0 of EMAC 2.53. *GMDD*, 2022.

[3] Sigrun Matthes, Katrin Dahlmann, Simone Dietmüller, Hiroshi Yamashita, Volker Grewe, Manuel Soler, Abolfazl Simorgh, Daniel González Arribas, Florian Linke, Benjamin Lührs, Maximilian Meuser, Federica Castino, and Feijia Yin. Concept for identifying robust eco-efficient aircraft trajectories: methodological concept of climate-optimized aircraft trajectories in FlyATM4E. *Aerospace*, 2022, in preparation.

[4] David S Lee, DW Fahey, Agniezka Skowron, MR Allen, Ulrike Burkhardt, Q Chen, SJ Doherty, S Freeman, PM Forster, J Fuglestvedt, and others. The contribution of global aviation to anthropogenic climate forcing for 2000 to 2018. *Atmospheric Environment*, 244:117834, 2021.

PYTHON MODULE INDEX

C