Supplement of Geosci. Model Dev., 18, 253–286, 2025
https://doi.org/10.5194/gmd-18-253-2025-supplement

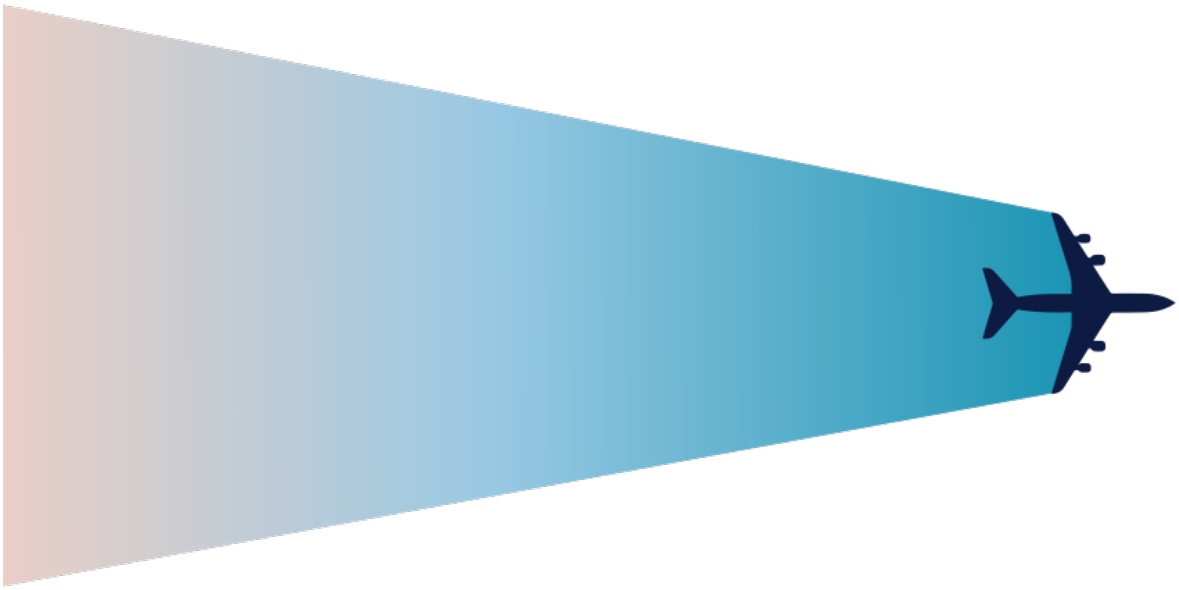
Geoscientific Model Development
Open Access
EGU

Supplement of

Forecasting contrail climate forcing for flight planning and air traffic management applications: the CocipGrid model in pycontrails 0.51.0

Zebediah Engberg et al.

Correspondence to: Marc L. Shapiro (marc.shapiro@breakthroughenergy.org)



pycontrails

Release 0.51.0

Breakthrough Energy

May 28, 2024

GETTING STARTED

Python library for modeling contrails and other aviation climate impacts.

Learn more on contrails.org.

Version	pypi v0.51.0 python 3.9 | 3.10 | 3.11 | 3.12
Citation	DOI 10.5281/zenodo.11283606
Tests	Unit tests passing Docs passing Release passing
License	license Apache-2.0
Community	discussions 19 total issues 9 open pull requests 5 open

CHAPTER

ONE

INSTALL

1.1 Requires

- Python (3.9 or later)

1.2 Core

Install the latest release using `pip`:

```
# core installation
$ pip install pycontrails

# install with all optional dependencies
$ pip install "pycontrails[complete]"
```

Install the latest development version directly from GitHub:

```
$ pip install git+https://github.com/contrailcirrus/pycontrails.git
```

1.3 Optional Dependencies

The `pycontrails` package uses optional dependencies for specific features:

```
# install all non-development optional dependencies
$ pip install "pycontrails[complete]"

# install specific optional dependencies
$ pip install "pycontrails[dev]"      # Development dependencies
$ pip install "pycontrails[docs]"     # Documentation / Sphinx dependencies
$ pip install "pycontrails[ecmwf]"    # ECMWF datalib interfaces
$ pip install "pycontrails[gcp]"      # Google Cloud Platform caching interface
$ pip install "pycontrails[gfs]"      # GFS datalib interfaces
$ pip install "pycontrails[jupyter]"  # Jupyter notebook and lab interface
$ pip install "pycontrails[vis]"      # Polygon construction methods and plotting support
$ pip install "pycontrails[zarr]"     # Load data from remote Zarr stores

# These packages may not support the latest python version
```

(continues on next page)

(continued from previous page)

```
# and are excluded from "complete"
$ pip install "pycontrails[open3d]"   # Polyhedra contruction methods
```

See `[project.optional-dependencies]` in pyproject.toml for the latest optional dependencies.

1.4 Pre-built wheels

Wheels for common platforms are available on PyPI. Currently, wheels are available for:

- Linux (x86_64)
- macOS (x86_64 and arm64)
- Windows (x86_64)

It's possible to build the wheels from source using Cython and a C compiler with the usual `pip install` process. The source distribution on PyPI includes the C files, so it's not necessary to have Cython installed to build from source.

1.5 Extensions

Some features of `pycontrails` are written as extensions that can be added manually:

1.5.1 BADA

This extension is private due to license restrictions

pycontrails-bada is an extension to interface with BADA aircraft performance data.

```
pip install "pycontrails-bada @ git+ssh://git@github.com/contrailcirrus/pycontrails-bada.
↪git"
```

1.5.2 Cirium

This extension is private due to license restrictions

pycontrails-cirium is an extension to the Cirium database of jet engines.

```
pip install "pycontrails-cirium @ git+ssh://git@github.com/contrailcirrus/pycontrails-
↪cirium.git"
```

1.5.3 ACCF

Interface to DLR / UMadrid ACCF model using a forked version of the climaccf repository.

```
pip install "climaccf @ git+ssh://git@github.com/contrailcirrus/climaccf.git"
```

CHAPTER

TWO

DEVELOP

2.1 Requires

- git
- Make. See Makefile for a list of `make` commands.

Developing documentation requires:

- pandoc for interpreting Jupyter notebooks
- LaTeX for pdf outputs. If you are using a Mac, MacTeX is the best option. Note that LaTeX can be fairly large to install (~6GB).

2.2 Environment

Create a dedicated virtual environment for development:

```
# create environment in <DIR>
$ python3 -m venv <DIR>

# activate environment (Unix-like)
$ source <DIR>/bin/activate
```

If using Anaconda / Miniconda Python, create a dedicated Anaconda environment:

```
# create conda environment
$ conda create -n contrails

# activate environment
$ conda activate contrails
```

2.3 Install

After activating the virtual environment, clone the pycontrails repository:

```
$ cd <install-path>
$ git clone git@github.com:contrailcirrus/pycontrails.git
$ cd pycontrails
```

These commands clone via SSH and may require adding an SSH key to your GitHub account. Alternatively, you can clone via HTTPS by running

```
$ cd <install-path>
$ git clone https://github.com/contrailcirrus/pycontrails.git
$ cd pycontrails
```

Install the development verison of `pycontrails` using `make`:

```
$ make dev-install
```

or install dependencies manually using `pip` in editable mode:

```
# core development installation
$ pip install -e ".[docs,dev]"

# install optional dependencies as above
$ pip install -e ".[ecmwf,gfs]"

# make sure to add pre-commit hooks if installing manually
$ pre-commit install
```

2.4 Test

Run all code quality checks and unit tests. This is run in the test workflow, but should also be run locally before submitting PRs:

```
$ make test
```

Lint the repository with ruff:

```
$ make ruff
```

Autoformat the repository with black:

```
$ make black
```

Run type checking with mypy:

```
$ make mypy
```

Run unit tests with pytest:

```
$ make pytest
```

Run notebook validation with nbval:

```
$ make nb-test
```

Run doctests with pytest:

```
$ make doctest
```

Notebook validation and doctests require Copernicus Climate Data Store (CDS) credentials, and doctests additionally require Google application credentials. If either are missing, the test suite will issue a warning and exit.

2.5 Documentation

Documentation is written in reStructuredText (rst) and built with Sphinx. The quick reStructuredText reference provides a decent rst syntax overview.

Sphinx includes many additional roles, directives, and extensions to enhance documentation.

Sphinx configuration is written in docs/conf.py. See the Sphinx configuration docs for the full list of configuration options.

Build HTML documentation:

```
# docs build to directory docs/_build/html
$ make docs-build

# automatically build docs on changes
# docs will be served at http://127.0.0.1:8000
$ make docs-serve

# clean up built documentation
$ make docs-clean
```

Build manually with `sphinx-build`:

```
$ sphinx-build -b html docs docs/_build/html      # HTML output
```

Sphinx caches builds between changes. To force the whole site to rebuild, use the options `-aE`:

```
$ sphinx-build -aE -b html docs docs/_build/html  # rebuild all output
```

See sphinx-build for a list of all the possible output builders.

2.5.1 Notebooks

Examples and tutorials should be written as isolated executable Jupyter Notebooks. The nbsphinx extension includes notebooks in the static documentation.

Notebooks will be automatically evaluated during tests, unless explicitly ignored. To exclude a notebook cell from evaluation during testing or automatic execution, add the tags `nbval-skip` and `skip-execution` to cell metadata.

To test notebooks locally, run:

```
$ make nb-test
```

To re-execute all notebooks, run:

```
$ make nb-execute
```

2.5.2 PDF Output

Building PDF output requires a LaTeX distribution.

Build pdf documentation:

```
$ make docs-pdf
```

A single pdf output (i.e. `pycontrails.pdf`) will be built within `docs/_build/latex`.

To build manually, run:

```
$ sphinx-build -b latex docs docs/_build/latex
$ cd docs/_build/latex
$ make
```

2.5.3 References

Literature references managed in the pycontrails Zotero library.

The documentation uses sphinxcontrib-bibtex to include citations and a bibliography.

All references should be cited through documentation and docstrings using the :cite: role.

To automatically sync the Zotero library with the docs/_static/pycontrails.bib Bibtex file:

- Install Zotero and add the pycontrails library.
- Install the Zotero Better Bibtex extension. Leave defaults during setup.
- Right click on the **pycontrails** library and select *Export Library*
- Export as *Better Bibtex*. You can optionally check *Keep Updated* if you want this file to update every time you make a change to the library.
- Select the file `_static/pycontrails.bib` and press *Save* to overwrite the file.
- Commit the updated `_static/pycontrails.bib`

2.5.4 Test

All doc tests first ensure ERA5 data is cached locally:

```
$ make ensure-era5-cached
```

Run rst linter with doc8:

```
$ make doc8
```

Run docstring example tests with doctest:

```
$ make doctest
```

Test notebook examples with nbval pytest plugin:

```
$ make nb-test
```

2.6 Conventions

2.6.1 Code

`pycontrails` aims to implement clear, consistent, performant data structures and models.

The project uses mypy for static type checking. All code should have specific, clear type annotations.

The project uses Black, ruff and doc8 to standardize code-formatting. These tools are run automatically in a pre-commit hook.

The project uses pytest to run unit tests. New code should include clear unit tests for implementation and output values. New test files should be included in the /tests/unit/ directory.

The project uses Github Actions to run code quality and unit tests on each pull request. Test locally before pushing using:

```
$ make test
```

2.6.2 Docstrings

Wherever possible, we adhere to the NumPy docstring conventions.

The following links are good references for writing *numpy* docstrings:

- numpydoc docstring guide
- pandas docstring guide
- scipy docstring guideline

General guidelines:

```
Use *italics*, **bold** and ``monospace`` if needed in any explanations
(but not for variable names and doctest code or multi-line code).
Variable, module, function, and class names
should be written between single back-ticks (`numpy`).
```

When specifying types in **Parameters** or **See Also**, Sphinx will automatically replace the text with the `napolean_type_aliases` specified in conf.py, e.g.

```
"""
Parameters
----------
x : np.ndarray
    Sphinx will automatically replace
    "np.ndarray" with the napolean type alias "numpy.ndarray"
"""
```

The **See Also** section is *not a list*. All of the following work:

```
"""
See Also
--------
:func:`segment_lengths`
segment_lengths
:class:`numpy.datetime64`
np.datetime64
"""
```

When you specify a type outside of **Parameters**, you have to use the sphinx cross-referencing syntax with the full module name:

```
"""
This is a :func:`pd.to_datetime`    # NO
and :func:`pandas.to_datetime`      # YES

This is a :class:`np.datetime64`    # NO
and :class:`numpy.datetime64`       # YES
"""
```

CHAPTER

THREE

RUNNING NOTEBOOKS

The /docs/notebooks directory in the repository contains Jupyter Notebooks demonstrating `pycontrails`.

3.1 Local

To run these notebooks locally, clone or download the repository, enter the `docs` directory, and launch `jupyter lab`:

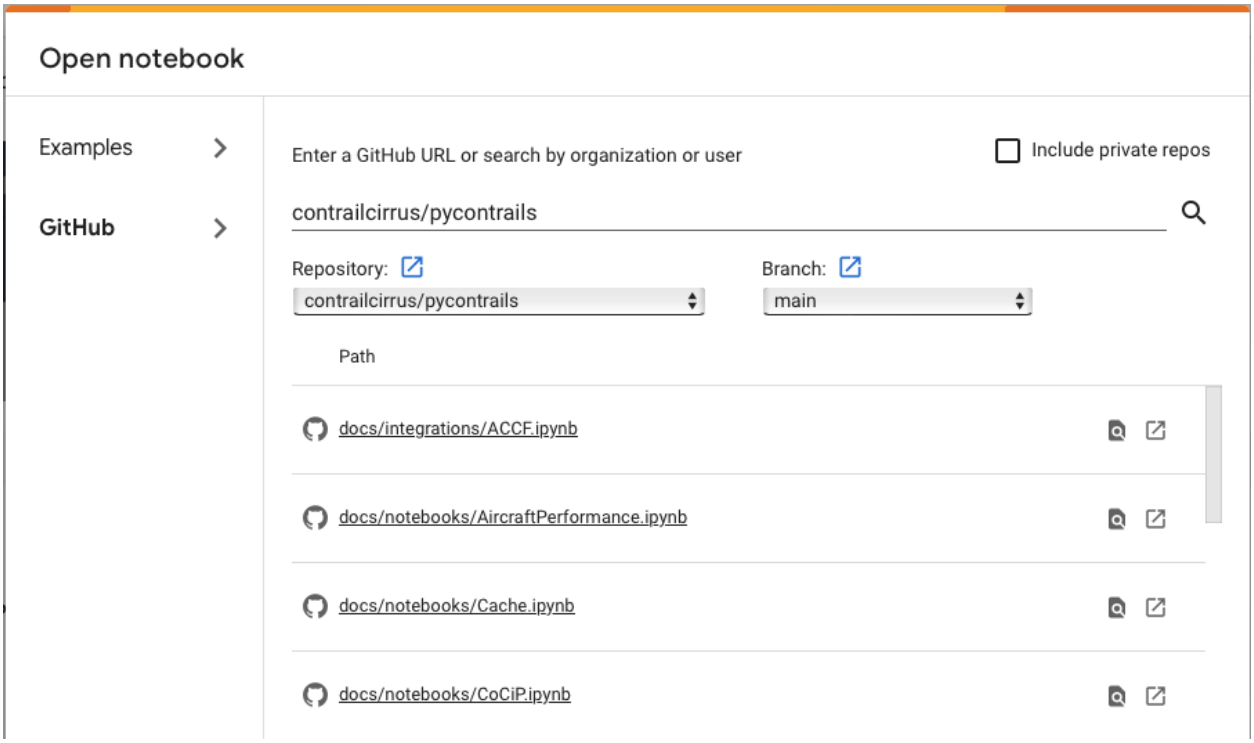
```
$ git clone https://github.com/contrailcirrus/pycontrails.git
$ cd pycontrails/docs
$ jupyter lab        # or notebook
```

If you don't have Jupyter installed, install the optional `pycontrails[jupyter]` dependencies:

```
$ pip install "pycontrails[jupyter]"  # Jupyter notebook and lab interface
```

3.2 Colab

Notebooks can also be opened in Google Colab. Select *File* -> *Open* -> **Github**, enter `contrailcirrus/pycontrails`, then select the Notebook to open.



Alternatively, any notebook directly with the url `https://colab.research.google.com/github/` + the path to the notebook file on github.com (e.g. CoCiP.ipynb)

`pycontrails` must be installed into Colab environment by adding the top cell:

```
!pip install pycontrails
```

Credentials and data must be added to each Colab environment. Colab integrates with Google Drive to load and save data and store secrets.

3.3 Data

3.3.1 ERA5

Many of these example notebooks make use of ERA5 met data. To avoid waiting for repeated CDS requests to load, it's best to request the full dataset up front (the full ERA5 request downloads and caches ~1GB of meteorology data).

For compatibility with the `pycontrails` cache, you can initiate this request through the `pycontrails` data interface. The snippet below coincides with the ERA5 request in CoCiP.ipynb. We recommend that you either run the `CoCiP.ipynb` notebook first, or run the snippet below before interacting with other notebooks.

```
from pycontrails.datalib.ecmwf import ERA5

time = ("2022-03-01 00:00:00", "2022-03-01 23:00:00")
pressure_levels = [350, 300, 250, 225, 200, 175, 150]
met_variables = ["t", "q", "u", "v", "w", "ciwc", "z", "cc"]
rad_variables = ["tsr", "ttr"]

ERA5(time=time, variables=met_variables, pressure_levels=pressure_levels).open_
```

(continues on next page)

(continued from previous page)

```
↪metdataset()
ERA5(time=time, variables=rad_variables).open_metdataset()
```

3.3.2 OpenSky

Because persistent contrails are a sparse phenomenon, it is a nontrivial task to construct authentic flights exhibiting characteristics from the vantage of contrail research. The data in the data/flight.csv file was constructed from the OpenSky database. We document the process here to ensure reproducibility.

OpenSky provides access to an impala shell to query their database of ADS-B flight data. The query below identifies flights at low altitude in the hour after 2022-03-01T00 (1646092800). Such flights are expected to be in an initial climb phase or a terminal descent phase. The two cases can be distinguished by consider the `vertrate`.

```
SELECT icao24, SUM(vertrate)
FROM state_vectors_data4
WHERE hour BETWEEN 1646092800 AND 1646092800 + 3600
    AND lon BETWEEN -80 and -30
    AND lat BETWEEN 30 and 40
    AND baroaltitude < 2000
GROUP BY icao24;
```

The query below selects one of the flights from the first output, keeping one waypoint at the start of each minute (the OpenSky database typically contains waypoint data with 1 second frequency.)

```
SELECT *
FROM state_vectors_data4
WHERE hour BETWEEN 1646092800 AND 1646092800 + 8 * 3600
    AND icao24 = 'acdd1b'
    AND time % 60 = 0
    AND baroaltitude IS NOT NULL
ORDER BY time;
```

The OpenSky impala shell simply streams text data over SSH. To convert to a CSV, the output of the impala shell can be piped (or copy-pasted) into the `query_output.txt` text file referenced below. The pandas code below converts the output of the above query to the `data/flight.csv` file included here.

```
import pandas as pd
df = pd.read_csv("query_output.txt", sep="|", skiprows=[0, 2], skipfooter=1, engine=
↪"python")
df = df.loc[:, ~df.columns.str.startswith("Unnamed")]
df.columns = df.columns.str.strip()

df = df.rename(columns={"lon": "longitude", "lat": "latitude", "baroaltitude": "altitude
↪"})
df["time"] = pd.to_datetime(df["time"], unit="s")
df = df[["longitude", "latitude", "altitude", "time"]]

# artificially clip at 38000 ft to ensure we stay within met bounds
df["altitude"] = df["altitude"].clip(upper=11582.4)
df.to_csv("data/flight.csv", index=False)
```

CHAPTER

FOUR

FLIGHT DATA

See *Running Notebooks* to interact with these notebooks

4.1 Load flight data

Create `Flight` data structure for working with flight trajectories.

[1]:
```
import pandas as pd
import numpy as np

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
```

4.1.1 Create Flight instance

From Numpy Arrays

[2]:
```
# waypoints
longitude = np.linspace(0, 50, 100)
latitude = np.linspace(0, 10, 100)
altitude = np.linspace(11000, 11500, 100)
time = pd.date_range("2022-03-01 00:00:00", "2022-03-01 02:00:00", periods=100)

fl = Flight(longitude=longitude, latitude=latitude, altitude=altitude, time=time, flight_
↪id="id")
fl
```

[2]:
```
Flight [4 keys x 100 length, 2 attributes]
    Keys: longitude, latitude, time, altitude
    Attributes:
    time                [2022-03-01 00:00:00, 2022-03-01 02:00:00]
    longitude           [0.0, 50.0]
    latitude            [0.0, 10.0]
    altitude            [11000.0, 11500.0]
    flight_id           id
    crs                 EPSG:4326
```

From Pandas DataFrame

[3]:
```
# Example flight
df = pd.DataFrame()
df["longitude"] = np.linspace(0, 50, 100)
df["latitude"] = np.linspace(0, 10, 100)
df["altitude"] = 11000
df["time"] = pd.date_range("2022-03-01 00:00:00", "2022-03-01 02:00:00", periods=100)
fl = Flight(data=df, flight_id="ABC")
fl
```

[3]:
```
Flight [4 keys x 100 length, 2 attributes]
    Keys: longitude, latitude, altitude, time
    Attributes:
    time                [2022-03-01 00:00:00, 2022-03-01 02:00:00]
    longitude           [0.0, 50.0]
    latitude            [0.0, 10.0]
    altitude            [11000.0, 11000.0]
    flight_id           ABC
    crs                 EPSG:4326
```

Create Flight without Waypoints

[4]:
```
# Example flight
attrs = dict(flight_id="1234", equip="A532")
fl = Flight.create_empty(attrs=attrs)
fl
```

[4]:
```
Flight [4 keys x 0 length, 3 attributes]
    Keys: longitude, latitude, time, altitude
    Attributes:
    flight_id           1234
    equip               A532
    crs                 EPSG:4326
```

Create from CSV file

[5]:
```
# load flight
df = pd.read_csv("data/flight.csv")
fl = Flight(data=df, flight_id="csv")
fl
```

[5]:
```
Flight [4 keys x 175 length, 2 attributes]
    Keys: longitude, latitude, altitude, time
    Attributes:
    time                [2022-03-01 00:50:00, 2022-03-01 03:47:00]
    longitude           [-97.026, -77.036]
    latitude            [32.931, 38.854]
    altitude            [190.5, 11582.4]
    flight_id           csv
    crs                 EPSG:4326
```

4.1.2 Using the Flight instance

The `flight.data` attribute is a dictionary with `np.ndarray` values

[6]:
```
# waypoints
longitude = np.linspace(0, 50, 10)
latitude = np.linspace(0, 10, 10)
altitude = np.linspace(11000, 11500, 10)
time = pd.date_range("2022-03-01 00:00:00", "2022-03-01 02:00:00", periods=10)
attrs = {"flight_id": "ABC123"}
fl = Flight(longitude=longitude, latitude=latitude, altitude=altitude, time=time,␣
↪attrs=attrs)
```

[7]:
```
fl.data.keys()
```

[7]:
```
dict_keys(['longitude', 'latitude', 'time', 'altitude'])
```

`Flight` attributes are stored in a dictionary on the `attrs` attribute. The `crs` attribute is always added by default, if not specified.

[8]:
```
fl.attrs
```

[8]:
```
{'flight_id': 'ABC123', 'crs': 'EPSG:4326'}
```

Data can be set / get from the `Flight` like a dictionary

[9]:
```
# get
lat = fl["latitude"]

# set
lat[5] = 20
fl["latitude"] = lat

# get updated
fl["latitude"][5]
```

[9]:
```
20.0
```

The `Flight` class contains the following convenience properties

[10]:
```
# Pressure altitude, in hPa
fl.level
```

[10]:
```
array([226.3170091 , 224.34300442, 222.3862176 , 220.44649846,
       218.52369813, 216.61766904, 214.7282649 , 212.85534072,
       210.99875274, 209.15835847])
```

[11]:
```
# Altitude, in ft
fl.altitude_ft
```

[11]:
```
array([36089.23884514, 36271.5077282 , 36453.77661126, 36636.04549431,
       36818.31437737, 37000.58326043, 37182.85214348, 37365.12102654,
       37547.38990959, 37729.65879265])
```

[12]:
```
# Values that are constant along the flight path
```

(continues on next page)

(continued from previous page)

```
# set constant value along flight waypoints
fl["constant"] = np.full(shape=fl.shape, fill_value=100)

fl.constants
```

[12]:
```
{'constant': 100, 'flight_id': 'ABC123', 'crs': 'EPSG:4326'}
```

[13]:
```
# Flight distance, in meters
fl.length
```

[13]:
```
7818835.115366629
```

[14]:
```
# Time start/end
print(fl.time_start)
print(fl.time_end)
```

```
2022-03-01 00:00:00
2022-03-01 02:00:00
```

[15]:
```
# Flight duration, as a pandas Timedelta
fl.duration
```

[15]:
```
Timedelta('0 days 02:00:00')
```

[16]:
```
# Max time gap between waypoints, as a pandas Timedelta
fl.max_time_gap
```

[16]:
```
Timedelta('0 days 00:13:20')
```

[17]:
```
# Max distance gap between waypoints, in meters
fl.max_distance_gap
```

[17]:
```
1831403.3492360476
```

4.1.3 Intersect with Met data

[18]:
```
# waypoints
longitude = np.linspace(0, 50, 50)
latitude = np.linspace(0, 10, 50)
altitude = np.linspace(11000, 11500, 50)
time = pd.date_range("2022-03-01 00:00:00", "2022-03-01 02:00:00", periods=50)
fl = Flight(longitude=longitude, latitude=latitude, altitude=altitude, time=time, flight_
↪id="ABC")
```

[19]:
```
# domain
time = ("2022-03-01 00:00:00", "2022-03-01 03:00:00")
variables = ["t", "q", "u", "v", "w", "ciwc", "z", "cc"]
pressure_levels = [300, 250, 200]

# get met data
era5 = ERA5(time=time, variables=variables, pressure_levels=pressure_levels)
met = era5.open_metdataset()
```

[20]:
```
# interpolate to nearest grid member
fl.intersect_met(met["air_temperature"], method="nearest")
```

[20]:
```
array([231.6297 , 231.5037 , 231.73001, 231.08833, 218.8252 , 218.8393 ,
       219.00676, 218.07823, 218.66353, 219.61859, 219.48346, 219.66501,
       219.8988 , 219.86067, 219.95186, 220.16989, 220.15166, 220.06793,
       219.71309, 219.5191 , 219.5647 , 219.54646, 219.39972, 219.1908 ,
       219.0192 , 219.02084, 219.36572, 219.42708, 219.39308, 219.49672,
       219.63931, 219.63351, 219.80347, 219.72304, 218.88406, 218.68758,
       219.5904 , 219.66667, 219.75786, 219.57713, 219.53734, 219.87642,
       219.74045, 219.47931, 219.345  , 219.28697, 219.14354, 219.04074,
       218.90147, 218.93463], dtype=float32)
```

[21]:
```
# linear interpolation
fl.intersect_met(met["air_temperature"], method="linear")
```

[21]:
```
array([225.77794, 225.39075, 225.21887, 224.88387, 225.13017, 224.75333,
       224.69849, 224.51218, 224.65642, 225.06851, 225.2779 , 225.23737,
       225.27412, 225.21637, 225.02275, 225.16205, 225.02489, 224.83875,
       224.50832, 224.36188, 224.2705 , 224.0278 , 223.71686, 223.553  ,
       223.37677, 223.35237, 223.489  , 223.45067, 223.46031, 223.51036,
       223.47456, 223.33977, 223.33707, 223.19572, 222.68132, 222.43277,
       222.99377, 222.90991, 222.94348, 222.70607, 222.66989, 222.84317,
       222.59155, 222.31973, 222.06894, 221.91066, 221.70157, 221.5247 ,
       221.33913, 221.26541], dtype=float32)
```

4.1.4 Get Lengths

[22]:
```
# total flight length in meters
fl.length
```

[22]:
```
5642421.5973290345
```

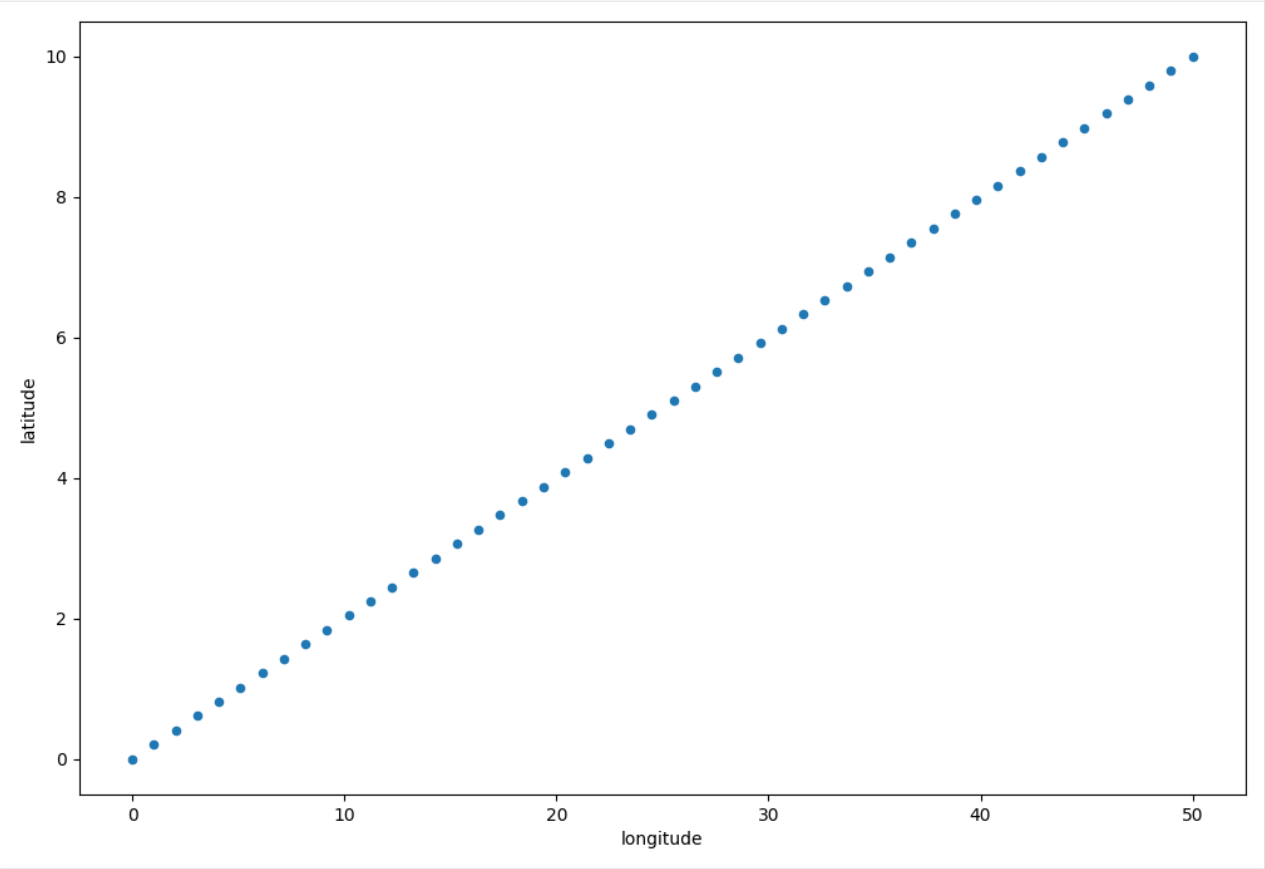
[23]:
```
# intersect flight with air temperature
fl["temp"] = fl.intersect_met(met["air_temperature"], method="nearest")

# get the length of the flight where ambient temperature is > 226 K
fl.length_met("temp", threshold=226)
```

[23]:
```
462850.7958842697
```

4.1.5 Plot and Resample

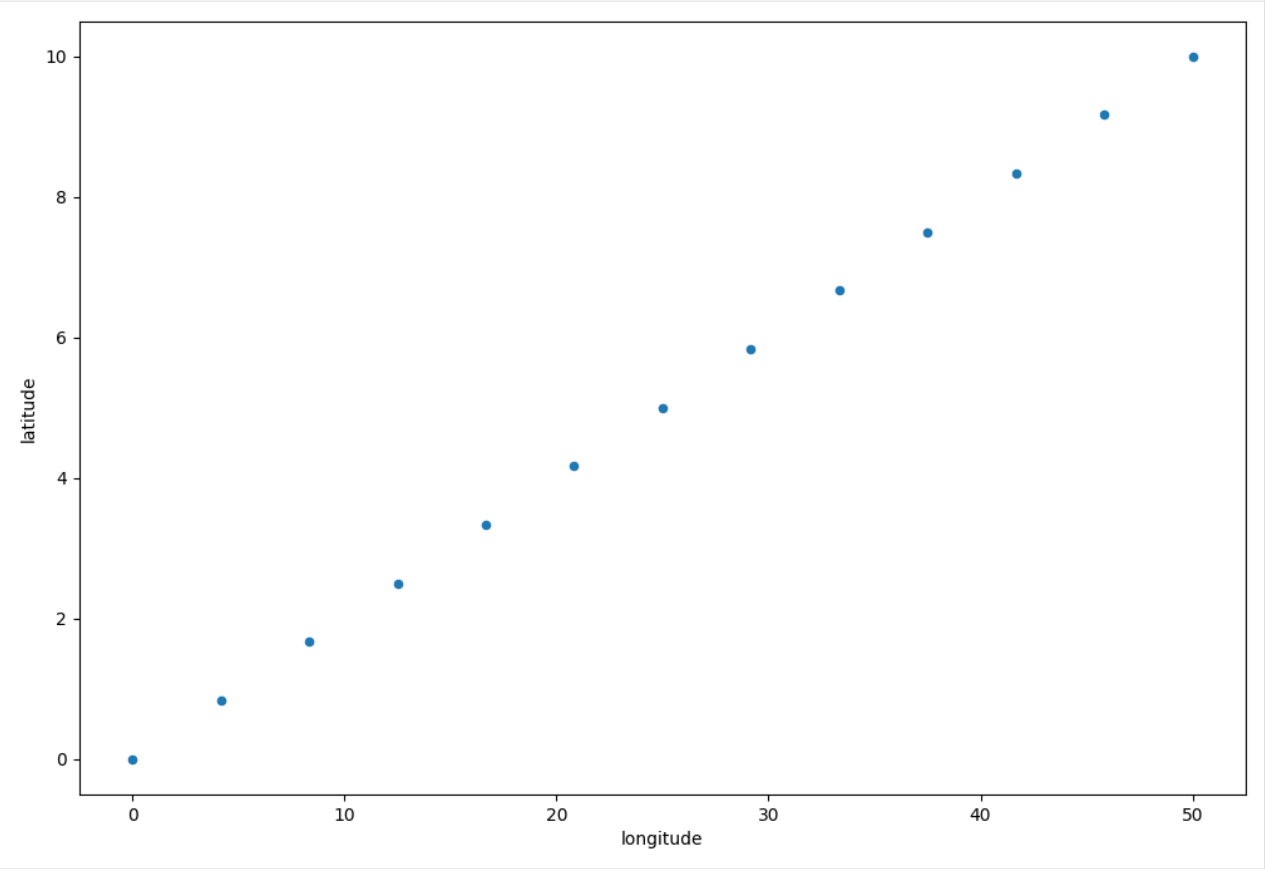
[24]:
```
fl.dataframe.plot.scatter(x="longitude", y="latitude", figsize=(12, 8))
fl.max_distance_gap
```

[24]:
```
115715.16936791067
```



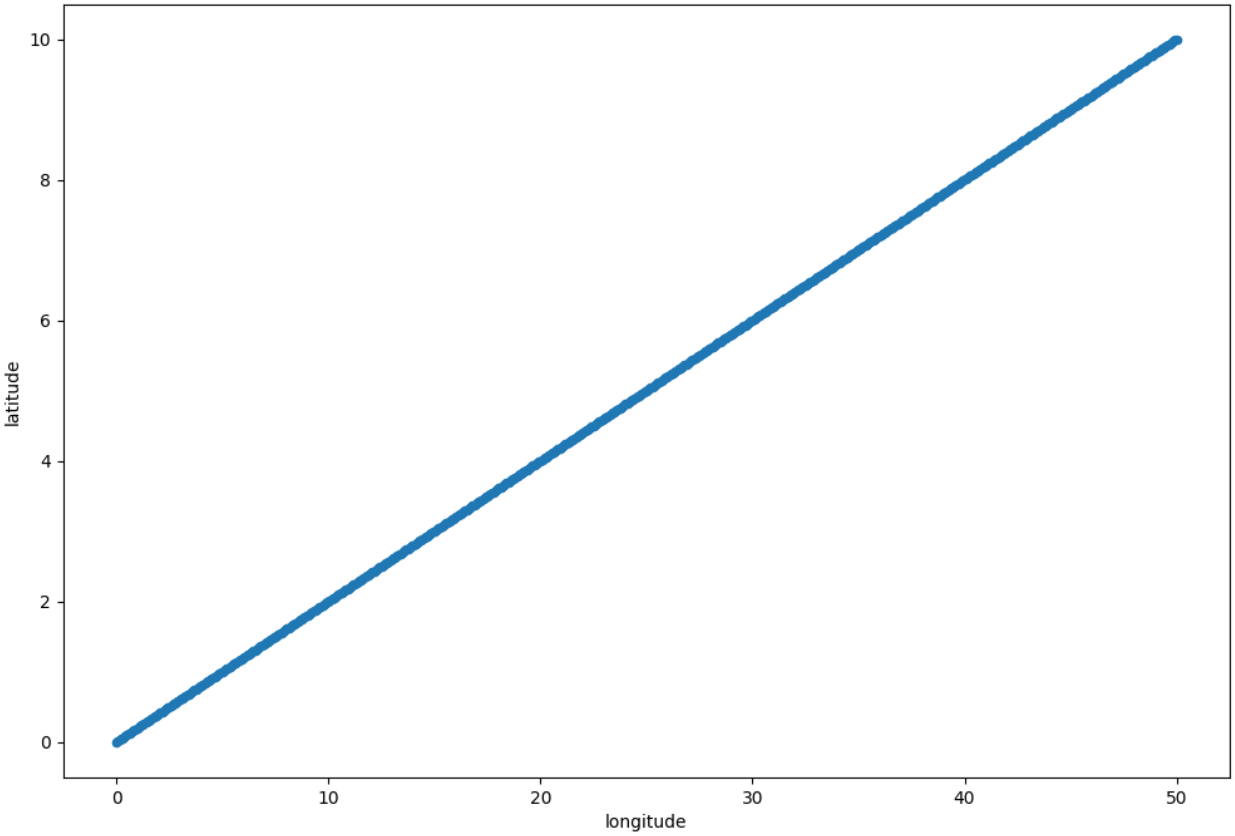
[25]:
```
# resample with 10 minute waypoints
fl = fl.resample_and_fill("10min")

fl.dataframe.plot.scatter(x="longitude", y="latitude", figsize=(12, 8))
fl.max_distance_gap
```

[25]:
```
472488.41576537804
```



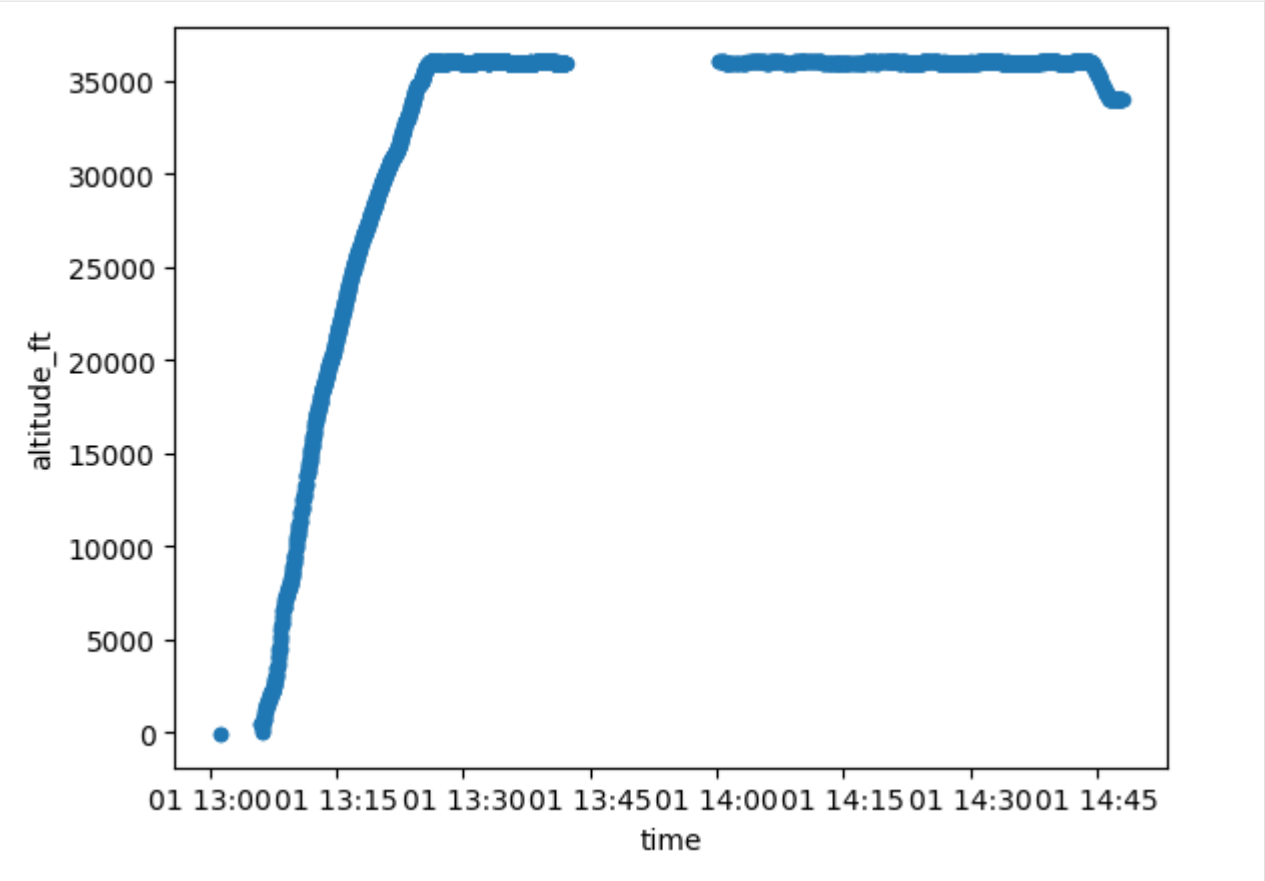
[26]:
```
# resample with 10 second waypoints
fl = fl.resample_and_fill("10s")

fl.dataframe.plot.scatter(x="longitude", y="latitude", figsize=(12, 8))
fl.max_distance_gap
```

[26]:
```
7874.969421029368
```

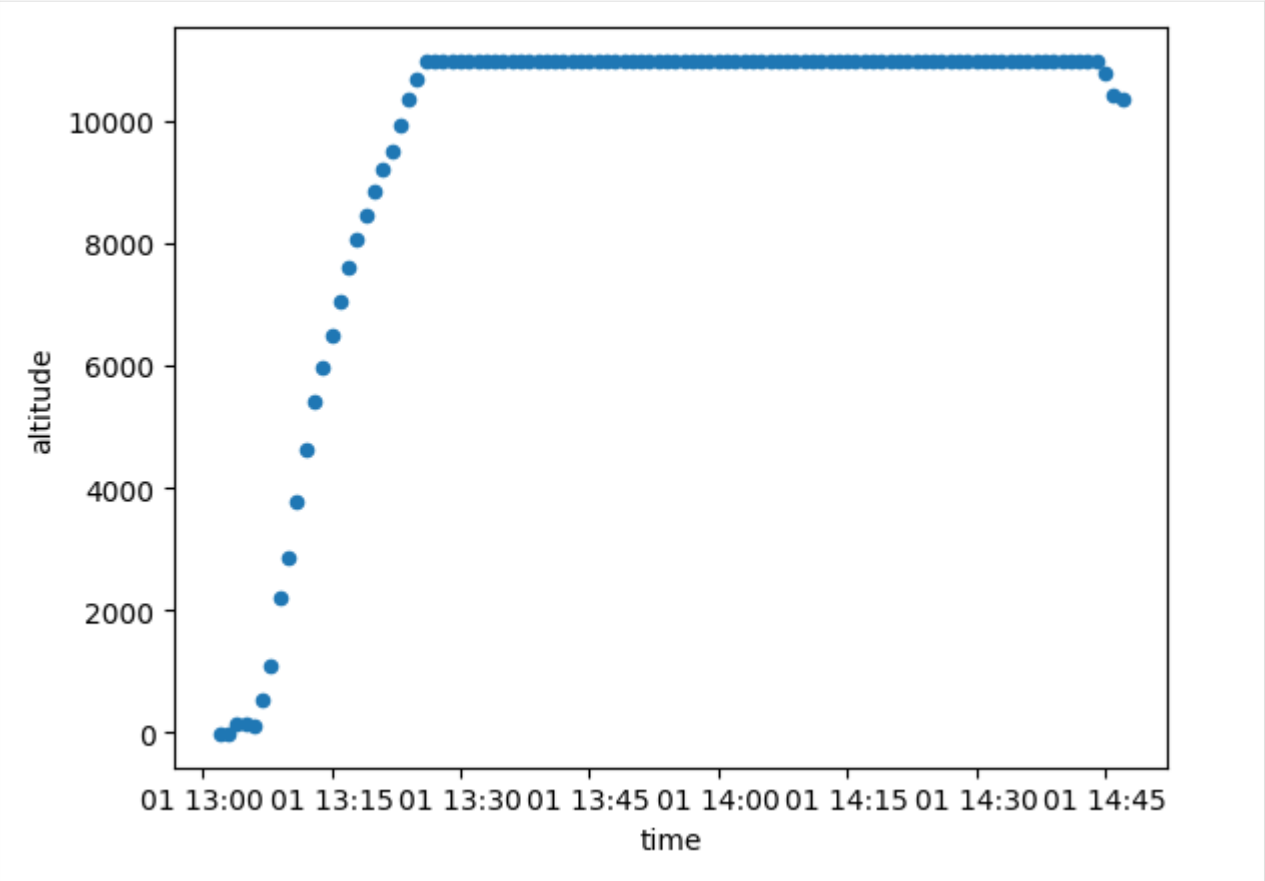


4.1.6 Clean and smooth a noisy flight with gaps

[27]:
```
df = pd.read_csv("data/flight-noisy.csv")
df["time"] = df["time"].map(lambda t: np.datetime64(t))
f = Flight(df, drop_duplicated_times=True)
_ = f.dataframe.plot.scatter(x="time", y="altitude_ft")
```



[28]:
```
f = f.clean_and_resample(nominal_rocd=20)
_ = f.dataframe.plot.scatter(x="time", y="altitude")
```



4.2 Access airport data

The `pycontrails.core.airports` module provides access to global airport data and helper functions for working with airport data.

Airport data is loaded at runtime from Our Airports from a fork of the ourairports-data repository.

[1]:
```
from pycontrails.core import airports
```

4.2.1 Download global airport database

The `global_airport_database` method downloads and caches airport data.

[2]:
```
db = airports.global_airport_database()
db.head()
```

[2]:

	type	name	latitude	longitude	\
0	small_airport	Rothera Point Airport	-67.566941	-68.126993	
1	small_airport	Angeles City Flying Club	15.254326	120.677772	
2	small_airport	Afutara Aerodrome	-9.191389	160.948611	
3	small_airport	Ulawa Airport	-9.860544	161.979547	
4	small_airport	Uru Harbour Airport	-8.873330	161.011002	

(continues on next page)

(continued from previous page)

```
   elevation_ft iso_country iso_region   municipality scheduled_service  \
0           0.0          AQ     AQ-U-A  Rothera Point                no
1         100.0          PH     PH-U-A             MG                no
2          23.0          SB      SB-ML           Bila               yes
3          40.0          SB      SB-MK          Arona                no
4           0.0          SB      SB-ML         Atoifi               yes

  icao_code iata_code  elevation_m
0      AAXX        NaN       0.0000
1      ACFC        AFC      30.4800
2      AGAF        AFT       7.0104
3      AGAR        RNA      12.1920
4      AGAT        ATD       0.0000
```

You can override the cache location by passing in a pycontrails CacheStore to the `cachestore` argument:

[3]:
```
from pycontrails import DiskCacheStore
```

[4]:
```
cache = DiskCacheStore(cache_dir="data/airports")
db = airports.global_airport_database(cachestore=cache)
db.head()
```

[4]:
```
            type                      name   latitude   longitude  \
0  small_airport     Rothera Point Airport -67.566941  -68.126993
1  small_airport  Angeles City Flying Club  15.254326  120.677772
2  small_airport         Afutara Aerodrome  -9.191389  160.948611
3  small_airport             Ulawa Airport  -9.860544  161.979547
4  small_airport       Uru Harbour Airport  -8.873330  161.011002

   elevation_ft iso_country iso_region   municipality scheduled_service  \
0           0.0          AQ     AQ-U-A  Rothera Point                no
1         100.0          PH     PH-U-A             MG                no
2          23.0          SB      SB-ML           Bila               yes
3          40.0          SB      SB-MK          Arona                no
4           0.0          SB      SB-ML         Atoifi               yes

  icao_code iata_code  elevation_m
0      AAXX        NaN       0.0000
1      ACFC        AFC      30.4800
2      AGAF        AFT       7.0104
3      AGAR        RNA      12.1920
4      AGAT        ATD       0.0000
```

You can force an update of the cached airports by passing in `update_cache=True`:

[5]:
```
# redownloads airport database
db = airports.global_airport_database(update_cache=True)
```

4.2.2 Find nearest airport

Find the nearest airport to waypoint location

[6]:
```
# a point in the Massachusetts bay close to BOS
longitude = -70.842554
latitude = 42.387466
altitude = 1000

# open the airport database
db = airports.global_airport_database()

# returns the icao code of the closest airport
airports.find_nearest_airport(db, longitude, latitude, altitude)
```

[6]:
```
'KBOS'
```

The function returns `None` if no airport is found within the default bounding box (2°)

[7]:
```
# Somewhere in the atlantic
longitude = -50
latitude = 35
altitude = 1000

airports.find_nearest_airport(db, longitude, latitude, altitude)
```

Pass in the `bbox` parameter (in units of °) to increase the search distance

[8]:
```
airports.find_nearest_airport(db, longitude, latitude, altitude, bbox=20)
```

[8]:
```
'CYYT'
```

4.2.3 Find distance to airports

Calculate the haversine distance (in `m`) to all input airports

[9]:
```
db = airports.global_airport_database()

# select BOS and JFK
subset = db[db["iata_code"].isin(["BOS", "JFK"])]

# Find distance to a point in the massachusetts bay close to BOS
longitude = -70.842554
latitude = 42.387466
altitude = 500

airports.distance_to_airports(subset, longitude, latitude, altitude)
```

[9]:
```
array([ 13616.14944594, 312341.96236736])
```

4.3 Parse a flight plan

The `pycontrails.core.flightplan` module provides utilities for working with standard flight plan formats.

Only ICAO ATC flight plan supported currently.

4.3.1 References

- ICAO Doc 4444 ATM/501 - Procedures for Air Navigation Services (Appendix 2)
- FAA Form 7233-4 - International Flight Plan

[1]:
```
from pycontrails.core import flightplan
```

[2]:
```
atc_plan = """
(FPL-GEC8145-IN -B77L/H-SDE2E3FGHIJ3J4J5M1RWXYZ/SB1D1
-EGGL1040 -N0474F360 IMVUR1Z IMVUR N63 SAM N19 ADKIK DCT
MOPAT DCT  LIMRI/M083F360 DCT 51N020W 47N030W/M083F380 40N040W
34N045W  28N050W/M083F400 24N055W 19N060W DCT AMTTO DCT ANU DCT
-PBN/A1B1C1D1L1O1S1S2 NAV/RNVD1E2A1 DAT/SVM DOF/140501 REG/DALFA
EET/OKAC0037 ORBB0052 LTAA0159 UKFV0308 UKOV0333 LUUU0344 UKLV0406
EPWW0427 ESAA0521 EKDK0540 ENOR0557 SEL/DFBH OPR/GEC RVR/200)
-E/0740 P/3 R/E S/ J/ A/WHITE BLUE TAIL
"""
```

[3]:
```
# parse flight plan into a dictionary
plan_dict = flightplan.parse_atc_plan(atc_plan)
plan_dict
```

[3]:
```
{'callsign': 'GEC8145',
 'flight_rules': 'I',
 'type_of_flight': 'N',
 'type_of_aircraft': 'B77L',
 'wake_category': 'H',
 'equipment': 'SDE2E3FGHIJ3J4J5M1RWXYZ',
 'transponder': 'SB1D1',
 'departure_icao': 'EGGL',
 'time': '1040',
 'speed_type': 'N',
 'speed': '0474',
 'level_type': 'F',
 'level': '360',
 'route': 'IMVUR1Z IMVUR N63 SAM N19 ADKIK DCT MOPAT DCT  LIMRI/M083F360 DCT 51N020W␣
↪47N030W/M083F380 40N040W 34N045W  28N050W/M083F400 24N055W 19N060W DCT AMTTO DCT ANU␣
↪DCT',
 'other_info': 'E/0740 P/3 R/E S/ J/ A/WHITE BLUE TAIL'}
```

[4]:
```
# write flight plan dictionary back to a string
flightplan.to_atc_plan(plan_dict)
```

[4]:
```
'(FPL-GEC8145-IN\n-B77L/H-SDE2E3FGHIJ3J4J5M1RWXYZ/SB1D1\n-EGGL1040\n-N0474F360 IMVUR1Z␣
↪IMVUR N63 SAM N19 ADKIK DCT MOPAT DCT  LIMRI/M083F360 DCT 51N020W 47N030W/M083F380␣
```

(continues on next page)

(continued from previous page)

```
↪40N040W 34N045W  28N050W/M083F400 24N055W 19N060W DCT AMTTO DCT ANU DCT\n\n-E/0740 P/3␣
↪R/E S/ J/ A/WHITE BLUE TAIL)'
```

CHAPTER

FIVE

METEOROLOGY

See *Running Notebooks* to interact with these notebooks

5.1 Create meteorology data

Create `MetDataset` and `MetDataArray` data structures for working with meteorology data.

[1]:
```
from datetime import datetime

import xarray as xr
import numpy as np

from pycontrails import MetDataset, MetDataArray, MetVariable
```

5.1.1 MetDataArray

Thin wrapper around xr.DataArray

[2]:
```
rng = np.random.default_rng(seed=2020)

# construct an xarray DataArray with coordinate labels for data
da = xr.DataArray(
    name="random",
    data=rng.random((20, 15, 4, 5)),
    dims=["longitude", "latitude", "level", "time"],
    coords={
        "longitude": np.arange(-100, -80, 1.0),
        "latitude": np.arange(30, 45, 1.0),
        "level": np.arange(100, 500, 100),
        "time": [datetime(2021, 1, 1, h) for h in range(5)],
    },
)
da
```

[2]:
```
<xarray.DataArray 'random' (longitude: 20, latitude: 15, level: 4, time: 5)>
array([[[[4.68307543e-01, 5.14342231e-01, 8.63988281e-01,
          7.19386871e-01, 3.33497882e-01],
         [8.81663616e-01, 5.18664864e-01, 5.23219417e-01,
          7.22388696e-01, 4.46782561e-01],
```

(continues on next page)

(continued from previous page)

```
         [8.50357081e-01, 6.83936263e-01, 6.65900069e-01,
          9.46717074e-01, 7.53638850e-01],
         [9.17964165e-02, 8.26090137e-01, 5.55186777e-01,
          2.31152608e-01, 9.82958913e-01]],

        [[5.84613086e-01, 4.26518507e-01, 8.45336094e-01,
          5.07850470e-01, 7.10216629e-01],
         [1.37352673e-01, 4.84569777e-01, 8.89744202e-01,
          9.52976849e-01, 3.76043941e-02],
         [9.54223457e-01, 8.98512171e-01, 4.14081276e-01,
          4.20221061e-01, 6.91400341e-02],
         [4.21756867e-01, 4.93848391e-01, 1.51696284e-01,
          9.41137061e-01, 1.88116189e-01]],

        [[1.90065507e-02, 9.44932088e-01, 9.93219943e-01,
          6.81811974e-01, 7.84335294e-01],
...
         [8.92843751e-01, 4.07684237e-01, 3.05168469e-01,
          1.73576037e-01, 3.34321840e-01]],

        [[7.61263799e-01, 9.91563651e-01, 9.47543921e-01,
          6.68735461e-01, 2.71486792e-01],
         [2.05356394e-01, 7.42337468e-01, 2.70578729e-01,
          7.15520327e-01, 7.30134504e-01],
         [6.68886804e-01, 5.15651993e-01, 8.40987212e-01,
          5.37939968e-01, 3.71399439e-01],
         [1.67243220e-01, 9.76645528e-01, 3.08876242e-01,
          8.19142068e-01, 9.69391866e-01]],

        [[4.44617360e-01, 5.02274334e-01, 2.32060554e-01,
          3.54429329e-01, 1.36471633e-01],
         [6.30717612e-01, 2.40955809e-01, 2.54390888e-01,
          7.20384123e-01, 3.42038201e-01],
         [8.20903975e-01, 6.73148542e-01, 8.67869039e-01,
          4.79726457e-01, 8.34115517e-01],
         [3.59797874e-01, 7.61370564e-01, 4.16116300e-01,
          7.46702020e-01, 6.39419576e-01]]]])
Coordinates:
  * longitude  (longitude) float64 -100.0 -99.0 -98.0 ... -83.0 -82.0 -81.0
  * latitude   (latitude) float64 30.0 31.0 32.0 33.0 ... 41.0 42.0 43.0 44.0
  * level      (level) int64 100 200 300 400
  * time       (time) datetime64[ns] 2021-01-01 ... 2021-01-01T04:00:00
```

[3]:
```
met = MetDataArray(da)
met
```

[3]:
```
MetDataArray with data:

<xarray.DataArray 'random' (longitude: 20, latitude: 15, level: 4, time: 5)>
array([[[[4.68307543e-01, 5.14342231e-01, 8.63988281e-01,
          7.19386871e-01, 3.33497882e-01],
         [8.81663616e-01, 5.18664864e-01, 5.23219417e-01,
```

(continues on next page)

(continued from previous page)

```
          7.22388696e-01, 4.46782561e-01],
         [8.50357081e-01, 6.83936263e-01, 6.65900069e-01,
          9.46717074e-01, 7.53638850e-01],
         [9.17964165e-02, 8.26090137e-01, 5.55186777e-01,
          2.31152608e-01, 9.82958913e-01]],

        [[5.84613086e-01, 4.26518507e-01, 8.45336094e-01,
          5.07850470e-01, 7.10216629e-01],
         [1.37352673e-01, 4.84569777e-01, 8.89744202e-01,
          9.52976849e-01, 3.76043941e-02],
         [9.54223457e-01, 8.98512171e-01, 4.14081276e-01,
          4.20221061e-01, 6.91400341e-02],
         [4.21756867e-01, 4.93848391e-01, 1.51696284e-01,
          9.41137061e-01, 1.88116189e-01]],

        [[1.90065507e-02, 9.44932088e-01, 9.93219943e-01,
          6.81811974e-01, 7.84335294e-01],
...
         [8.92843751e-01, 4.07684237e-01, 3.05168469e-01,
          1.73576037e-01, 3.34321840e-01]],

        [[7.61263799e-01, 9.91563651e-01, 9.47543921e-01,
          6.68735461e-01, 2.71486792e-01],
         [2.05356394e-01, 7.42337468e-01, 2.70578729e-01,
          7.15520327e-01, 7.30134504e-01],
         [6.68886804e-01, 5.15651993e-01, 8.40987212e-01,
          5.37939968e-01, 3.71399439e-01],
         [1.67243220e-01, 9.76645528e-01, 3.08876242e-01,
          8.19142068e-01, 9.69391866e-01]],

        [[4.44617360e-01, 5.02274334e-01, 2.32060554e-01,
          3.54429329e-01, 1.36471633e-01],
         [6.30717612e-01, 2.40955809e-01, 2.54390888e-01,
          7.20384123e-01, 3.42038201e-01],
         [8.20903975e-01, 6.73148542e-01, 8.67869039e-01,
          4.79726457e-01, 8.34115517e-01],
         [3.59797874e-01, 7.61370564e-01, 4.16116300e-01,
          7.46702020e-01, 6.39419576e-01]]]])
Coordinates:
  * longitude     (longitude) float64 -100.0 -99.0 -98.0 ... -83.0 -82.0 -81.0
  * latitude      (latitude) float64 30.0 31.0 32.0 33.0 ... 41.0 42.0 43.0 44.0
  * level         (level) float64 100.0 200.0 300.0 400.0
  * time          (time) datetime64[ns] 2021-01-01 ... 2021-01-01T04:00:00
    air_pressure  (level) float64 1e+04 2e+04 3e+04 4e+04
    altitude      (level) float64 1.618e+04 1.178e+04 9.164e+03 7.185e+03
```

[4]:
```
# data attribute contains copy of the original xr.DataArray
isinstance(met.data, xr.DataArray)
```

[4]:
```
True
```

[5]:
```
# broadcast coords to the DataArray
met.broadcast_coords("air_pressure")
```

[5]:
```
<xarray.DataArray 'air_pressure' (longitude: 20, latitude: 15, level: 4, time: 5)>
array([[[[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        ...,

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
...
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        ...,

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]]]])
Coordinates:
  * longitude     (longitude) float64 -100.0 -99.0 -98.0 ... -83.0 -82.0 -81.0
  * latitude      (latitude) float64 30.0 31.0 32.0 33.0 ... 41.0 42.0 43.0 44.0
  * level         (level) float64 100.0 200.0 300.0 400.0
  * time          (time) datetime64[ns] 2021-01-01 ... 2021-01-01T04:00:00
    air_pressure  (level) float64 1e+04 2e+04 3e+04 4e+04
    altitude      (level) float64 1.618e+04 1.178e+04 9.164e+03 7.185e+03
```

5.1.2 MetDataset

Thin wrapper around xr.Dataset

[6]:
```
ds = xr.Dataset(
    {
        "random": (["longitude", "latitude", "level", "time"], rng.random((20, 15, 4,␣
↪5))),
        "ones": (["longitude", "latitude", "level", "time"], np.ones((20, 15, 4, 5))),
    },
    coords={
        "longitude": np.arange(-100, -80, 1.0),
        "latitude": np.arange(30, 45, 1.0),
        "level": np.arange(100, 500, 100),
        "time": [datetime(2021, 1, 1, h) for h in range(5)],
    },
)
```

[7]:
```
ds
```

[7]:
```
<xarray.Dataset>
Dimensions:    (longitude: 20, latitude: 15, level: 4, time: 5)
Coordinates:
  * longitude  (longitude) float64 -100.0 -99.0 -98.0 ... -83.0 -82.0 -81.0
  * latitude   (latitude) float64 30.0 31.0 32.0 33.0 ... 41.0 42.0 43.0 44.0
  * level      (level) int64 100 200 300 400
  * time       (time) datetime64[ns] 2021-01-01 ... 2021-01-01T04:00:00
Data variables:
    random     (longitude, latitude, level, time) float64 0.08219 ... 0.3966
    ones       (longitude, latitude, level, time) float64 1.0 1.0 ... 1.0 1.0
```

[8]:
```
met = MetDataset(ds)
met
```

[8]:
```
MetDataset with data:

<xarray.Dataset>
Dimensions:       (longitude: 20, latitude: 15, level: 4, time: 5)
Coordinates:
  * longitude     (longitude) float64 -100.0 -99.0 -98.0 ... -83.0 -82.0 -81.0
  * latitude      (latitude) float64 30.0 31.0 32.0 33.0 ... 41.0 42.0 43.0 44.0
  * level         (level) float64 100.0 200.0 300.0 400.0
  * time          (time) datetime64[ns] 2021-01-01 ... 2021-01-01T04:00:00
    air_pressure  (level) float64 1e+04 2e+04 3e+04 4e+04
    altitude      (level) float64 1.618e+04 1.178e+04 9.164e+03 7.185e+03
Data variables:
    random        (longitude, latitude, level, time) float64 0.08219 ... 0.3966
    ones          (longitude, latitude, level, time) float64 1.0 1.0 ... 1.0 1.0
```

[9]:
```
# broadcast coords to the DataArray
met.broadcast_coords("air_pressure")
```

[9]:
```
<xarray.DataArray 'air_pressure' (longitude: 20, latitude: 15, level: 4, time: 5)>
array([[[[10000., 10000., 10000., 10000., 10000.],
```

(continues on next page)

(continued from previous page)

```
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        ...,

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
...
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        ...,

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]],

        [[10000., 10000., 10000., 10000., 10000.],
         [20000., 20000., 20000., 20000., 20000.],
         [30000., 30000., 30000., 30000., 30000.],
         [40000., 40000., 40000., 40000., 40000.]]])
Coordinates:
  * longitude     (longitude) float64 -100.0 -99.0 -98.0 ... -83.0 -82.0 -81.0
  * latitude      (latitude) float64 30.0 31.0 32.0 33.0 ... 41.0 42.0 43.0 44.0
  * level         (level) float64 100.0 200.0 300.0 400.0
  * time          (time) datetime64[ns] 2021-01-01 ... 2021-01-01T04:00:00
    air_pressure  (level) float64 1e+04 2e+04 3e+04 4e+04
    altitude      (level) float64 1.618e+04 1.178e+04 9.164e+03 7.185e+03
```

5.1.3 MetVariable

Dataclass to contain metadata for meteorology variables.

[10]:
```
mv = MetVariable(short_name="mv", standard_name="met_variable", long_name="Custom Met␣
↪Variable")
mv
```

[10]:
```
MetVariable(short_name='mv', standard_name='met_variable', long_name='Custom Met Variable
↪', level_type=None, ecmwf_id=None, grib1_id=None, grib2_id=None, units=None, amip=None,
↪ description=None)
```

[11]:
```
# attrs shorthand
mv.attrs
```

[11]:
```
{'short_name': 'mv',
 'standard_name': 'met_variable',
 'long_name': 'Custom Met Variable'}
```

Built-in Variables

`pycontrails` contains many standard met variables in the `pycontrails.core.met_var` module.

The `standard_name` attribute is used by default as string references through the repository.

[12]:
```
from pycontrails.core.met_var import (
    AirTemperature,
    SpecificHumidity,
)
```

[13]:
```
AirTemperature.attrs
```

[13]:
```
{'short_name': 't',
 'standard_name': 'air_temperature',
 'long_name': 'Air Temperature',
 'units': 'K'}
```

[14]:
```
SpecificHumidity.attrs
```

[14]:
```
{'short_name': 'q',
 'standard_name': 'specific_humidity',
 'long_name': 'Specific Humidity',
 'units': 'kg kg**-1'}
```

5.2 Load ECMWF data

Requires `[ecmwf]` optional dependencies:

```
$ pip install pycontrails[ecmwf]
```

Support provided for:

- ERA5 via the Copernicus Data Store (CDS) using cdsapi or user provided files

- HRES and ENS via MARS using ecmwf-api-client or user provided files.

For both ERA5 and HRES, we provide interfaces for accessing "pressure-level data" (fields pre-interpolated to a fixed set of pressure levels) or "model-level data" (fields retrieved on the native vertical grid and *interpolated after retrieval to an arbitrary set of pressure levels*). We recommend using model-level data when possible, as the resolution of pressure-level data is coarse relative to the vertical scale of ice-supersaturated regions.

Note that tools for accessing ECMWF data are not thoroughly tested in CI because they are vulnerable to upstream failures in external APIs. If you think you have found a problem please open an issue!

5.2.1 ERA5

Access

- Requires account with Copernicus Data Portal
- Provide `url` and `key` credentials on input, or refer to the CDS API Documentation for how to create `~/.cdsapirc` file to configure access.

Reference

- ERA 5 Data Documentation
- ERA5 Parameter Listing

ERA5 Pressure Levels

- ERA5 Pressure Level Parameters

[1]:
```
from pycontrails.datalib.ecmwf import ERA5
```

[2]:
```
# get a single time
era5 = ERA5(
    time="2022-03-01 00:00:00",
    variables=["t", "q", "u", "v", "w", "ciwc", "z", "cc"],  # supports CF name or short␣
↪names
    pressure_levels=[200, 250, 300],
    # url="https://cds.climate.copernicus.eu/api/v2",
    # key="<key>"
)
era5
```

[2]:
```
ERA5
        Timesteps: ['2022-03-01 00']
        Variables: ['t', 'q', 'u', 'v', 'w', 'ciwc', 'z', 'cc']
        Pressure levels: [200, 250, 300]
        Grid: 0.25
        Dataset: reanalysis-era5-pressure-levels
        Product type: reanalysis
```

[3]:
```
# get a range of time and all available pressure levels between 450 and 125 hPa
era5 = ERA5(
    time=("2022-03-01 00:00:00", "2022-03-01 03:00:00"),
```

(continues on next page)

(continued from previous page)

```
    variables=[
        "air_temperature",
        "q",
        "u",
        "v",
        "w",
        "ciwc",
        "z",
        "cc",
    ],  # supports CF name or short names
    pressure_levels=[300, 250, 200],
    # url="https://cds.climate.copernicus.eu/api/v2",
    # key="<key>"
)
era5
```

[3]:
```
ERA5
        Timesteps: ['2022-03-01 00', '2022-03-01 01', '2022-03-01 02', '2022-03-01 03']
        Variables: ['t', 'q', 'u', 'v', 'w', 'ciwc', 'z', 'cc']
        Pressure levels: [200, 250, 300]
        Grid: 0.25
        Dataset: reanalysis-era5-pressure-levels
        Product type: reanalysis
```

[4]:
```
# this triggers a download from CDS if file isn't in cache store
met_pl = era5.open_metdataset()
met_pl
```

[4]:
```
MetDataset with data:

<xarray.Dataset> Size: 797MB
Dimensions:                                (longitude: 1440, latitude: 721,
                                            level: 3, time: 4)
Coordinates:
  * latitude                               (latitude) float64 6kB -90.0 ... 90.0
  * level                                  (level) float64 24B 200.0 250.0 300.0
  * time                                   (time) datetime64[ns] 32B 2022-03-01...
  * longitude                              (longitude) float64 12kB -180.0 ... ...
    air_pressure                           (level) float64 24B 2e+04 2.5e+04 3e+04
    altitude                               (level) float64 24B 1.178e+04 ... 9...
Data variables:
    air_temperature                        (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
    specific_humidity                      (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
    eastward_wind                          (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
    northward_wind                         (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
    lagrangian_tendency_of_air_pressure    (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
    specific_cloud_ice_water_content       (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    geopotential                          (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
    fraction_of_cloud_cover               (longitude, latitude, level, time) float64␣
↪100MB dask.array<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
Attributes:
    Conventions:          CF-1.6
    history:              2024-04-17 23:08:13 GMT by grib_to_netcdf-2.25.1: /...
    pycontrails_version:  0.50.3.dev18
    provider:             ECMWF
    dataset:              ERA5
    product:              reanalysis
```

ERA5 Single Level

- ERA5 Single Level Parameters (invariant)
- ERA5 Single Level Parameters (instantaenous)
- ERA5 Single Level Parameters (accumulations)

[5]:
```
era5 = ERA5(
    time=("2022-03-01 00:00:00", "2022-03-01 03:00:00"),
    variables=["tsr", "ttr"],
    # url="https://cds.climate.copernicus.eu/api/v2",
    # key="<key>"
)
era5
```

[5]:
```
ERA5
        Timesteps: ['2022-03-01 00', '2022-03-01 01', '2022-03-01 02', '2022-03-01 03']
        Variables: ['tsr', 'ttr']
        Pressure levels: [-1]
        Grid: 0.25
        Dataset: reanalysis-era5-single-levels
        Product type: reanalysis
```

[6]:
```
met = era5.open_metdataset()
met
```

[6]:
```
MetDataset with data:

<xarray.Dataset> Size: 66MB
Dimensions:                    (level: 1, time: 4, latitude: 721,
                                longitude: 1440)
Coordinates:
  * level                      (level) float64 8B -1.0
  * latitude                   (latitude) float64 6kB -90.0 -89.75 ... 90.0
  * time                       (time) datetime64[ns] 32B 2022-03-01 ... 2022-...
  * longitude                  (longitude) float64 12kB -180.0 -179.8 ... 179.8
Data variables:
    top_net_solar_radiation    (longitude, latitude, level, time) float64 33MB dask.array
↪<chunksize=(1440, 721, 1, 1), meta=np.ndarray>
    top_net_thermal_radiation  (longitude, latitude, level, time) float64 33MB dask.array
```

(continues on next page)

(continued from previous page)

```
↪<chunksize=(1440, 721, 1, 1), meta=np.ndarray>
Attributes:
    Conventions:          CF-1.6
    history:              2024-04-24 21:36:27 GMT by grib_to_netcdf-2.28.1: /...
    pycontrails_version:  0.50.3.dev18
    provider:             ECMWF
    dataset:              ERA5
    product:              reanalysis
```

ERA5 Model Levels

- ERA5 Model Level Parameters
- ERA5 Catalog (nominal reanalysis)
- ERA5 Catalog (ensemble members)

[7]:
```
from pycontrails.datalib.ecmwf import ERA5ModelLevel
```

Model-level data has much higher vertical resolution than pressure-level data, so we download at coarser horizontal resolution to decrease data volume.

If target pressure levels are not explicitly provided, `ERA5ModelLevel` defaults to pressure levels near model levels between 20,000 and 50,000 feet. These levels are determined by reading a static file based on https://confluence.ecmwf.int/display/UDOC/L137+model+level+definitions.

[8]:
```
era5 = ERA5ModelLevel(
    time=("2022-03-01 00:00:00", "2022-03-01 03:00:00"),
    variables=["t", "q", "u", "v", "w", "ciwc"],
    grid=1.0,
)
era5
```

[8]:
```
ERA5ModelLevel
        Timesteps: ['2022-03-01 00', '2022-03-01 01', '2022-03-01 02', '2022-03-01 03']
        Variables: ['t', 'q', 'u', 'v', 'w', 'ciwc']
        Pressure levels: [121, 127, 134, 141, 148, 155, 163, 171, 180, 188, 197, 207,␣
↪217, 227, 237, 248, 260, 272, 284, 297, 310, 323, 337, 352, 367, 383, 399, 416, 433,␣
↪451]
        Grid: 1.0
        Dataset: reanalysis-era5-complete
        Product type: reanalysis
```

[9]:
```
met_ml = era5.open_metdataset()
met_ml
```

[9]:
```
MetDataset with data:

<xarray.Dataset> Size: 188MB
Dimensions:                                (longitude: 360, latitude: 181,
                                            level: 30, time: 4)

Coordinates:
  * time                                   (time) datetime64[ns] 32B 2022-03-01...
```

(continues on next page)

(continued from previous page)

```
    step                                timedelta64[ns] 8B 00:00:00
  * level                               (level) float64 240B 121.0 ... 451.0
  * latitude                            (latitude) float64 1kB -90.0 ... 90.0
    valid_time                          (time) datetime64[ns] 32B 2022-03-01...
  * longitude                           (longitude) float64 3kB -180.0 ... 1...
    air_pressure                        (level) float32 120B 1.21e+04 ... 4...
    altitude                            (level) float32 120B 1.497e+04 ... 6...
Data variables:
    air_temperature                     (longitude, latitude, level, time) float32 31MB␣
↪dask.array<chunksize=(360, 181, 30, 1), meta=np.ndarray>
    specific_humidity                   (longitude, latitude, level, time) float32 31MB␣
↪dask.array<chunksize=(360, 181, 30, 1), meta=np.ndarray>
    eastward_wind                       (longitude, latitude, level, time) float32 31MB␣
↪dask.array<chunksize=(360, 181, 30, 1), meta=np.ndarray>
    northward_wind                      (longitude, latitude, level, time) float32 31MB␣
↪dask.array<chunksize=(360, 181, 30, 1), meta=np.ndarray>
    lagrangian_tendency_of_air_pressure  (longitude, latitude, level, time) float32 31MB␣
↪dask.array<chunksize=(360, 181, 30, 1), meta=np.ndarray>
    specific_cloud_ice_water_content    (longitude, latitude, level, time) float32 31MB␣
↪dask.array<chunksize=(360, 181, 30, 1), meta=np.ndarray>
Attributes:
    GRIB_edition:            2
    GRIB_centre:             ecmf
    GRIB_centreDescription:  European Centre for Medium-Range Weather Forecasts
    GRIB_subCentre:          0
    Conventions:             CF-1.7
    institution:             European Centre for Medium-Range Weather Forecasts
    history:                 2024-04-24T23:56 GRIB to CDM+CF via cfgrib-0.9.1...
    pycontrails_version:     0.50.3.dev24
    provider:                ECMWF
    dataset:                 ERA5
    product:                 reanalysis
```

5.2.2 HRES

Access

Users within ECMWF Member and Co-operating States may contact their Computing Representative to obtain access to MARS. All other users may request a username and password and then get an api key.

Provide `url`, `key`, and `email` credentials on input, or see ECMWF API Client documentation to configure local `~/.ecmwfapirc` file:

```
{
    "url": "https://api.ecmwf.int/v1",
    "email": "<email>",
    "key": "<key>"
}
```

Reference

- HRES High resolution forecast
- ENS Ensemble forecast

HRES Pressure Levels

[10]:
```
from datetime import datetime

from pycontrails.datalib.ecmwf import HRES
```

[11]:
```
# NOTE / TODO: Including the "ciwc" variable here, the HRES request
# fails with on historic data. However, the request seems to go through
# when the time field is recent (within the last 48 hours?)
time = datetime(2022, 3, 26, 0), datetime(2022, 3, 26, 2)
hres = HRES(
    time=time,
    variables=["t", "q", "u", "v", "w", "z"],
    pressure_levels=[300, 250, 200],
    grid=1,
    # url="https://api.ecmwf.int/v1",
    # key="<key>"
    # email="<email>"
)
hres
```

[11]:
```
HRES
        Timesteps: ['2022-03-26 00', '2022-03-26 01', '2022-03-26 02']
        Variables: ['t', 'q', 'u', 'v', 'w', 'z']
        Pressure levels: [200, 250, 300]
        Grid: 1
        Forecast time: 2022-03-26 00:00:00
        Steps: [0, 1, 2]
```

[12]:
```
# convience method to see the underlying MARS request
print(hres.generate_mars_request())
```

```
retrieve,
        class=od,
        stream=oper,
        expver=1,
        date=20220326,
        time=00,
        type=fc,
        param=t/q/u/v/w/z,
        step=0/1/2,
        grid=1/1,
        levtype=pl,
        levelist=200/250/300
```

[13]:
```
# this triggers a download if file isn't in cache store
met_pl = hres.open_metdataset()
met_pl
```

[13]:
```
MetDataset with data:

<xarray.Dataset> Size: 14MB
Dimensions:                                (longitude: 360, latitude: 181,
                                            level: 3, time: 3)
Coordinates:
    forecast_time                          datetime64[ns] 8B 2022-03-26
  * level                                  (level) float64 24B 200.0 250.0 300.0
  * latitude                               (latitude) float64 1kB -90.0 ... 90.0
  * time                                   (time) datetime64[ns] 24B 2022-03-26...
  * longitude                              (longitude) float64 3kB -180.0 ... 1...
    air_pressure                           (level) float32 12B 2e+04 2.5e+04 3e+04
    altitude                               (level) float32 12B 1.178e+04 ... 9...
Data variables:
    air_temperature                        (longitude, latitude, level, time) float32 2MB␣
↪dask.array<chunksize=(360, 181, 3, 1), meta=np.ndarray>
    specific_humidity                      (longitude, latitude, level, time) float32 2MB␣
↪dask.array<chunksize=(360, 181, 3, 1), meta=np.ndarray>
    eastward_wind                          (longitude, latitude, level, time) float32 2MB␣
↪dask.array<chunksize=(360, 181, 3, 1), meta=np.ndarray>
    northward_wind                         (longitude, latitude, level, time) float32 2MB␣
↪dask.array<chunksize=(360, 181, 3, 1), meta=np.ndarray>
    lagrangian_tendency_of_air_pressure  (longitude, latitude, level, time) float32 2MB␣
↪dask.array<chunksize=(360, 181, 3, 1), meta=np.ndarray>
    geopotential                           (longitude, latitude, level, time) float32 2MB␣
↪dask.array<chunksize=(360, 181, 3, 1), meta=np.ndarray>
Attributes:
    GRIB_edition:            1
    GRIB_centre:             ecmf
    GRIB_centreDescription:  European Centre for Medium-Range Weather Forecasts
    GRIB_subCentre:          0
    Conventions:             CF-1.7
    institution:             European Centre for Medium-Range Weather Forecasts
    history:                 2024-04-24T23:57 GRIB to CDM+CF via cfgrib-0.9.1...
    pycontrails_version:     0.50.3.dev24
    provider:                ECMWF
    dataset:                 HRES
    product:                 forecast
    radiation_accumulated:   True
```

HRES Single Level

Note that accumulated parameters (i.e. `top_net_thermal_radiation`, `toa_incident_solar_radiation` and other radiation parameters) are accumulated from the *start of the forecast*

[14]:
```
hres = HRES(
    time=time,
    variables=["tsr", "ttr"],
    grid=1,
    # url="https://api.ecmwf.int/v1",
    # key="<key>"
    # email="<email>"
)
```

[15]:
```
met = hres.open_metdataset()
met
```

[15]:
```
MetDataset with data:

<xarray.Dataset> Size: 2MB
Dimensions:                    (level: 1, time: 3, latitude: 181, longitude: 360)
Coordinates:
  * level                      (level) float64 8B -1.0
    forecast_time              datetime64[ns] 8B 2022-03-26
    surface                    float64 8B 0.0
  * latitude                   (latitude) float64 1kB -90.0 -89.0 ... 89.0 90.0
  * time                       (time) datetime64[ns] 24B 2022-03-26 ... 2022-...
  * longitude                  (longitude) float64 3kB -180.0 -179.0 ... 179.0
Data variables:
    top_net_solar_radiation    (longitude, latitude, level, time) float32 782kB dask.
↪array<chunksize=(360, 181, 1, 1), meta=np.ndarray>
    top_net_thermal_radiation  (longitude, latitude, level, time) float32 782kB dask.
↪array<chunksize=(360, 181, 1, 1), meta=np.ndarray>
Attributes:
    GRIB_edition:            1
    GRIB_centre:             ecmf
    GRIB_centreDescription:  European Centre for Medium-Range Weather Forecasts
    GRIB_subCentre:          0
    Conventions:             CF-1.7
    institution:             European Centre for Medium-Range Weather Forecasts
    history:                 2024-04-24T23:57 GRIB to CDM+CF via cfgrib-0.9.1...
    pycontrails_version:     0.50.3.dev24
    provider:                ECMWF
    dataset:                 HRES
    product:                 forecast
    radiation_accumulated:   True
```

Specify forecast by runtime

Select data from specific forecast run by `forecast_time`

[16]:
```
hres = HRES(
    time=("2022-03-26 01:00:00", "2022-03-26 02:00:00"),
    variables=["t", "q"],
    pressure_levels=[300, 250, 200],
    forecast_time="2022-03-25 12:00:00",
    # url="https://api.ecmwf.int/v1",
    # key="<key>"
    # email="<email>"
)
hres
```

[16]:
```
HRES
        Timesteps: ['2022-03-26 01', '2022-03-26 02']
        Variables: ['t', 'q']
        Pressure levels: [200, 250, 300]
        Grid: 0.25
        Forecast time: 2022-03-25 12:00:00
        Steps: [13, 14]
```

HRES Model Levels

- Operational Archive Catalog

[17]:
```
from pycontrails.datalib.ecmwf import HRESModelLevel
```

Similar to the model-level ERA5 demo, we download at a relatively coarse horizontal resolution to decrease data volume.

[18]:
```
hres = HRESModelLevel(
    time=("2022-03-26 01:00:00", "2022-03-26 02:00:00"),
    variables=["t", "q"],
    forecast_time="2022-03-25 12:00:00",
    grid=1.0,
)
hres
```

[18]:
```
HRESModelLevel
        Timesteps: ['2022-03-26 01', '2022-03-26 02']
        Variables: ['t', 'q']
        Pressure levels: [121, 127, 134, 141, 148, 155, 163, 171, 180, 188, 197, 207,␣
↪217, 227, 237, 248, 260, 272, 284, 297, 310, 323, 337, 352, 367, 383, 399, 416, 433,␣
↪451]
        Grid: 1.0
        Forecast time: 2022-03-25 12:00:00
        Steps: [13, 14]
```

[19]:
```
met_ml = hres.open_metdataset()
met_ml
```

[19]:
```
MetDataset with data:

<xarray.Dataset> Size: 31MB
Dimensions:               (longitude: 360, latitude: 181, level: 30, time: 2)
Coordinates:
    initialization_time  datetime64[ns] 8B 2022-03-25T12:00:00
  * time                 (time) datetime64[ns] 16B 2022-03-26T01:00:00 2022-0...
  * level                (level) float64 240B 121.0 127.0 134.0 ... 433.0 451.0
  * latitude             (latitude) float64 1kB -90.0 -89.0 -88.0 ... 89.0 90.0
    valid_time           (time) datetime64[ns] 16B 2022-03-26T01:00:00 2022-0...
  * longitude            (longitude) float64 3kB -180.0 -179.0 ... 178.0 179.0
    air_pressure         (level) float32 120B 1.21e+04 1.27e+04 ... 4.51e+04
    altitude             (level) float32 120B 1.497e+04 1.466e+04 ... 6.328e+03
Data variables:
    air_temperature      (longitude, latitude, level, time) float32 16MB dask.array
↪<chunksize=(360, 181, 30, 1), meta=np.ndarray>
    specific_humidity    (longitude, latitude, level, time) float32 16MB dask.array
↪<chunksize=(360, 181, 30, 1), meta=np.ndarray>
Attributes:
    GRIB_edition:            2
    GRIB_centre:             ecmf
    GRIB_centreDescription:  European Centre for Medium-Range Weather Forecasts
    GRIB_subCentre:          0
    Conventions:             CF-1.7
    institution:             European Centre for Medium-Range Weather Forecasts
    history:                 2024-04-24T23:59 GRIB to CDM+CF via cfgrib-0.9.1...
    pycontrails_version:     0.50.3.dev24
    provider:                ECMWF
    dataset:                 HRES
    product:                 forecast
    radiation_accumulated:   True
```

5.2.3 IFS

In development

Integrated Forecasting System from ECMWF

- IFS Documentation

Access

IFS files must be downloaded to a local directory before accessing.

Reference

- IFS Confluence

[20]:
```
from pycontrails.datalib.ecmwf import IFS
```

[21]:
```
ifs = IFS(
    time=("2021-10-02 00:00:00", "2021-10-02 14:00:00"),
    variables=["air_temperature"],
    forecast_path="ifs",
    forecast_date="2021-10-01",
)
```

5.2.4 ECMWF Variables

`ECMWF_VARIABLES` attribute lists the supported parameters from the ECMWF Pameter DB as a `list[MetVariable]`

[22]:
```
from pycontrails.datalib.ecmwf import ECMWF_VARIABLES
```

[23]:
```
[met_var.standard_name for met_var in ECMWF_VARIABLES]
```

[23]:
```
['air_temperature',
 'specific_humidity',
 'geopotential',
 'eastward_wind',
 'northward_wind',
 'lagrangian_tendency_of_air_pressure',
 'relative_humidity',
 'atmosphere_upward_relative_vorticity',
 'fraction_of_cloud_cover',
 'specific_cloud_ice_water_content',
 'specific_cloud_liquid_water_content',
 'potential_vorticity',
 'surface_air_pressure',
 'toa_incident_solar_radiation',
 'top_net_solar_radiation',
 'top_net_thermal_radiation',
 'total_cloud_cover',
 'surface_solar_downward_radiation']
```

[24]:
```
from pycontrails.datalib.ecmwf import TopNetSolarRadiation
```

[25]:
```
# ECMWF variables contain a link to the param-db entry
TopNetSolarRadiation.ecmwf_link
```

[25]:
```
'https://apps.ecmwf.int/codes/grib/param-db?id=178'
```

5.2.5 Cache Data Files to GCP

Requires `[gcp]` optional dependencies:

```
$ pip install pycontrails[gcp]
```

By default, data files are cached to the local disk in the users `Caches` directory.

To cache files to a remote Google Cloud Storage bucket, use the `GCPCacheStore`

ERA5

[26]:
```
from pycontrails import GCPCacheStore
```

[27]:
```
variables = ["air_temperature", "relative_humidity"]

gcp = GCPCacheStore(bucket="contrails-301217-unit-test", cache_dir="test/era5", read_
↪only=False)

era5 = ERA5(
    time=(datetime(2019, 1, 1, 0), datetime(2019, 1, 1, 2)),
    variables=variables,
    pressure_levels=[300, 250, 150],
    cachestore=gcp,
    # url="https://cds.climate.copernicus.eu/api/v2",
    # key="<key>"
)
```

[28]:
```
# download data to cache - uncomment to run
# met = era5.open_metdataset()
```

5.3 Load GFS data

Requires `[gfs]` optional dependencies:

```
$ pip install pycontrails[gfs]
```

5.3.1 References

- NOAA GFS - AWS Open Data Registry
- NCEI GFS Documentation
- NOAA GFS Documentation
- Parameter definitions

See API Reference for usage.

[1]:
```
import numpy as np

from pycontrails.datalib.gfs import GFSForecast
```

[2]:
```
# get a single time
gfs = GFSForecast(
    time="2022-03-01 01:00:00",
    variables=["t", "q"],  # Supports CF name or short names
    pressure_levels=[200, 250, 300],
    show_progress=True,  # Shows download progress from AWS
)
gfs
```

[2]:
```
GFSForecast
        Timesteps: ['2022-03-01 01']
        Variables: ['t', 'q']
        Pressure levels: [200, 250, 300]
        Grid: 0.25
        Forecast time: 2022-03-01 00:00:00
```

[3]:
```
# get a range of time
gfs = GFSForecast(
    time=("2022-03-01 00:00:00", "2022-03-01 02:00:00"),
    variables=[
        "air_temperature",
        "q",
    ],  # supports CF name or short names
    pressure_levels=[200, 250, 300],
    show_progress=True,
)
gfs
```

[3]:
```
GFSForecast
        Timesteps: ['2022-03-01 00', '2022-03-01 01', '2022-03-01 02']
        Variables: ['t', 'q']
        Pressure levels: [200, 250, 300]
        Grid: 0.25
        Forecast time: 2022-03-01 00:00:00
```

[4]:
```
# this triggers a download from AWS if file isn't in cache store
met = gfs.open_metdataset()
met
```

[4]:
```
MetDataset with data:

<xarray.Dataset>
Dimensions:            (longitude: 1440, latitude: 721, level: 3, time: 3)
Coordinates:
  * level              (level) float64 200.0 250.0 300.0
  * latitude           (latitude) float64 -90.0 -89.75 -89.5 ... 89.5 89.75 90.0
  * longitude          (longitude) float64 -180.0 -179.8 -179.5 ... 179.5 179.8
    forecast_time      datetime64[ns] 2022-03-01
  * time               (time) datetime64[ns] 2022-03-01 ... 2022-03-01T02:00:00
    air_pressure       (level) float32 2e+04 2.5e+04 3e+04
    altitude           (level) float32 1.178e+04 1.036e+04 9.164e+03
Data variables:
    air_temperature    (longitude, latitude, level, time) float32 dask.array
↪<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
```

(continues on next page)

(continued from previous page)

```
    specific_humidity  (longitude, latitude, level, time) float32 dask.array
↪<chunksize=(1440, 721, 3, 1), meta=np.ndarray>
Attributes:
    GRIB_edition:            2
    GRIB_centre:             kwbc
    GRIB_centreDescription:  US National Weather Service - NCEP
    GRIB_subCentre:          0
    Conventions:             CF-1.7
    institution:             US National Weather Service - NCEP
    history:                 2023-07-12T15:51 GRIB to CDM+CF via cfgrib-0.9.1...
    pycontrails_version:     0.49.3.dev50
    provider:                NCEP
    dataset:                 GFS
    product:                 forecast
```

[5]:
```
# get steps for a specific forecast time
time_bounds = ("2022-11-18 16:00:00", "2022-11-18 18:00:00")
gfs = GFSForecast(
    time_bounds,
    variables=["t", "q"],
    pressure_levels=[200, 250, 300, 350],
    forecast_time=np.datetime64("2022-11-17 00:06:00"),
    show_progress=True,
)
gfs
```

[5]:
```
GFSForecast
        Timesteps: ['2022-11-18 16', '2022-11-18 17', '2022-11-18 18']
        Variables: ['t', 'q']
        Pressure levels: [200, 250, 300, 350]
        Grid: 0.25
        Forecast time: 2022-11-17 00:00:00
```

Run CoCiP with GFS

See the CoCiP Notebook for more details on running the CoCiP model.

[6]:
```
from pycontrails.models.cocip import Cocip
from pycontrails.physics import units
from pycontrails import Flight

import numpy as np
import pandas as pd
```

[7]:
```
time_bounds = ("2022-03-01 00:00:00", "2022-03-01 23:00:00")
pressure_levels = [300, 250, 200]

gfs_met = GFSForecast(
    time_bounds, variables=Cocip.met_variables, pressure_levels=pressure_levels, show_
↪progress=True
)
```

(continues on next page)

(continued from previous page)

```
gfs_rad = GFSForecast(time_bounds, variables=Cocip.rad_variables, show_progress=True)
```

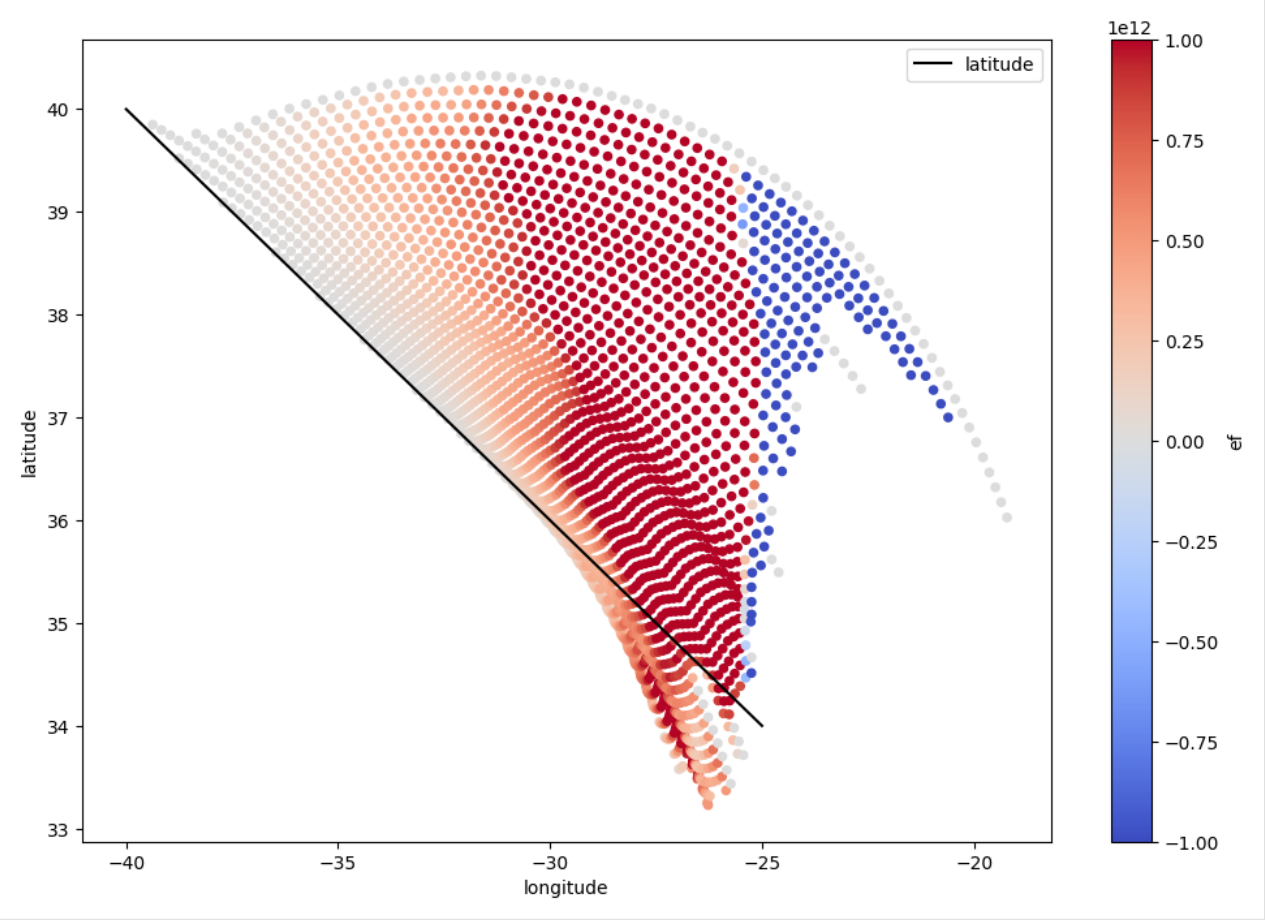
[8]:
```
# download data from AWS (or open from cache)
met = gfs_met.open_metdataset()
rad = gfs_rad.open_metdataset()
```

[9]:
```
# demo synthetic flight
flight_attrs = {
    "flight_id": "test",
    # set constants along flight path
    "true_airspeed": 226.099920796651,  # true airspeed, m/s
    "thrust": 0.22,  # thrust_setting
    "nvpm_ei_n": 1.897462e15,  # non-volatile emissions index
    "aircraft_type": "E190",
    "wingspan": 48,  # m
    "n_engine": 2,
}

# Example flight
df = pd.DataFrame()
df["longitude"] = np.linspace(-25, -40, 100)
df["latitude"] = np.linspace(34, 40, 100)
df["altitude"] = np.linspace(10900, 10900, 100)
df["engine_efficiency"] = np.linspace(0.34, 0.35, 100)
df["fuel_flow"] = np.linspace(2.1, 2.4, 100)  # kg/s
df["aircraft_mass"] = np.linspace(154445, 154345, 100)  # kg
df["time"] = pd.date_range("2022-03-01T00:15:00", "2022-03-01T02:30:00", periods=100)

flight = Flight(df, attrs=flight_attrs)
```

[10]:
```
# set up CoCiP model with GFS meteorology and radiation
params = {"dt_integration": np.timedelta64(10, "m")}
cocip = Cocip(met=met, rad=rad, params=params)
```

[11]:
```
# evaluate flight in CoCiP model
output_flight = cocip.eval(source=flight)
```

[12]:
```
# review the energy forcing from flight segments
ax = cocip.source.dataframe.plot("longitude", "latitude", color="k", figsize=(12, 8))
cocip.contrail.plot.scatter(
    "longitude", "latitude", c="ef", cmap="coolwarm", vmin=-1e12, vmax=1e12, ax=ax
);
```



5.4 ARCO ERA5

This notebook demonstrates how to load ARCO ERA5 data from Google Cloud Storage through the pycontrails `ARCOERA5` interface.

The `ARCOERA5` interface requires the Metview Python API, which itself requires a working installation of the Metview binary. Installing the Metview binary is beyond the scope of this notebook, but you can follow the ECMWF instructions. We have had success with the `conda` installation method.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import xarray as xr

from pycontrails import Flight, MetDataset
from pycontrails.datalib.ecmwf import ARCOERA5
from pycontrails.models.cocip import Cocip
from pycontrails.models.humidity_scaling import ConstantHumidityScaling
from pycontrails.models.issr import ISSR
from pycontrails.models.ps_model import PSFlight
from pycontrails.physics import units
```

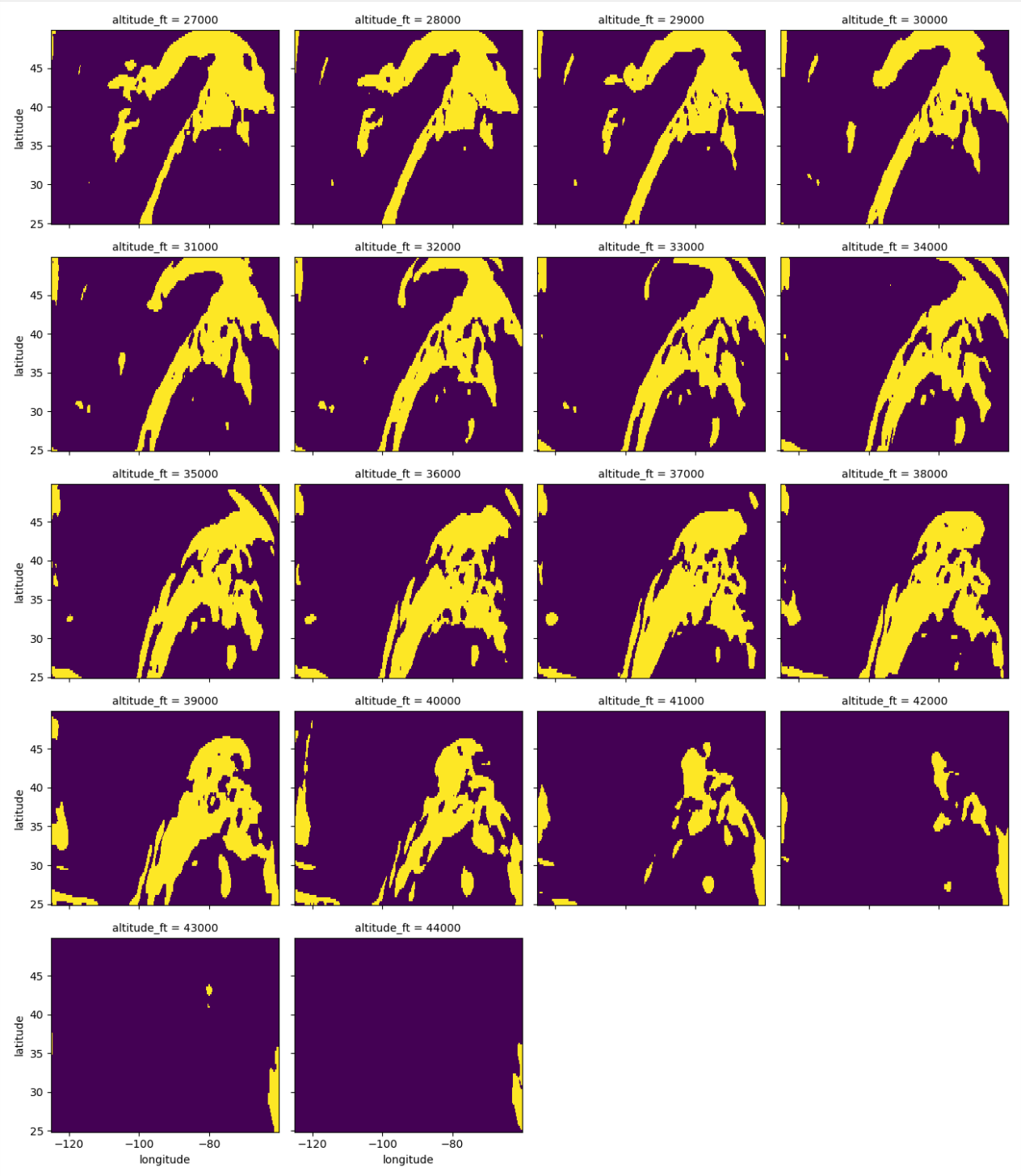
Load 13 hours of pressure level and single level data from the ARCO ERA5 dataset. Select all the variables needed for the Cocip model.

[2]:
```
time = ("2019-01-01T00", "2019-01-01T12")

era5_pl = ARCOERA5(
    time=time,
    variables=(*Cocip.met_variables, *Cocip.optional_met_variables),
    n_jobs=4,  # run conversion in parallel among 4 separate processes
)
met = era5_pl.open_metdataset()


era5_sl = ARCOERA5(
    time=time,
    variables=Cocip.rad_variables,
    pressure_levels=-1,
)
rad = era5_sl.open_metdataset()
```

5.4.1 ISSRs

Visualize ISSRs over common cruise altitudes over the continental US.

[3]:
```
issr = ISSR(met=met, humidity_scaling=ConstantHumidityScaling(rhi_adj=0.95))
source = MetDataset(
    xr.Dataset(
        coords={
            "time": ["2019-01-01T00"],
            "longitude": np.arange(-125, -60, 0.25),
            "latitude": np.arange(25, 50, 0.25),
            "level": units.ft_to_pl(np.arange(27000, 45000, 1000)),
        }
    )
)
da = issr.eval(source).data["issr"]

# Use altitude_ft as the vertical coordinate for plotting
da["altitude_ft"] = units.pl_to_ft(da["level"]).round().astype(int)
da = da.swap_dims(level="altitude_ft")
da = da.sel(altitude_ft=da["altitude_ft"].values[::-1])
da = da.squeeze()

da.plot(x="longitude", y="latitude", col="altitude_ft", col_wrap=4, add_colorbar=False);
```


altitude_ft = 27000
altitude_ft = 28000
altitude_ft = 29000
altitude_ft = 30000
altitude_ft = 31000
altitude_ft = 32000
altitude_ft = 33000
altitude_ft = 34000
altitude_ft = 35000
altitude_ft = 36000
altitude_ft = 37000
altitude_ft = 38000
altitude_ft = 39000
altitude_ft = 40000
altitude_ft = 41000
altitude_ft = 42000
altitude_ft = 43000
altitude_ft = 44000
latitude
longitude
25
30
35
40
45
−120
−100
−80

5.4.2 CoCiP

Run CoCiP on a collection of synthetic flights at different cruise altitudes over the continental US.

The flights constructed below are in no way realistic. They are simply used to showcase running CoCiP with model level met data.

[4]:
```
jfk = 40.64, -73.78
lax = 33.94, -118.40

fl_list = [
    Flight(
        longitude=[lax[1], jfk[1]],
        latitude=[lax[0], jfk[0]],
        altitude_ft=[altitude_ft, altitude_ft],
        time=[np.datetime64("2019-01-01T00"), np.datetime64("2019-01-01T04:30")],
        aircraft_type="B738",
        flight_id=f"cruise {altitude_ft} ft",
        origin="LAX",
        destination="JFK",
    ).resample_and_fill()
    for altitude_ft in range(27000, 45000, 1000)
]
```

[5]:
```
cocip = Cocip(
    met=met,
    rad=rad,
    dt_integration="1min",
    max_age="8h",
    aircraft_performance=PSFlight(),
    humidity_scaling=ConstantHumidityScaling(rhi_adj=0.95),
)
fl_list = cocip.eval(fl_list)
```

5.4.3 Visualize CoCiP results

Visualize the results of CoCiP on the synthetic flights.

[6]:
```
summary = {fl.attrs["flight_id"]: fl["ef"].sum() / 1e12 for fl in fl_list}
pd.Series(summary).to_frame("Energy Forcing [TJ]").astype(int)
```

[6]:

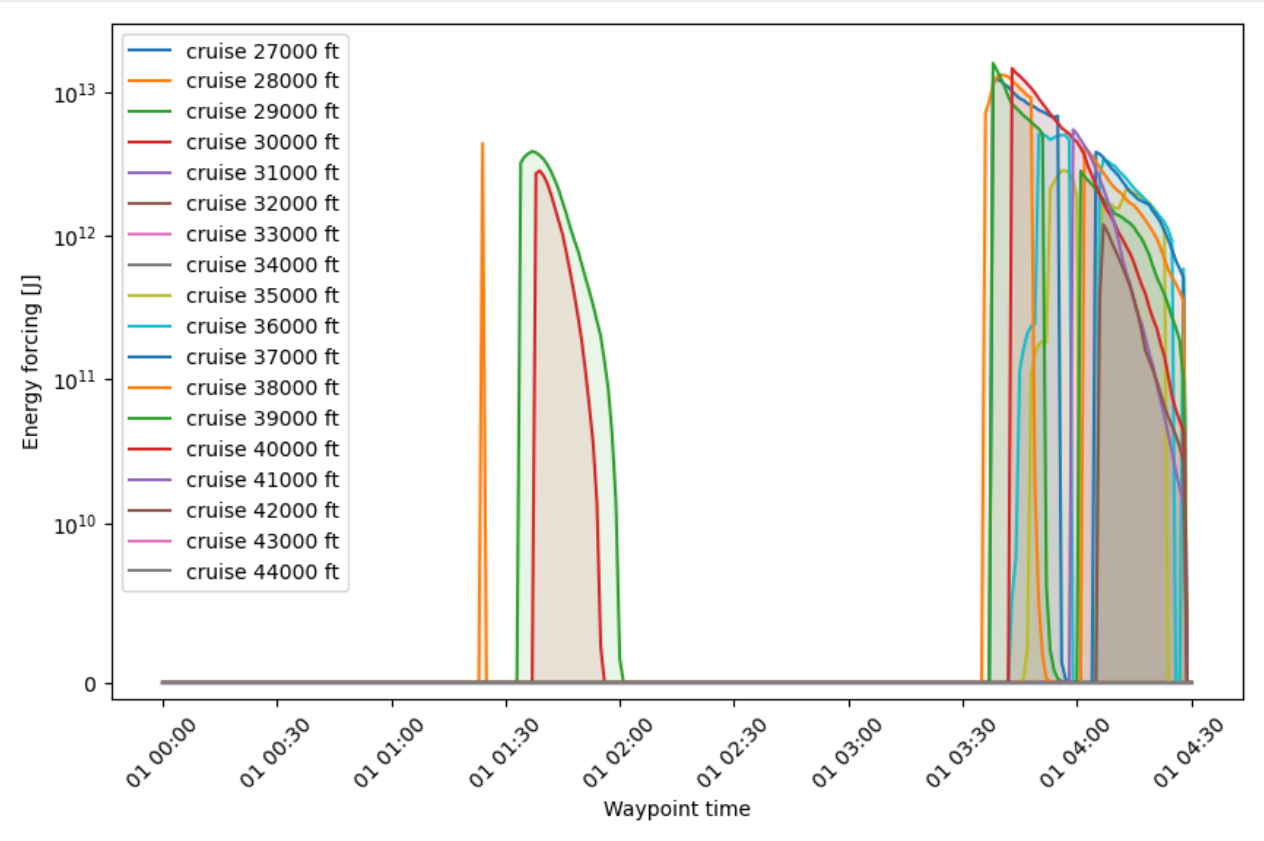
	Energy Forcing [TJ]
cruise 27000 ft	0
cruise 28000 ft	4
cruise 29000 ft	44
cruise 30000 ft	18
cruise 31000 ft	0
cruise 32000 ft	0
cruise 33000 ft	0
cruise 34000 ft	0
cruise 35000 ft	50
cruise 36000 ft	88
cruise 37000 ft	208

(continues on next page)

(continued from previous page)

```
cruise 38000 ft                  188
cruise 39000 ft                  155
cruise 40000 ft                  185
cruise 41000 ft                   42
cruise 42000 ft                    8
cruise 43000 ft                    0
cruise 44000 ft                    0
```

[7]:
```
fig, ax = plt.subplots(figsize=(10, 6))

for fl in fl_list:
    x = fl["time"]
    y = fl["ef"]
    ax.plot(x, y, label=fl.attrs["flight_id"])
    ax.fill_between(x, y, alpha=0.1)

ax.set_ylabel("Energy forcing [J]")
ax.set_xlabel("Waypoint time")

for tick in ax.get_xticklabels():
    tick.set_rotation(45)

ax.set_yscale("symlog", linthresh=1e10)
ax.set_ylim(-1e9, 3e13)

ax.legend();
```


cruise 27000 ft
cruise 28000 ft
cruise 29000 ft
cruise 30000 ft
cruise 31000 ft
cruise 32000 ft
cruise 33000 ft
cruise 34000 ft
cruise 35000 ft
cruise 36000 ft
cruise 37000 ft
cruise 38000 ft
cruise 39000 ft
cruise 40000 ft
cruise 41000 ft
cruise 42000 ft
cruise 43000 ft
cruise 44000 ft
Energy forcing [J]
10^{13}
10^{12}
10^{11}
10^{10}
0
01 00:00
01 00:30
01 01:00
01 01:30
01 02:00
01 02:30
01 03:00
01 03:30
01 04:00
01 04:30
Waypoint time

CHAPTER

SIX

MODELS

See *Running Notebooks* to interact with these notebooks

6.1 ISSR

Model ice super-saturated regions (ISSR) of the atmosphere.

6.1.1 Met Data

- Requires account with Copernicus Data Portal and local `~/.cdsapirc file` with credentials.

[1]:
```
import matplotlib.pyplot as plt
import pandas as pd
from matplotlib.colors import ListedColormap

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.issr import ISSR

# ignore pycontrails warning about ECMWF humidity scaling
import warnings

warnings.filterwarnings(message=r"[\s\S]* humidity scaling [\s\S]*", action="ignore")
```

Get Data

[2]:
```
time = ("2022-03-01 00:00:00", "2022-03-01 08:00:00")
pressure_levels = [300, 250, 200]
variables = ["t", "q"]
```

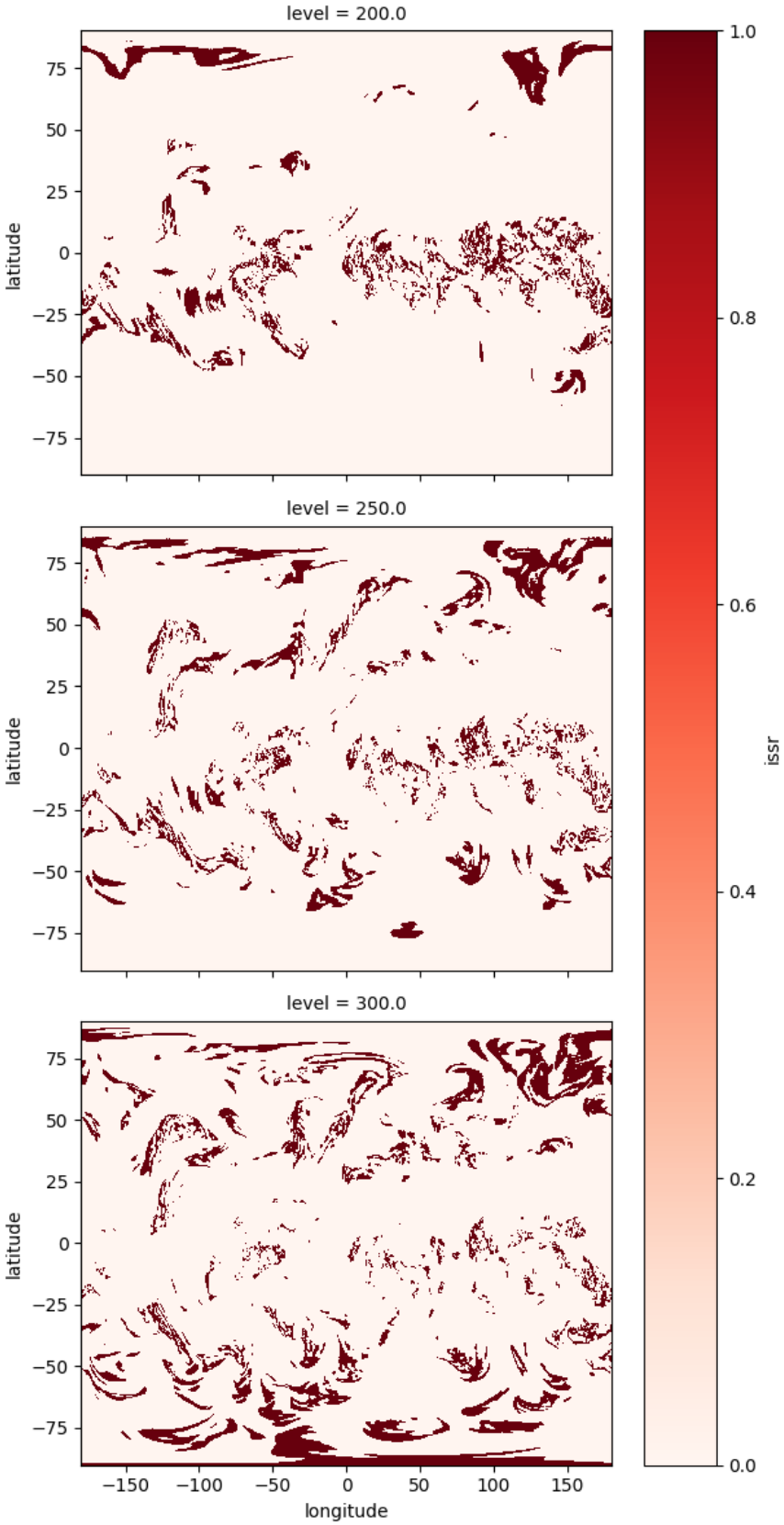
[3]:
```
era5 = ERA5(time=time, variables=variables, pressure_levels=pressure_levels)
met = era5.open_metdataset()
```

Evaluate model

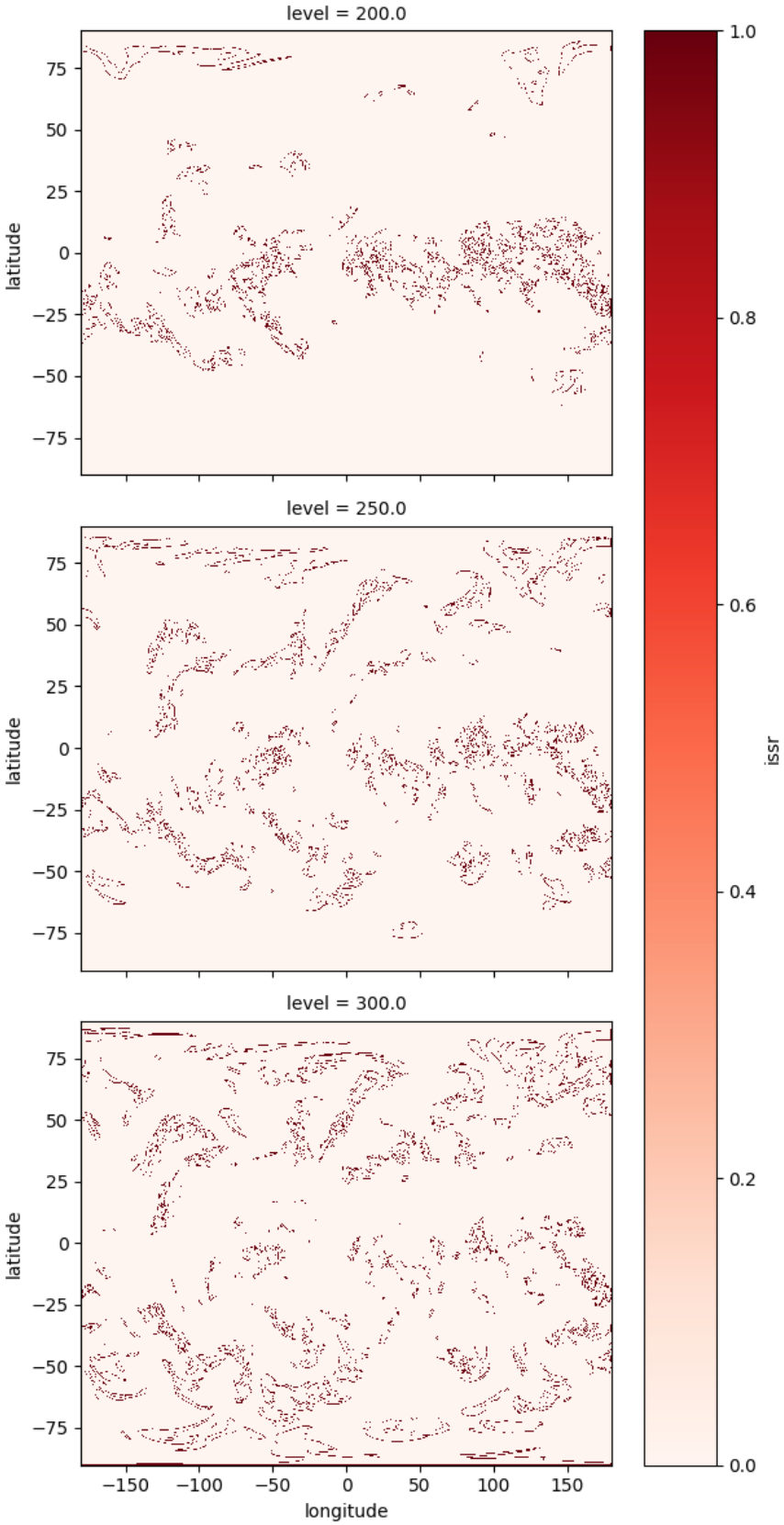
[4]:
```
# run model for across full input domain
# outputs global ice super-saturated regions as 1.0, all other regions as 0.0
issr_mds = ISSR(met).eval()
issr = issr_mds["issr"]
```

[5]:
```
# edge detection algorithm using differentiation to reduce the areas to lines
issr_edges = issr.find_edges()
```

[6]:
```
da = issr.data.isel(time=0)
da.plot(x="longitude", y="latitude", row="level", cmap="Reds", figsize=(6, 12));
```


level = 200.0
level = 250.0
level = 300.0
latitude
longitude
issr
1.0
0.8
0.6
0.4
0.2
0.0
75
50
25
0
−25
−50
−75
−150
−100
−50
0
50
100
150

[7]:
```
# plot issr edges for each pressure level
da = issr_edges.data.isel(time=0)
da.plot(x="longitude", y="latitude", row="level", cmap="Reds", figsize=(6, 12));
```


level = 200.0
level = 250.0
level = 300.0
latitude
longitude
issr
75
50
25
0
−25
−50
−75
−150
−100
−50
0
50
100
150
1.0
0.8
0.6
0.4
0.2
0.0

Interpolate

Run model along a flight path

[8]:
```
# Load flight
df = pd.read_csv("data/flight.csv", parse_dates=["time"])
fl = Flight(data=df, flight_id="acdd1b", callsign="AAL1158")

fl
```

[8]:
```
Flight [4 keys x 175 length, 3 attributes]
    Keys: longitude, latitude, altitude, time
    Attributes:
    time                    [2022-03-01 00:50:00, 2022-03-01 03:47:00]
    longitude               [-97.026, -77.036]
    latitude                [32.931, 38.854]
    altitude                [190.5, 11582.4]
    flight_id               acdd1b
    callsign                AAL1158
    crs                     EPSG:4326
```

[9]:
```
# run model for across full input domain
# outputs global ice super-saturated regions as 1, all other regions as 0
# np.nan is returned outside of the met domain
fl_out = ISSR(met=met).eval(source=fl)
fl_out["issr"]
```

[9]:
```
array([nan, nan,  0., nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
        0.,  0.,  0.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  0.,  1., nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan])
```

[10]:
```
# Get the length of the Flight in the ISSR region
fl_out.length_met("issr")
```

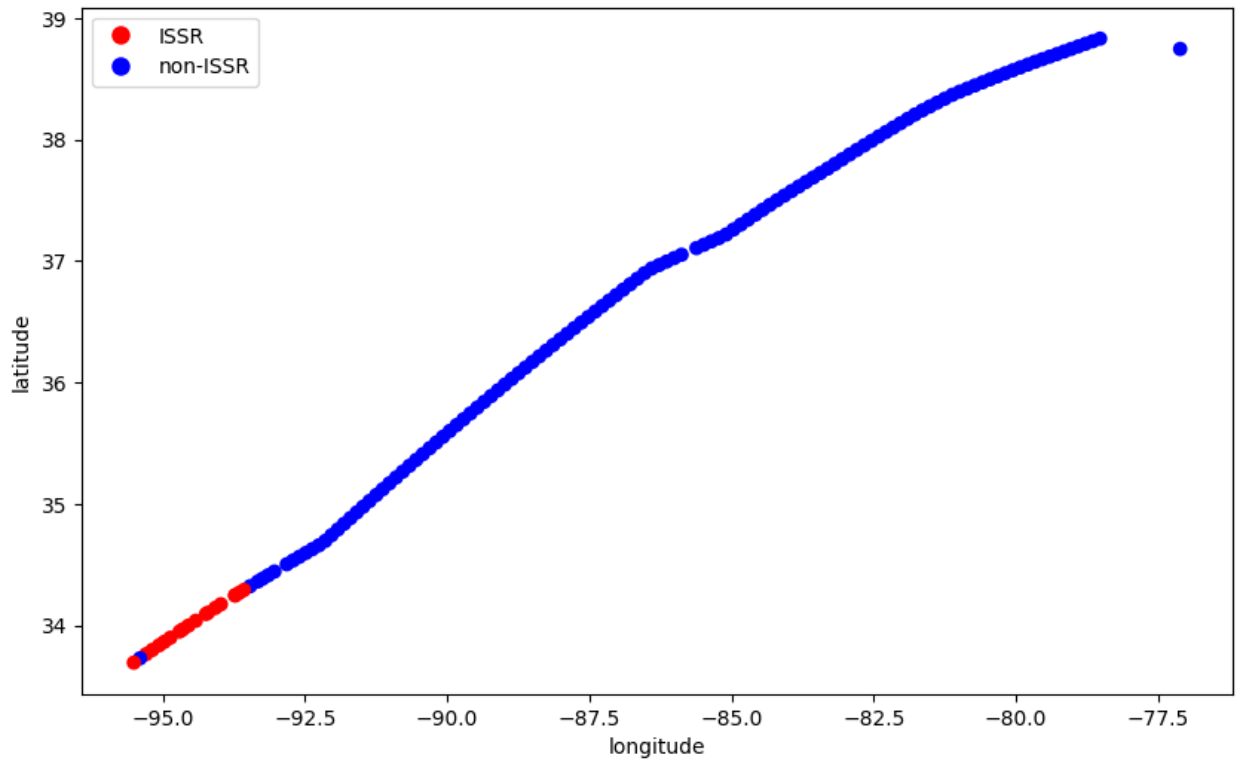
[10]:
```
190812.60842238227
```

[11]:
```
fig, ax = plt.subplots(figsize=(10, 6))

# Create colormap with red for ISSR and blue for non-ISSR
cmap = ListedColormap(["b", "r"])

ax.scatter(fl_out["longitude"], fl_out["latitude"], c=fl_out["issr"], cmap=cmap)
```

(continues on next page)

(continued from previous page)

```
# Create legend
legend_elements = [
    plt.Line2D([0], [0], marker="o", color="w", label="ISSR", markerfacecolor="r",␣
↪markersize=10),
    plt.Line2D(
        [0], [0], marker="o", color="w", label="non-ISSR", markerfacecolor="b",␣
↪markersize=10
    ),
]
ax.legend(handles=legend_elements, loc="upper left")

ax.set(xlabel="longitude", ylabel="latitude");
```



6.2 Schmidt-Appleman criterion

Model areas of the atmosphere that satisfy the Schmidt-Appleman criterion (SAC).

6.2.1 Met Data

- Requires account with Copernicus Data Portal and local `~/.cdsapirc file` with credentials.

[1]:
```
import pandas as pd

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.sac import SAC

# ignore pycontrails warning about ECMWF humidity scaling
import warnings

warnings.filterwarnings(message=r"[\s\S]* humidity scaling [\s\S]*", action="ignore")
```

Get Data

[2]:
```
time = ("2022-03-01 00:00:00", "2022-03-01 03:00:00")
pressure_levels = [300, 250, 200]
variables = ["t", "q"]  # only temperature and humidity are needed for SAC
```

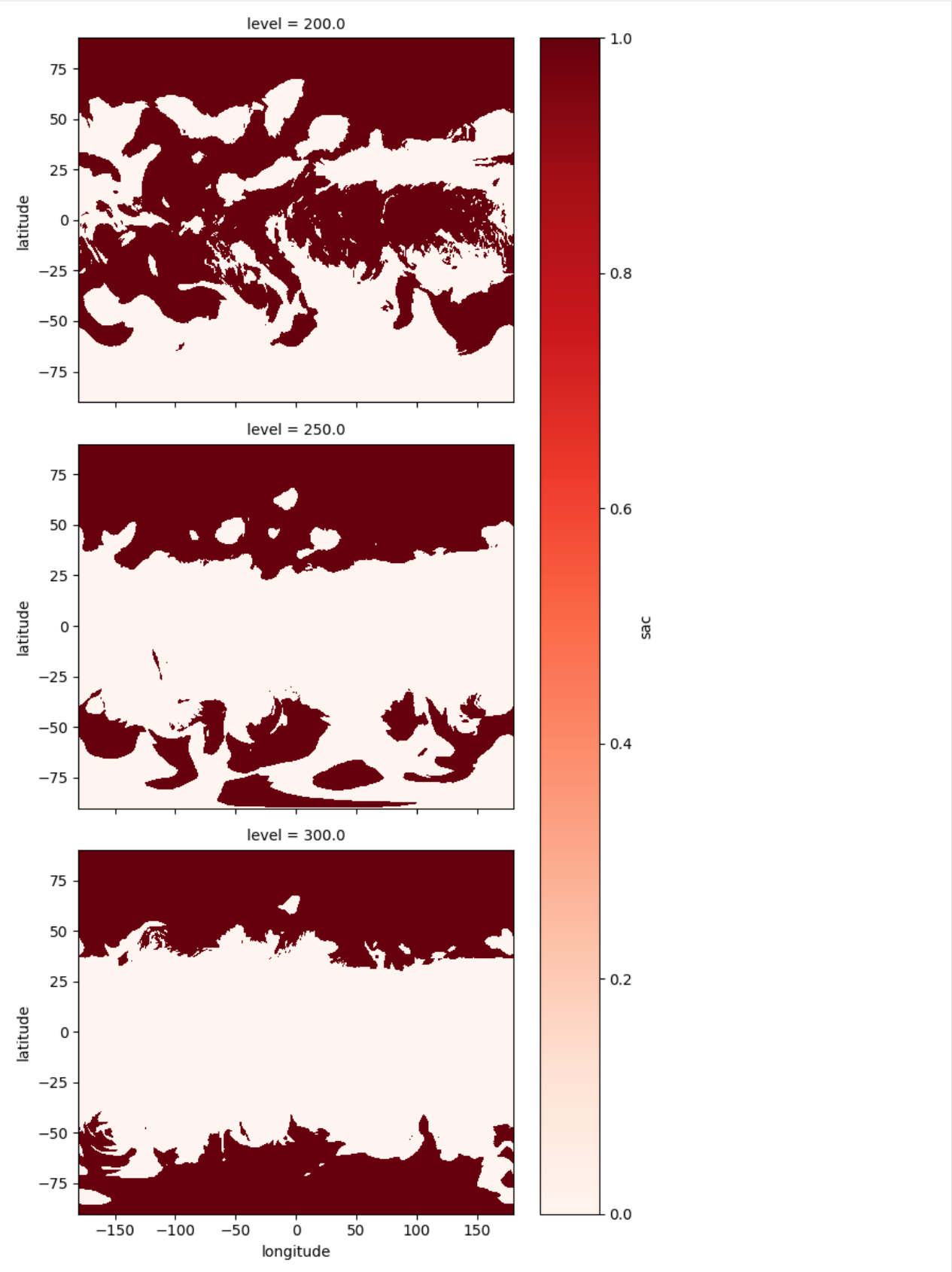
[3]:
```
era5 = ERA5(time=time, variables=variables, pressure_levels=pressure_levels)
met = era5.open_metdataset()
```

Calculations

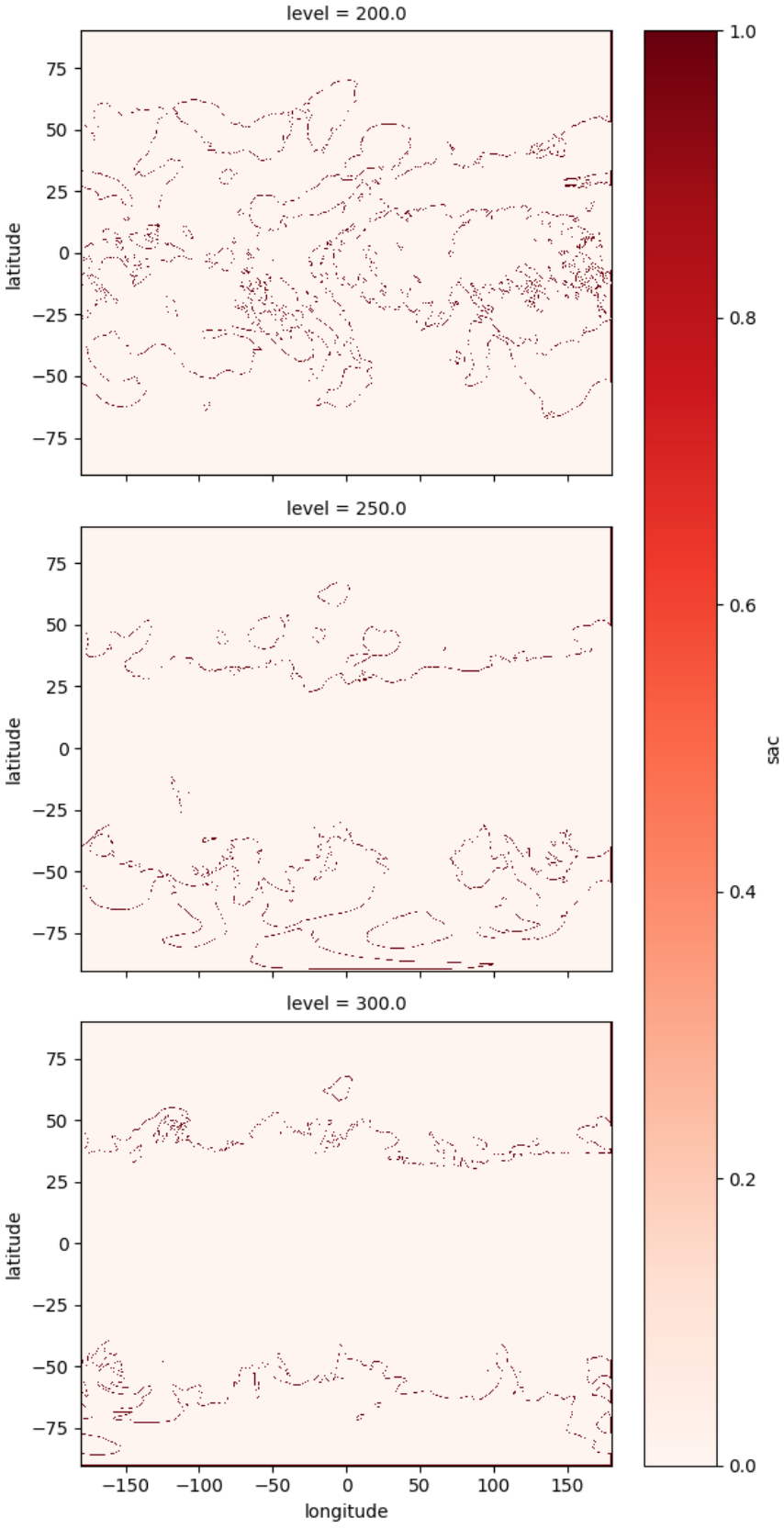
[4]:
```
# evaluate SAC on met grid
sac_mds = SAC(met=met).eval()  # returns a MetDataset
sac = sac_mds["sac"]  # extract the SAC MetDataArray

# edge detection algorithm using differentiation to reduce the areas to lines
sac_edges = sac.find_edges()
```

Figures

[5]:
```
# plot sac regions for each pressure level
da = sac.data.isel(time=0)
da.plot(x="longitude", y="latitude", row="level", cmap="Reds", figsize=(6, 12));
```


level = 200.0
level = 250.0
level = 300.0
latitude
longitude
sac
75
50
25
0
−25
−50
−75
−150
−100
−50
0
50
100
150
1.0
0.8
0.6
0.4
0.2
0.0

[6]:
```
# plot issr edges for each pressure level
da = sac_edges.data.isel(time=0)
da.plot(x="longitude", y="latitude", row="level", cmap="Reds", figsize=(6, 12));
```


level = 200.0
level = 250.0
level = 300.0
latitude
longitude
sac
1.0
0.8
0.6
0.4
0.2
0.0
75
50
25
0
−25
−50
−75
−150
−100
−50
0
50
100
150

Interpolate

Run model along a flight path

[7]:
```
# Load flight
df = pd.read_csv("data/flight.csv", parse_dates=["time"])
fl = Flight(data=df, flight_id="acdd1b", callsign="AAL1158")
```

[8]:
```
# run model for across full input domain
# outputs global ice super-saturated regions as 1, all other regions as 0
# np.nan is returned outside of the met domain
fl_out = SAC(met).eval(source=fl)
fl_out["sac"]
```

[8]:
```
array([nan, nan,  1., nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1., nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan])
```

[9]:
```
# Get the length of the Flight in the SAC region
fl_out.length_met("sac")
```

[9]:
```
1390898.4268105943
```

[10]:
```
# The SAC is sensitive to the engine efficiency of the aircraft.
# As this value decreases, fewer waypoints satisfy the SAC.
fl_out = SAC(met, engine_efficiency=0.0).eval(source=fl)
fl_out["sac"]
```

[10]:
```
array([nan, nan,  0., nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
        0.,  1.,  1.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,
        1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1., nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan,
       nan, nan, nan, nan, nan, nan])
```

6.3 Aircraft Performance

Aircraft performance affects engine emissions and contrail formation.

`pycontrails` includes built-in support for two aircraft performance models:

- Poll-Schumann Model
- BADA (*Requires* pycontrails-bada *extension and data files obtained through the* BADA license.)

6.3.1 Alternate Models

Aircraft performance can be pre-computed and provided directly to models. Other performance models include:

- OpenAP: Open-source aircraft performance and emissions model
- PianoX: Aircraft design, performance, and emissions tool

Aircraft OEMs and flight planning systems generally have the most accurate performance models.

6.3.2 Poll-Schumann Trajectory Model

The Poll-Schumann model was developed to address the need for simple, yet accurate, methods for the estimation of cruise fuel burn and other important aircraft performance parameters.

References

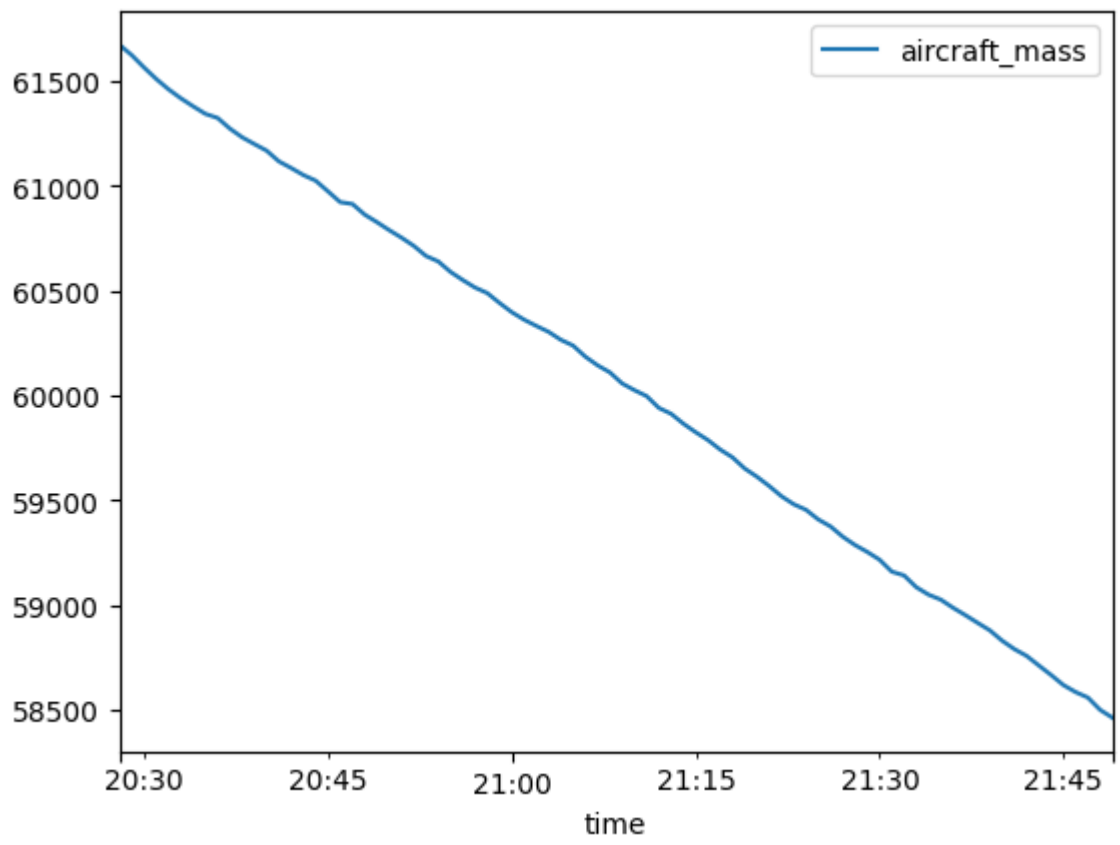
- D.I.A. Poll and U. Schumann. An estimation method for the fuel burn and other performance characteristics of civil transport aircraft in the cruise. Part 1 fundamental quantities and governing relations for a general atmosphere. The Aeronautical Journal, 125(1284):257–295, February 2021. doi:10.1017/aer.2020.62.
- D.I.A. Poll and U. Schumann. An estimation method for the fuel burn and other performance characteristics of civil transport aircraft during cruise: part 2, determining the aircraft's characteristic parameters. The Aeronautical Journal, 125(1284):296–340, February 2021. doi:10.1017/aer.2020.124.

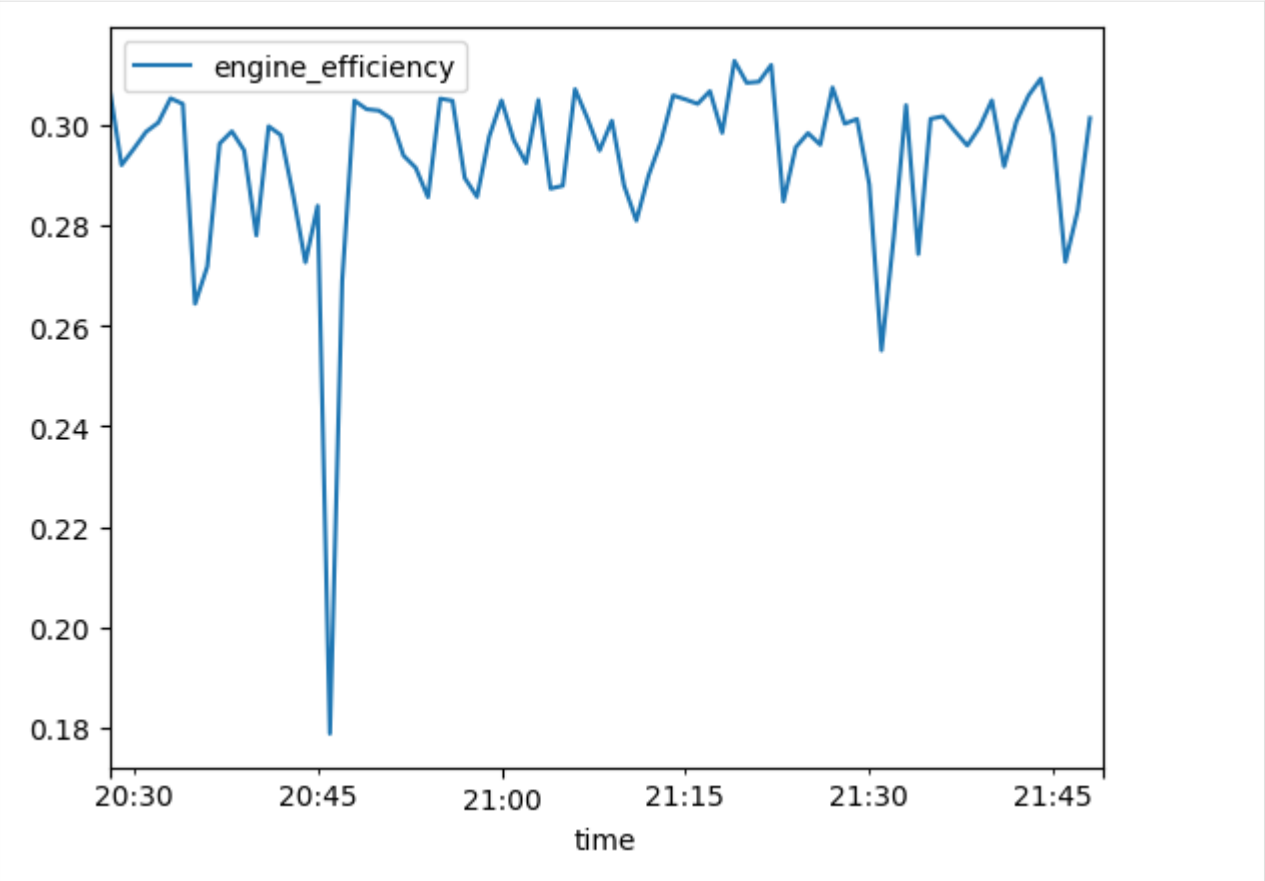
[1]:
```
import pandas as pd

from pycontrails import Flight
from pycontrails.physics import units

from pycontrails.models.ps_model import PSFlight
```

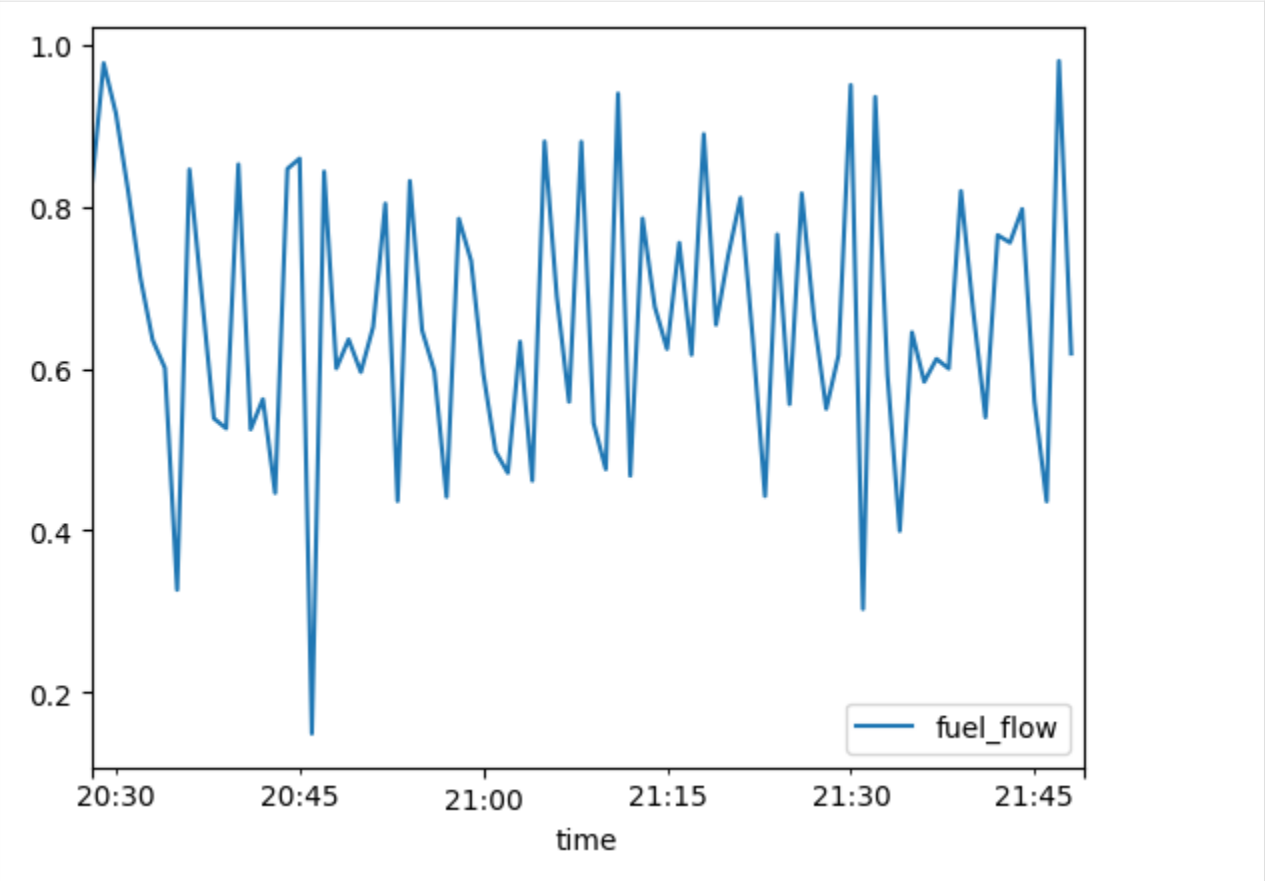
[2]:
```
# load flight
attrs = {"flight_id": "1", "aircraft_type": "A320"}
flight = Flight(data=pd.read_csv("data/flight-ap.csv"), attrs=attrs)

# select waypoints between initial climb and final decent
flight = flight.filter(flight["altitude"] > units.ft_to_m(33000))

# Estimate temperature using ISA
flight["air_temperature"] = units.m_to_T_isa(flight["altitude"])

# Estimate airspeed by using groundspeed
flight["true_airspeed"] = flight.segment_groundspeed()
```

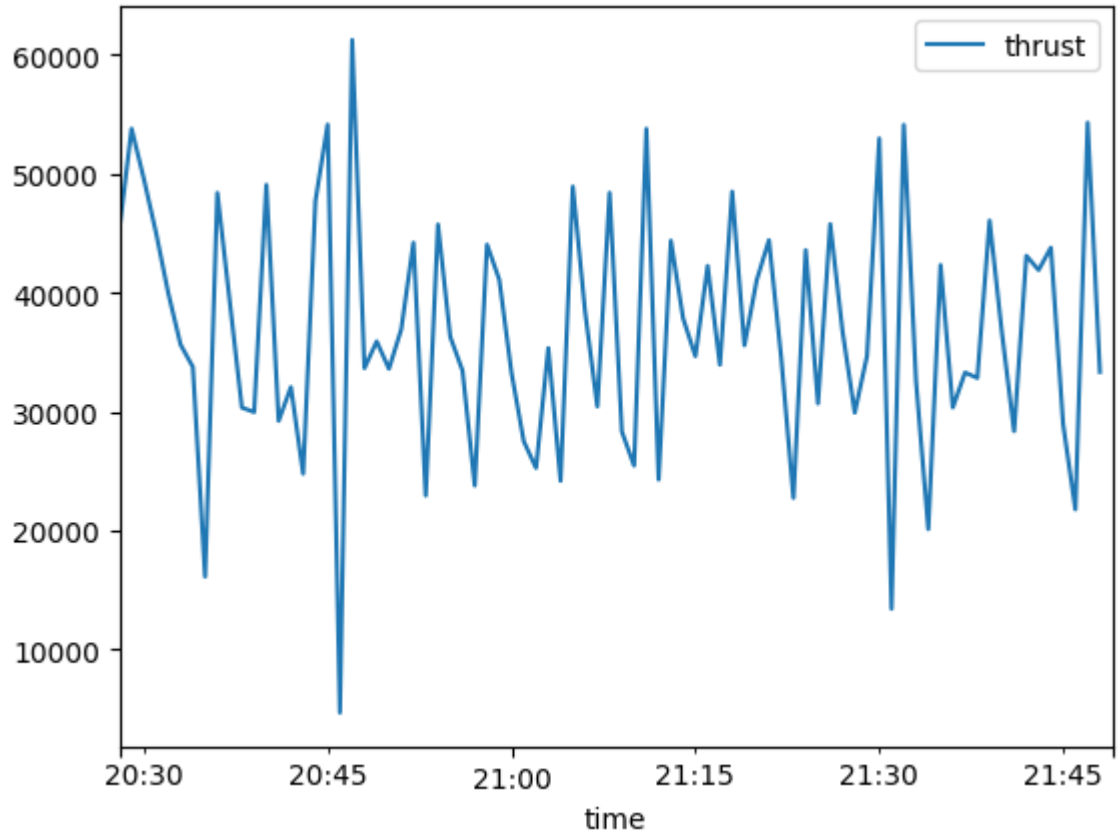
[3]:
```
# Create PS Flight model and evaluate
ps_model = PSFlight()
out = ps_model.eval(flight)
```

[4]:
```
# Visualize outputs
out.dataframe.plot(x="time", y="aircraft_mass")
out.dataframe.plot(x="time", y="engine_efficiency")
out.dataframe.plot(x="time", y="fuel_flow")
out.dataframe.plot(x="time", y="thrust");
```




engine_efficiency
0.30
0.28
0.26
0.24
0.22
0.20
0.18
20:30
20:45
21:00
21:15
21:30
21:45
time


1.0
0.8
0.6
0.4
0.2
fuel_flow
20:30
20:45
21:00
21:15
21:30
21:45
time



6.3.3 Poll-Schumann Grid Model

The `PSGrid` model exists to compute nominal Poll-Schumann aircraft performance on a meteorological grid.

Goal

For a given aircraft type, altitude, aircraft mass, air temperature, and mach number, the PS model computes a theoretical engine efficiency and fuel flow rate for an aircraft under cruise conditions. Letting the aircraft mass vary and fixing the other parameters, the engine efficiency curve attains a single maximum at a particular aircraft mass. By solving this implicit equation, the PS model can be used to compute the aircraft mass that maximizes engine efficiency for a given set of parameters. This is the "nominal" aircraft mass computed by this model.

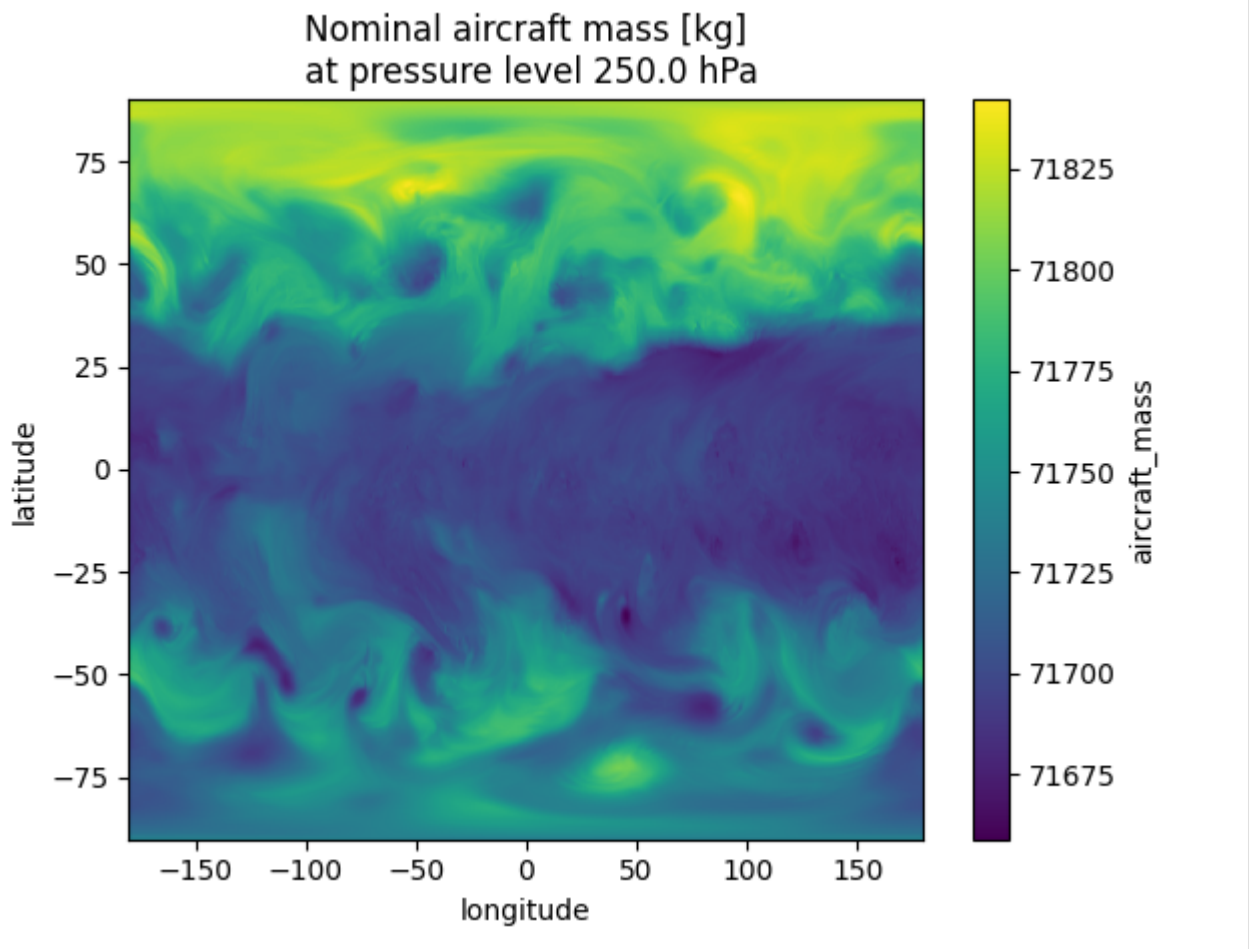
This nominal aircraft mass is not always realizable. For example, the maximum engine efficiency may be attained at an aircraft mass that is less than the operating empty mass of the aircraft. This model determines the minimum and maximum possible aircraft mass for a given set of parameters using a simple heuristic. The nominal aircraft mass is then clipped to this range.

[5]:
```
from matplotlib import pyplot as plt

from pycontrails.datalib.ecmwf import ERA5

from pycontrails.models.ps_model import PSGrid
```

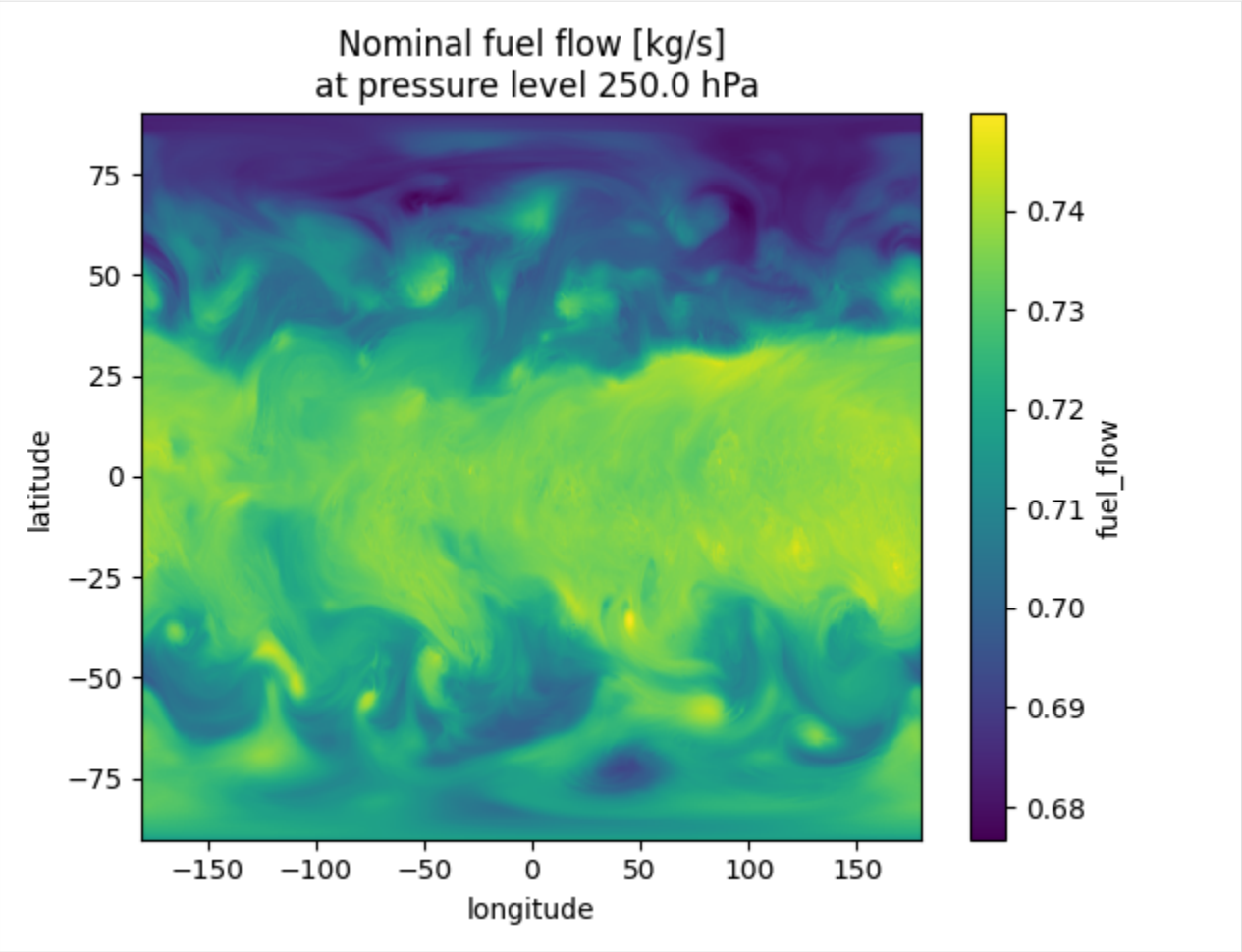
[6]:
```
# load meteorology data
time = "2022-03-01 07:00:00"
pressure_levels = [250]
variables = ["air_temperature"]

era5 = ERA5(time=time, variables=variables, pressure_levels=pressure_levels)
met = era5.open_metdataset()
```

[7]:
```
# Run PSGrid model
model = PSGrid(met, aircraft_type="A320")
out = model.eval()
```

[8]:
```
out["aircraft_mass"].data.plot(x="longitude", y="latitude")
plt.title(f"Nominal aircraft mass [kg]\n at pressure level {out.data['level'].values[0]}␣
↪hPa");
```



[9]:
```
out["fuel_flow"].data.plot(x="longitude", y="latitude")
plt.title(f"Nominal fuel flow [kg/s]\n at pressure level {out.data['level'].values[0]}␣
↪hPa");
```



6.4 CoCiP

Contrail Cirrus Predicition (CoCiP) model evaluation along a flight trajectory.

6.4.1 References

- Schumann, U. "A Contrail Cirrus Prediction Model." Geoscientific Model Development 5, no. 3 (May 3, 2012): 543–80. https://doi.org/10.5194/gmd-5-543-2012.
- Schumann, U., B. Mayer, K. Graf, and H. Mannstein. "A Parametric Radiative Forcing Model for Contrail Cirrus." Journal of Applied Meteorology and Climatology 51, no. 7 (July 2012): 1391–1406. https://doi.org/10.1175/JAMC-D-11-0242.1.
- Schumann, Ulrich, Robert Baumann, Darrel Baumgardner, Sarah T. Bedka, David P. Duda, Volker Freudenthaler, Jean-Francois Gayet, et al. 2017. "Properties of Individual Contrails: A Compilation of Observations and Some Comparisons." Atmospheric Chemistry and Physics 17 (1): 403–38. https://doi.org/10.5194/acp-17-403-2017.
- Teoh, Roger, Ulrich Schumann, Arnab Majumdar, and Marc E. J. Stettler. "Mitigating the Climate Forcing of Aircraft Contrails by Small-Scale Diversions and Technology Adoption." Environmental Science & Technology 54, no. 5 (March 3, 2020): 2941–50. https://doi.org/10.1021/acs.est.9b05608.

- Teoh, Roger, Ulrich Schumann, Edward Gryspeerdt, Marc Shapiro, Jarlath Molloy, George Koudis, Christiane Voigt, and Marc E. J. Stettler. 2022. "Aviation Contrail Climate Effects in the North Atlantic from 2016 to 2021." Atmospheric Chemistry and Physics 22 (16): 10919–35. https://doi.org/10.5194/acp-22-10919-2022.
- Teoh, Roger, Ulrich Schumann, Christiane Voigt, Tobias Schripp, Marc Shapiro, Zebediah Engberg, Jarlath Molloy, George Koudis, and Marc E. J. Stettler. 2022. "Targeted Use of Sustainable Aviation Fuel to Maximize Climate Benefits." Environmental Science & Technology, November. https://doi.org/10.1021/acs.est.2c05781.

[1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt

from pycontrails import Flight, MetDataset
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.cocip import (
    Cocip,
    flight_waypoint_summary_statistics,
    contrail_flight_summary_statistics,
    natural_cirrus_properties_to_hi_res_grid,
)
from pycontrails.models.humidity_scaling import ConstantHumidityScaling

plt.rcParams["figure.figsize"] = (10, 6)
```

6.4.2 Download meteorology data from ECMWF

This demo uses ERA5 via the Copernicus Data Store (CDS) for met data. This requires account with Copernicus Data Portal and local `~/.cdsapirc` file with credentials.

> Note this will download ~1 GB of meteorology data to your computer

[2]:
```
time_bounds = ("2022-03-01 00:00:00", "2022-03-01 23:00:00")
pressure_levels = (300, 250, 200)
```

[3]:
```
era5pl = ERA5(
    time=time_bounds,
    variables=Cocip.met_variables + Cocip.optional_met_variables,
    pressure_levels=pressure_levels,
)
era5sl = ERA5(time=time_bounds, variables=Cocip.rad_variables)
```

[4]:
```
# download data from ERA5 (or open from cache)
met = era5pl.open_metdataset()
rad = era5sl.open_metdataset()
```

6.4.3 Load Flight Data

`Flight` can be loaded from CSV, parquet, or created from a pandas DataFrame

[5]:
```
# demo synthetic flight
flight_attrs = {
    "flight_id": "test",
    # set constants along flight path
    "true_airspeed": 226.099920796651,  # true airspeed, m/s
    "thrust": 0.22,  # thrust_setting
    "nvpm_ei_n": 1.897462e15,  # non-volatile emissions index
    "aircraft_type": "E190",
    "wingspan": 48,  # m
    "n_engine": 2,
}

# Example flight
df = pd.DataFrame()
df["longitude"] = np.linspace(-25, -40, 100)
df["latitude"] = np.linspace(34, 40, 100)
df["altitude"] = np.linspace(10900, 10900, 100)
df["engine_efficiency"] = np.linspace(0.34, 0.35, 100)
df["fuel_flow"] = np.linspace(2.1, 2.4, 100)  # kg/s
df["aircraft_mass"] = np.linspace(154445, 154345, 100)  # kg
df["time"] = pd.date_range("2022-03-01T00:15:00", "2022-03-01T02:30:00", periods=100)

flight = Flight(df, attrs=flight_attrs)
flight
```

[5]:
```
Flight [7 keys x 100 length, 8 attributes]
    Keys: longitude, latitude, altitude, engine_efficiency, fuel_flow, ..., time
    Attributes:
    time                [2022-03-01 00:15:00, 2022-03-01 02:30:00]
    longitude           [-40.0, -25.0]
    latitude            [34.0, 40.0]
    altitude            [10900.0, 10900.0]
    flight_id           test
    true_airspeed       226.099920796651
    thrust              0.22
    nvpm_ei_n           1897462000000000.0
    aircraft_type       E190
    wingspan            48
    n_engine            2
    crs                 EPSG:4326
```

6.4.4 Run `Cocip` on a single flight

In this first example, the `Flight` has aircraft performance (i.e. `nvpm_ei_n`) hardcoded into the data as constants. This data is assumed to be constant at every flight waypoint.

Caveat

- When the `Cocip` model is run on one `Flight`, the default behavior is to **downselect the meteorology** to a region surrounding the flight and **process the meteorology** (i.e. humidity scaling, `tau_cirrus`) on the smaller domain. The implications of this processing is each instance of a single-flight `Cocip` model should *only be run once*.
- We ignore the warning here about humidity scaling for ECMWF data sources

[6]:
```
params = {
    "dt_integration": np.timedelta64(10, "m"),
    # The humidity_scaling parameter is only used for ECMWF ERA5 data
    # based on Teoh 2020 and Teoh 2022 - https://acp.copernicus.org/preprints/acp-2022-
↪169/acp-2022-169.pdf
    # Here we use an example of constantly scaling the humidity value by 0.99
    "humidity_scaling": ConstantHumidityScaling(rhi_adj=0.99),
}
cocip = Cocip(met=met, rad=rad, params=params)
```

[7]:
```
output_flight = cocip.eval(source=flight)
```

6.4.5 Explore Flight Output

The `output_flight` object holds roughly 50 variables of interest. The energy forcing `ef` field is a primary model output. Waypoints not producing persistent contrails are assigned an `ef` value of 0.

[8]:
```
df = output_flight.dataframe
df.head()
```

[8]:

	waypoint	longitude	latitude	altitude	engine_efficiency	fuel_flow	\
0	0	-25.000000	34.000000	10900.0	0.340000	2.100000	
1	1	-25.151515	34.060606	10900.0	0.340101	2.103030	
2	2	-25.303030	34.121212	10900.0	0.340202	2.106061	
3	3	-25.454545	34.181818	10900.0	0.340303	2.109091	
4	4	-25.606061	34.242424	10900.0	0.340404	2.112121	

	aircraft_mass	time	flight_id	level	...	\
0	154445.000000	2022-03-01 00:15:00.000000000	test	229.908663	...	
1	154443.989899	2022-03-01 00:16:21.818181818	test	229.908663	...	
2	154442.979798	2022-03-01 00:17:43.636363636	test	229.908663	...	
3	154441.969697	2022-03-01 00:19:05.454545454	test	229.908663	...	
4	154440.959596	2022-03-01 00:20:27.272727272	test	229.908663	...	

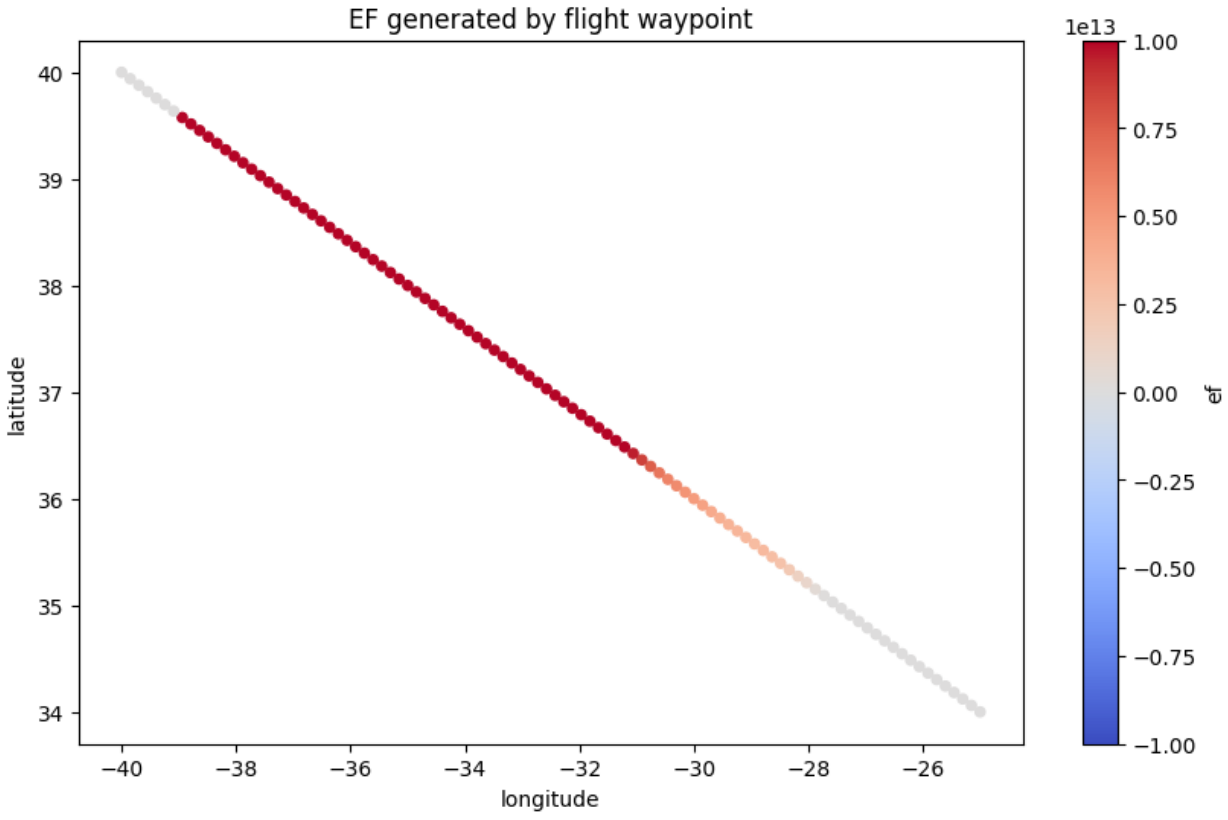
	n_ice_per_m_1	ef	contrail_age	sdr_mean	rsr_mean	olr_mean	rf_sw_mean	\
0	1.211936e+13	0.0	0 days	NaN	NaN	NaN	NaN	
1	1.299153e+13	0.0	0 days	NaN	NaN	NaN	NaN	
2	1.363941e+13	0.0	0 days	NaN	NaN	NaN	NaN	
3	1.410562e+13	0.0	0 days	NaN	NaN	NaN	NaN	
4	1.438106e+13	0.0	0 days	NaN	NaN	NaN	NaN	

(continues on next page)

(continued from previous page)

```
   rf_lw_mean  rf_net_mean  cocip
0         NaN          NaN    0.0
1         NaN          NaN    0.0
2         NaN          NaN    0.0
3         NaN          NaN    0.0
4         NaN          NaN    0.0

[5 rows x 45 columns]
```

[9]:
```
df.plot.scatter(
    x="longitude",
    y="latitude",
    c="ef",
    cmap="coolwarm",
    vmin=-1e13,
    vmax=1e13,
    title="EF generated by flight waypoint",
);
```

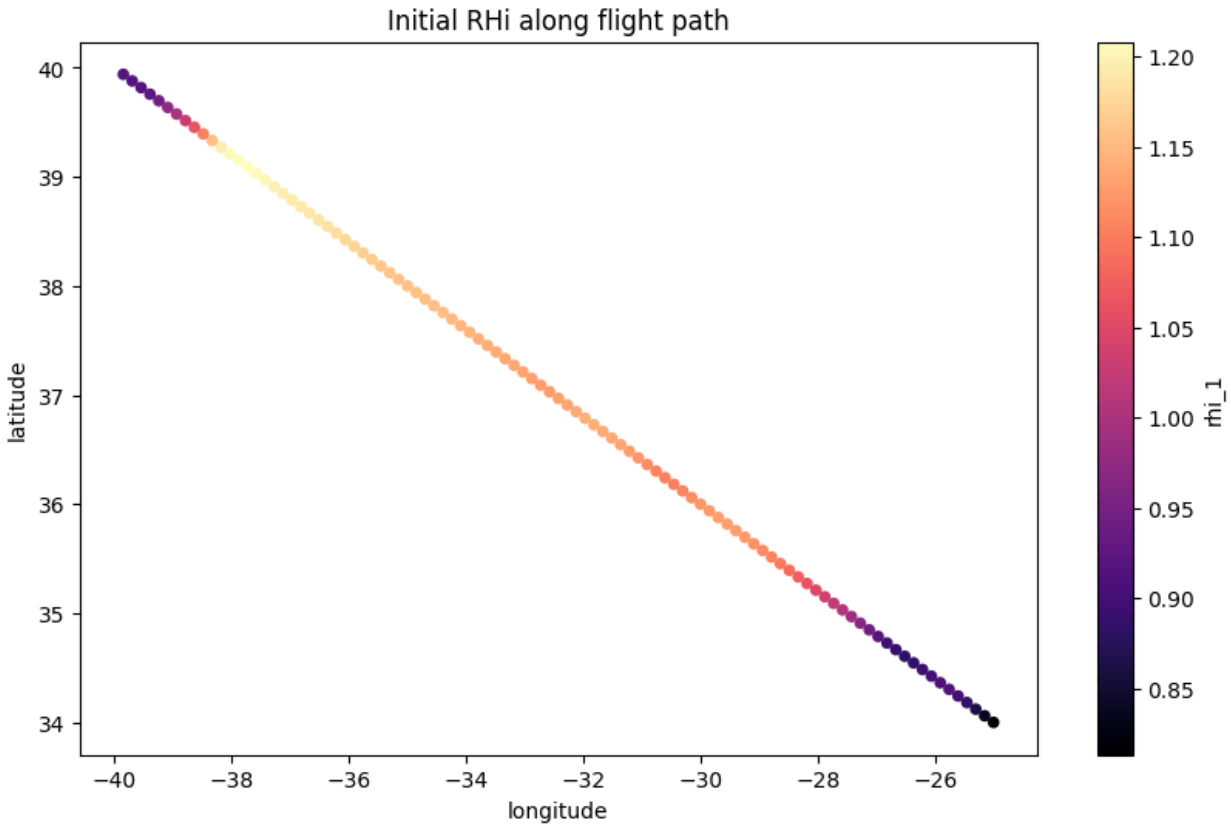


[10]:
```
df.plot.scatter(
    x="longitude",
    y="latitude",
    c="rhi_1",
    cmap="magma",
    title="Initial RHi along flight path",
```

(continues on next page)

(continued from previous page)

```
);
```



6.4.6 Explore Contrail Output

- The `cocip.contrail` attribute is a `pandas.DataFrame <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html>`__ representation of the contrail waypoints.
- The `cocip.contrail_dataset` attribute is a `xarray.Dataset <https://xarray.pydata.org/en/stable/generated/xarray.Dataset.html>`__ representation of the contrail waypoints.

[11]:
```
contrail = cocip.contrail
contrail.head()
```

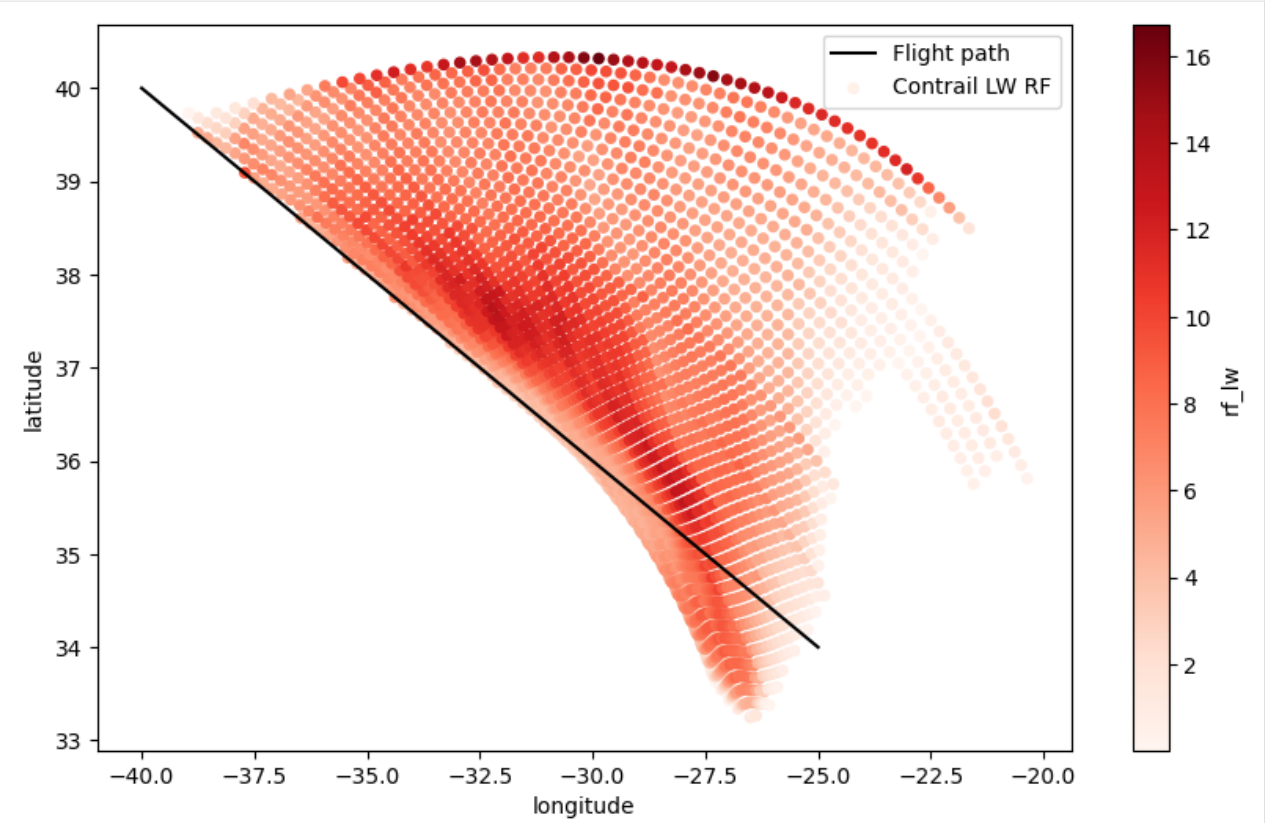
[11]:
```
   waypoint flight_id               formation_time                time  \
0        16      test 2022-03-01 00:36:49.090909090 2022-03-01 00:40:00
1        17      test 2022-03-01 00:38:10.909090909 2022-03-01 00:40:00
2        18      test 2022-03-01 00:39:32.727272727 2022-03-01 00:40:00
0        17      test 2022-03-01 00:38:10.909090909 2022-03-01 00:50:00
1        18      test 2022-03-01 00:39:32.727272727 2022-03-01 00:50:00

                        age  longitude   latitude      altitude       level  \
0 0 days 00:03:10.909090910 -27.378049  34.922418  10855.165152  231.533890
1 0 days 00:01:49.090909091 -27.548832  35.003469  10855.511035  231.521316
2           0 days 00:00:00 -27.720407  35.084249  10855.340639  231.527510
0 0 days 00:11:49.090909091 -27.401596  34.855488  10855.704079  231.514299
```

(continues on next page)

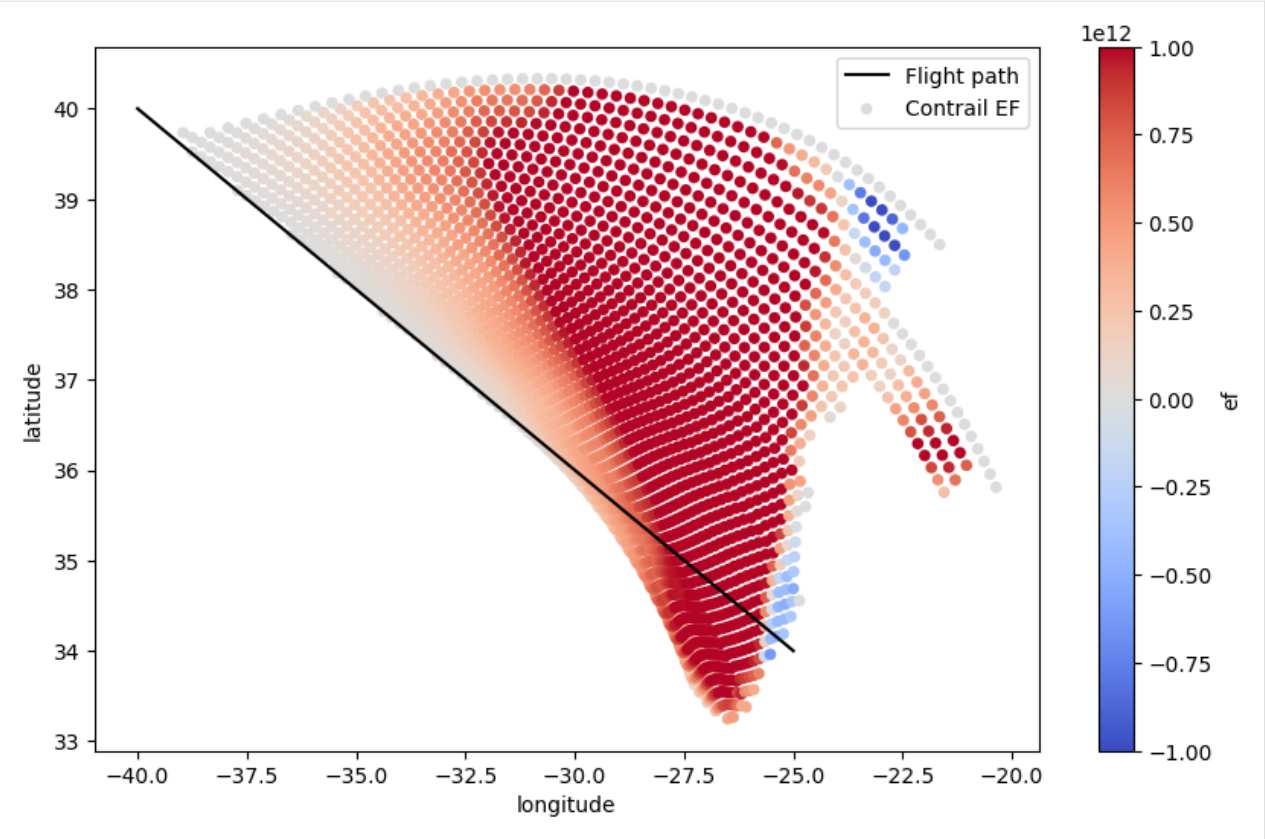
(continued from previous page)

```
1 0 days 00:10:27.272727273 -27.569577  34.937639  10857.240565  231.458453

   continuous  ...  tau_contrail     dn_dt_agg  dn_dt_turb  rf_sw     rf_lw  \
0        True  ...      0.058053  7.476665e-20    0.000051    0.0  0.555465
1        True  ...      0.165581  2.538742e-19    0.000077    0.0  1.976229
2       False  ...      0.408836  4.056703e-19    0.000233    0.0  4.600390
0        True  ...      0.036824  9.873238e-20    0.000026    0.0  0.561865
1        True  ...      0.047199  2.494850e-19    0.000030    0.0  0.854236

     rf_net  persistent            ef  timestep  age_hours
0  0.555465        True  1.099225e+09         2   0.053030
1  1.976229        True  9.428853e+08         2   0.030303
2  4.600390        True  0.000000e+00         2   0.000000
0  0.561865        True  7.060254e+09         3   0.196970
1  0.854236        True  8.330392e+09         3   0.174242

[5 rows x 56 columns]
```

[12]:
```
ax = plt.axes()

cocip.source.dataframe.plot(
    "longitude",
    "latitude",
    color="k",
    ax=ax,
    label="Flight path",
)
cocip.contrail.plot.scatter(
    "longitude",
    "latitude",
    c="rf_lw",
    cmap="Reds",
    ax=ax,
    label="Contrail LW RF",
)
ax.legend();
```



[13]:
```
ax = plt.axes()

cocip.source.dataframe.plot(
    "longitude",
    "latitude",
    color="k",
    ax=ax,
    label="Flight path",
)
cocip.contrail.plot.scatter(
    "longitude",
    "latitude",
    c="ef",
    cmap="coolwarm",
    vmin=-1e12,
    vmax=1e12,
    ax=ax,
    label="Contrail EF",
)
ax.legend();
```



6.4.7 Explore Flight Summary

A curated set of statistics available after a `Flight` has been run through `eval`

[14]:
```
# flight_statistics = cocip.output_flight_statistics()
# flight_statistics

waypoint_summary = flight_waypoint_summary_statistics(cocip.source, cocip.contrail)
flight_summary = contrail_flight_summary_statistics(waypoint_summary)
flight_summary
```

[14]:
```
  flight_id  total_flight_distance_flown  total_contrails_formed  \
0      test                1.489373e+06            1.489373e+06

   total_persistent_contrails_formed  mean_lifetime_contrail_altitude  \
0                       1.489373e+06                     10924.967626

   mean_lifetime_rhi  mean_lifetime_n_ice_per_m  mean_lifetime_r_ice_vol  \
0           1.115407               9.352614e+12                 0.000006

   mean_lifetime_contrail_width  mean_lifetime_contrail_depth  ...  \
0                   15384.89933                     535.5604  ...

   mean_lifetime_tau_cirrus  mean_contrail_lifetime  max_contrail_lifetime  \
0                  0.251802                5.364979              10.583333
```

(continues on next page)

(continued from previous page)

```
   mean_lifetime_rf_sw  mean_lifetime_rf_lw  mean_lifetime_rf_net  \
0            -0.157952             5.550725              5.392773

   total_energy_forcing  mean_lifetime_olr  mean_lifetime_sdr  \
0          2.340121e+15         192.821976          22.198721

   mean_lifetime_rsr
0           9.819747

[1 rows x 21 columns]
```

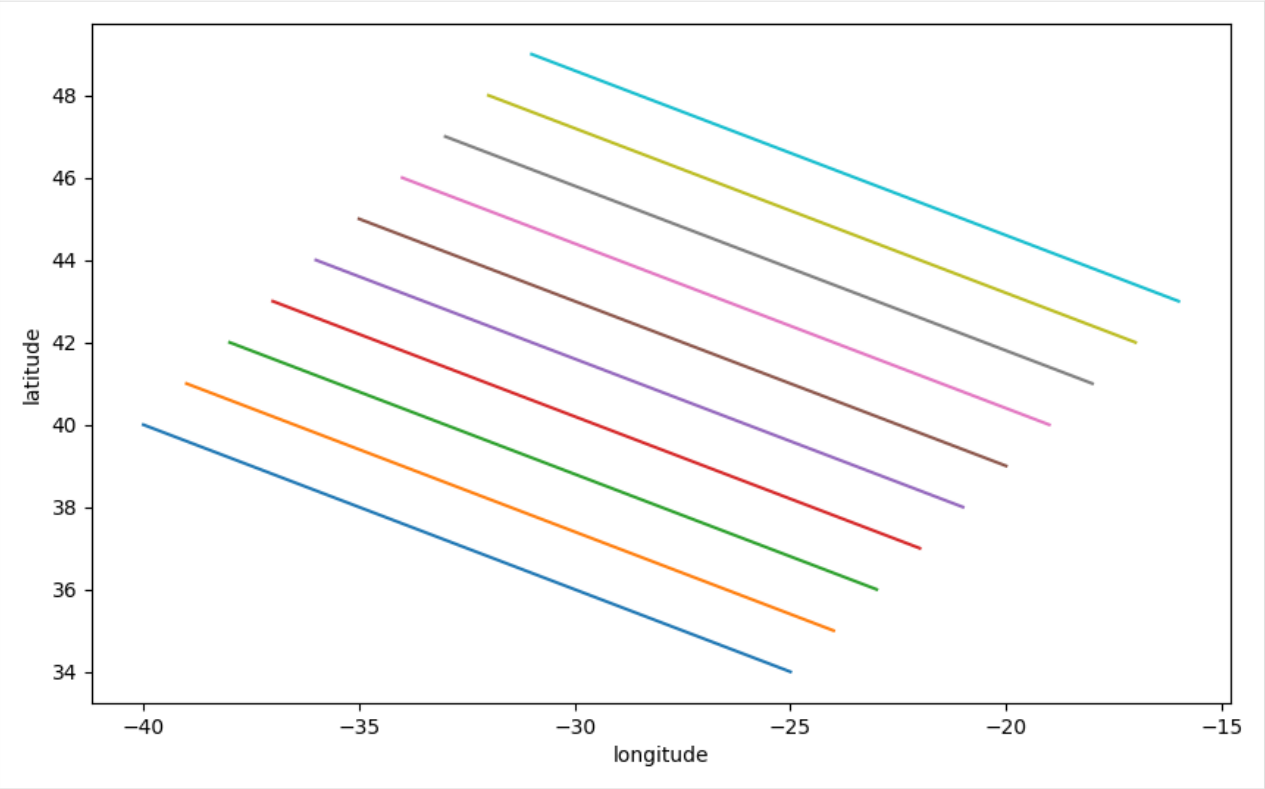
6.4.8 Run Cocip on Multiple Flights

Run multiple `Flight` inputs on a single set of meteorology.

For this demo, we'll copy the original flight and tweak its longitude and latitudes values.

[15]:
```
flights = []
for i in range(10):
    fl = flight.copy()
    fl.attrs.update(flight_id=f"test-{i:02d}")
    fl.update(latitude=flight["latitude"] + i)
    fl.update(longitude=flight["longitude"] + i)
    flights.append(fl)
```

[16]:
```
# Visualize the fleet of 10 flights
ax = plt.axes()
for fl in flights:
    fl.plot(ax=ax)
```



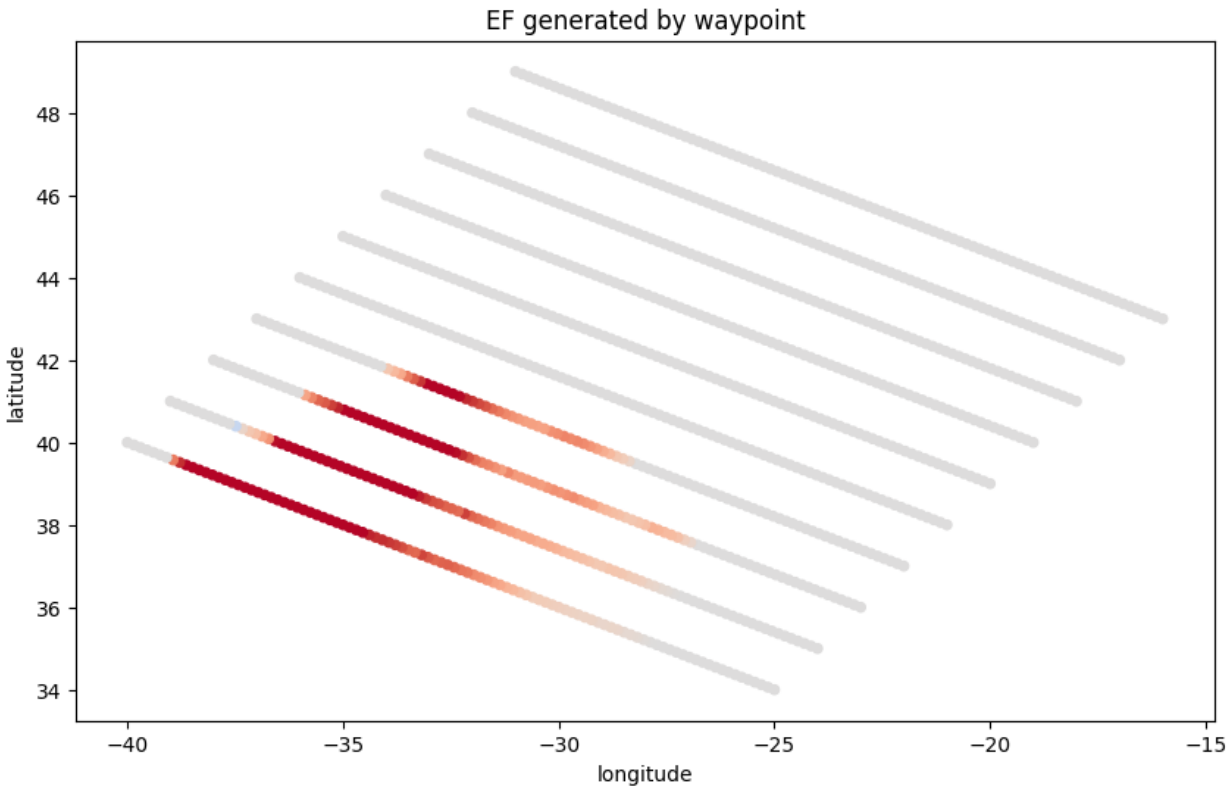
Run the `Cocip` model over a `list[Flight]` objects

[17]:
```
cocip = Cocip(
    met=met,
    rad=rad,
    process_emissions=False,
    humidity_scaling=ConstantHumidityScaling(rhi_adj=0.99),
)

# returns list of Flight outputs
output_flights = cocip.eval(source=flights)
```

[18]:
```
# print EF for each flight
for fl in output_flights:
    print(f"{fl.attrs['flight_id']}: {np.sum(fl['ef'])}")
```

```
test-00: 2121480685044636.5
test-01: 1541994819791262.5
test-02: 1319374480518468.0
test-03: 703275057307035.0
test-04: 637174779003.3167
test-05: 4260062451.204625
test-06: 0.0
test-07: 0.0
test-08: 0.0
test-09: 0.0
```

[19]:
```
# Visualize the "ef" of each flight
ax = plt.axes()
for fl in output_flights:
    fl.dataframe.plot.scatter(
        x="longitude",
        y="latitude",
        c="ef",
        cmap="coolwarm",
        vmin=-3e13,
        vmax=3e13,
        title="EF generated by waypoint",
        ax=ax,
        colorbar=False,
    );
```



6.4.9 Use the Poll-Schumann aircraft performance model

First create a Flight that does not have emissions data associated:

[20]:
```
# demo synthetic flight
flight_attrs = {
    "flight_id": "test-ps-model",
    "aircraft_type": "E195",
}

# Example flight
```

(continues on next page)

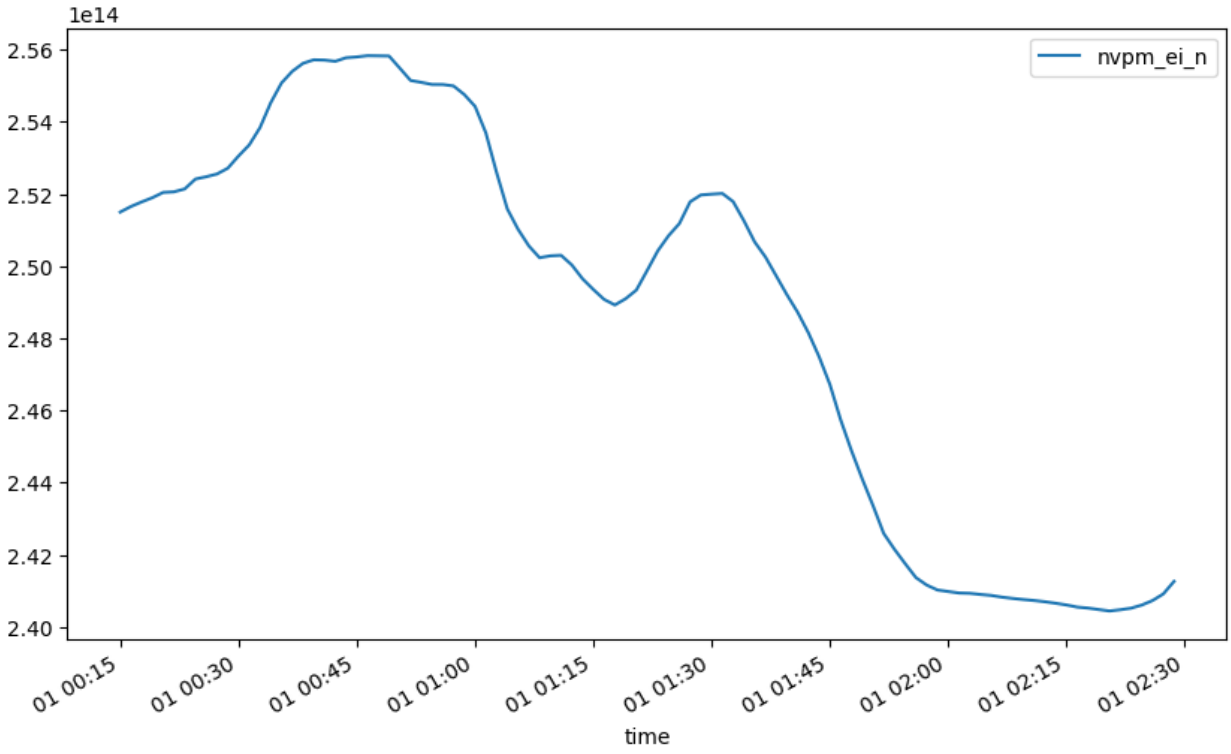
(continued from previous page)

```
df = pd.DataFrame()
df["longitude"] = np.linspace(-40, -55, 100)
df["latitude"] = np.linspace(38, 45, 100)
df["altitude"] = np.linspace(10900, 10900, 100)
df["time"] = pd.date_range("2022-03-01T00:15:00", "2022-03-01T02:30:00", periods=100)
fl = Flight(data=df, attrs=flight_attrs)
```

[21]:
```
from pycontrails.models.ps_model import PSFlight
```

[22]:
```
cocip = Cocip(
    met=met,
    rad=rad,
    humidity_scaling=ConstantHumidityScaling(rhi_adj=0.99),
    aircraft_performance=PSFlight(),
)

output_flight = cocip.eval(source=fl)
```

[23]:
```
output_flight.dataframe.plot(x="time", y="nvpm_ei_n");
```



6.4.10 Output contrail cirrus optical depth

Note this is only a preliminary implementation and will be changed in the future

The example below uses the contrail cirrus output from 1 flight, but the `df_contrails` input can include contrail cirrus from multiple flights.

To run multiple flights, concatenate `Cocip.contrail` outputs from multiple flights and feed in to `grid_cirrus.<>` methods as `df_contrails`. Unique `flight_id` column will have to be added to the `Cocip.contrail` output before concatenation.

[24]:
```
# demo synthetic flight
flight_attrs = {
    "flight_id": "test",
    "true_airspeed": 226.099920796651,  # true airspeed, m/s
    "thrust": 0.22,  # thrust_setting
    "nvpm_ei_n": 1.897462e15,
    "aircraft_type": "E190",
    "wingspan": 48,
    "n_engine": 2,
}

# Example flight
df = pd.DataFrame()
df["longitude"] = np.linspace(-40, -55, 100)
df["latitude"] = np.linspace(38, 45, 100)
df["altitude"] = np.linspace(10900, 10900, 100)
df["engine_efficiency"] = np.linspace(0.34, 0.35, 100)  # ope
df["fuel_flow"] = np.linspace(2.1, 2.4, 100)  # kg/s
df["aircraft_mass"] = np.linspace(154445, 154345, 100)  # kg
df["time"] = pd.date_range("2022-03-01T00:15:00", "2022-03-01T02:30:00", periods=100)
fl = Flight(data=df, attrs=flight_attrs)
```

[25]:
```
# run model
cocip = Cocip(
    met=met,
    rad=rad,
    process_emissions=False,
    humidity_scaling=ConstantHumidityScaling(rhi_adj=0.99),
)
output_flight = cocip.eval(source=fl)
```

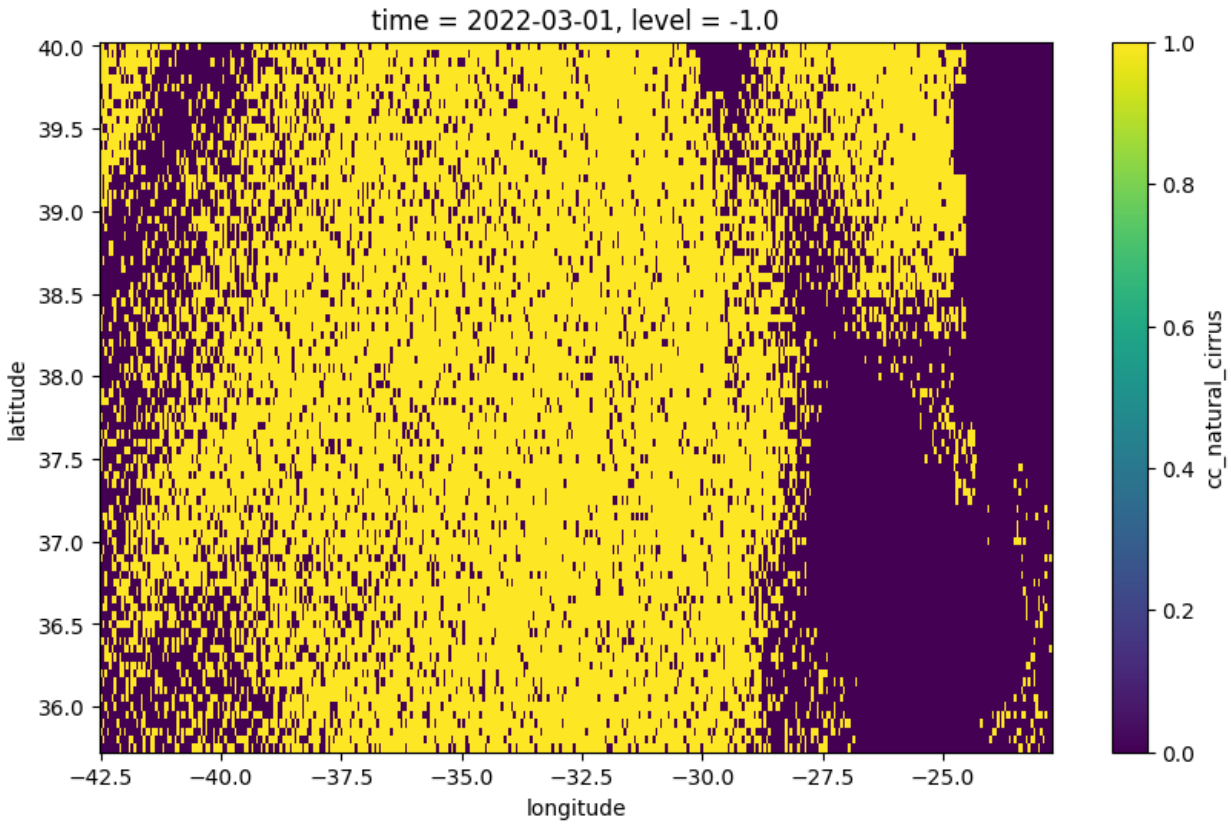
[26]:
```
# get dataframe of contrail waypoints
df_contrails = cocip.contrail
df_contrails["flight_id"] = cocip.source.attrs["flight_id"]
```

[27]:
```
w = df_contrails["longitude"].min()
e = df_contrails["longitude"].max()
s = df_contrails["latitude"].min()
n = df_contrails["latitude"].max()
bbox = (w, s, e, n)

met_bbox = MetDataset(met.data.isel(time=[0])).downselect(bbox)
```

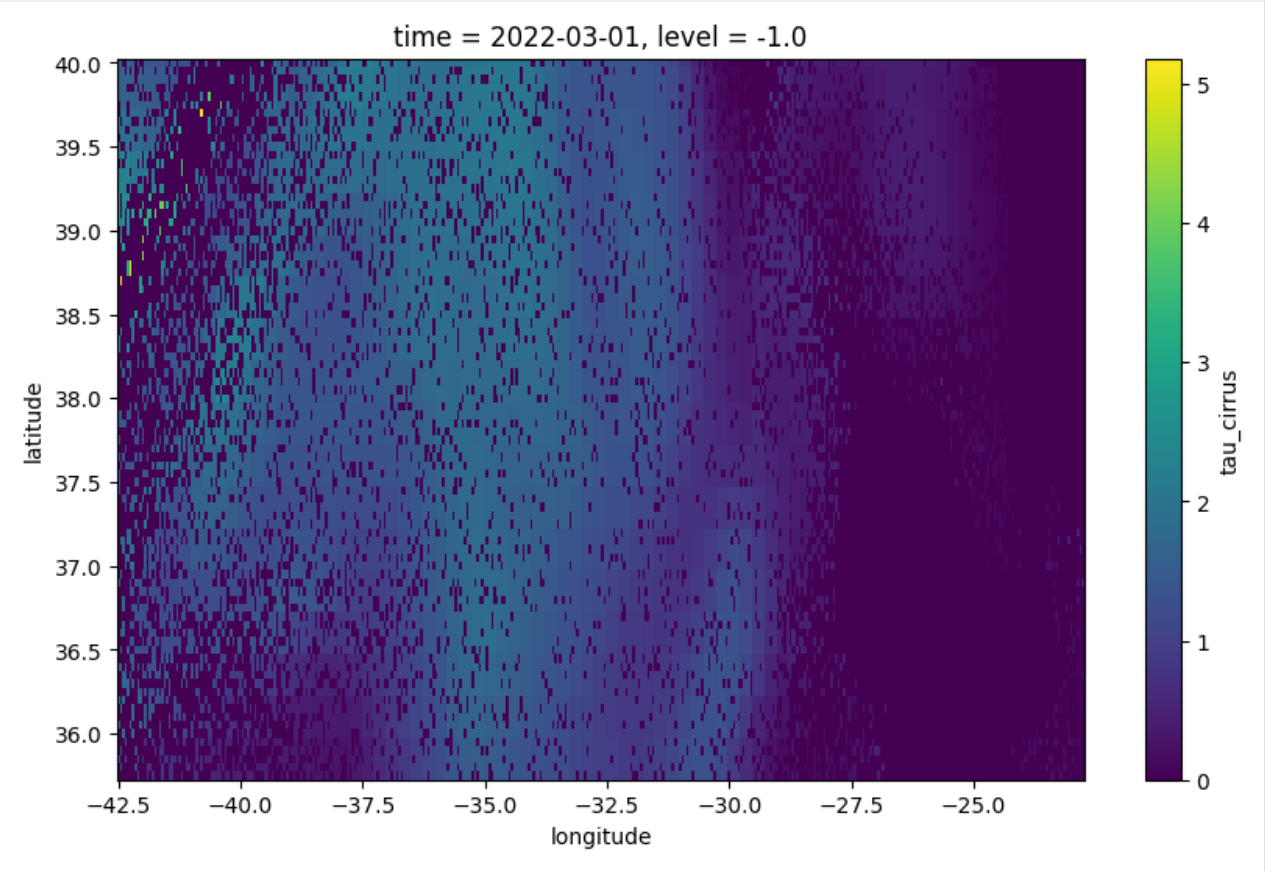
[28]:
```
ds_cirrus = natural_cirrus_properties_to_hi_res_grid(met_bbox)
```

```
/Users/marcshapiro/computing/contrailcirrus/pycontrails/pycontrails/core/met.py:755:␣
↪UserWarning: Overwriting data in keys `['tau_cirrus']`. Use `.update(...)` to suppress␣
↪warning.
  warnings.warn(
```

[29]:
```
ds_cirrus["cc_natural_cirrus"].data.squeeze().plot(x="longitude", y="latitude");
```



[30]:
```
ds_cirrus["tau_cirrus"].data.squeeze().plot(x="longitude", y="latitude");
```



6.5 Dry Advection

The `Cocip` model is designed to simulate the evolution and radiative forcing of a contrail over its lifetime. The model is intricate, relying on a number of meteorological variables and parameters governing the cloud microphysics.

If we are only interested in the evolution of the exhaust plume, we can use the `DryAdvection` model instead. This model is much simpler, and only requires the initial plume properties and a handful of meteorological variables (wind and temperature). This notebook demonstrates how to use the `DryAdvection` model and compares the results to the `Cocip` model.

[1]:
```
import pathlib

import matplotlib.pyplot as plt
import pandas as pd

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.cocip import Cocip
from pycontrails.models.dry_advection import DryAdvection
from pycontrails.models.humidity_scaling import ConstantHumidityScaling
from pycontrails.models.ps_model import PSFlight
from pycontrails.physics import units

plt.rcParams["figure.figsize"] = (10, 6)
```

6.5.1 Load data

Following patterns already used in the in the *CoCiP notebook*.

Meteorological data

[2]:
```
time_bounds = ("2022-03-01 00:00:00", "2022-03-01 12:00:00")
pressure_levels = (300, 250, 200)

era5pl = ERA5(
    time=time_bounds,
    variables=Cocip.met_variables + Cocip.optional_met_variables,
    pressure_levels=pressure_levels,
)
era5sl = ERA5(time=time_bounds, variables=Cocip.rad_variables)

met = era5pl.open_metdataset()
rad = era5sl.open_metdataset()
```

6.5.2 Example flight

[3]:
```
df = pd.read_csv("data/flight.csv")

# Clip at 33,000 ft to hit more ISSR
df["altitude"] = df["altitude"].clip(upper=units.ft_to_m(33000))

# Create a flight instance
fl = Flight(df, aircraft_type="A320", flight_id="example")
```

6.5.3 Run `DryAdvection` and visualize plume evolution

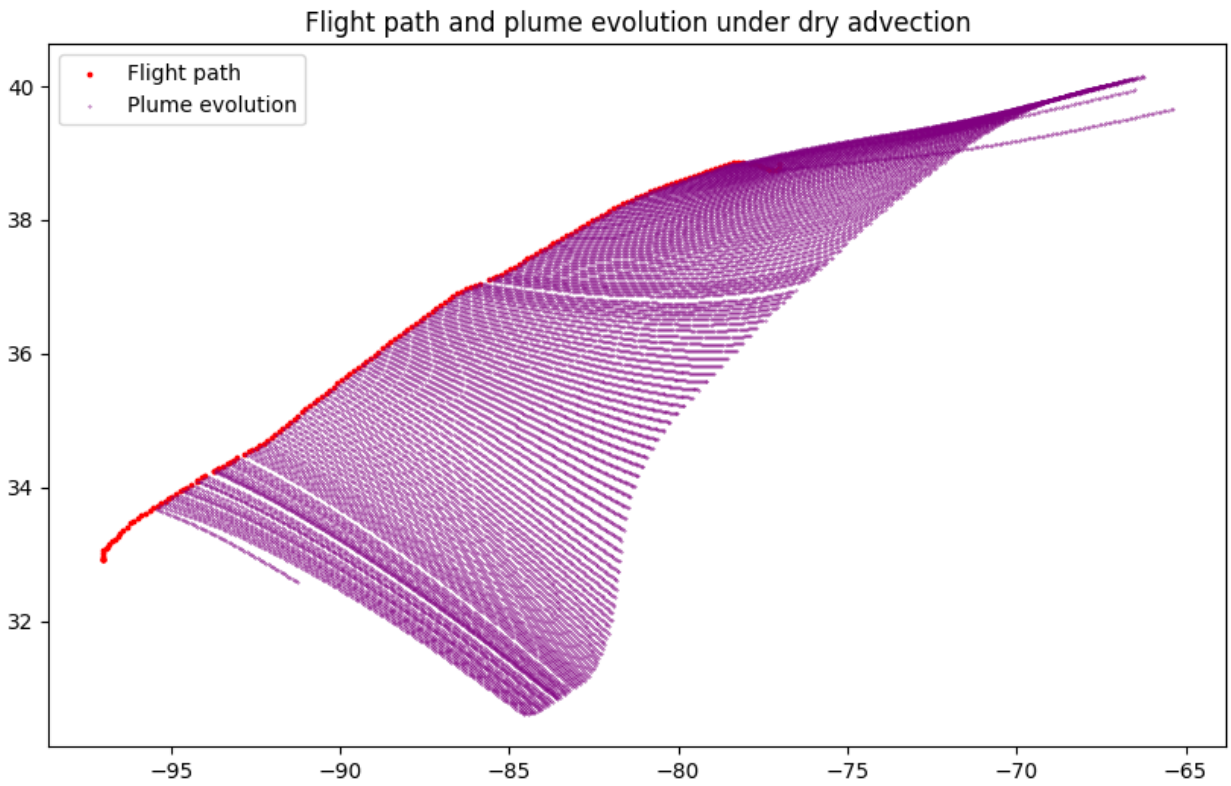
[4]:
```
dt_integration = pd.Timedelta(minutes=2)
max_age = pd.Timedelta(hours=6)

params = {
    "dt_integration": dt_integration,
    "max_age": max_age,
    "depth": 50.0,  # initial plume depth, [m]
    "width": 40.0,  # initial plume width, [m]
}
dry_adv = DryAdvection(met, params)
dry_adv_df = dry_adv.eval(fl).dataframe
```

[5]:
```
ax = plt.axes()

ax.scatter(fl["longitude"], fl["latitude"], s=3, color="red", label="Flight path")
ax.scatter(
    dry_adv_df["longitude"], dry_adv_df["latitude"], s=0.1, color="purple", label="Plume␣
```

(continues on next page)

(continued from previous page)

```
↪evolution"
)
ax.legend()
ax.set_title("Flight path and plume evolution under dry advection");
```



6.5.4 Run `Cocip` and visualize evolution

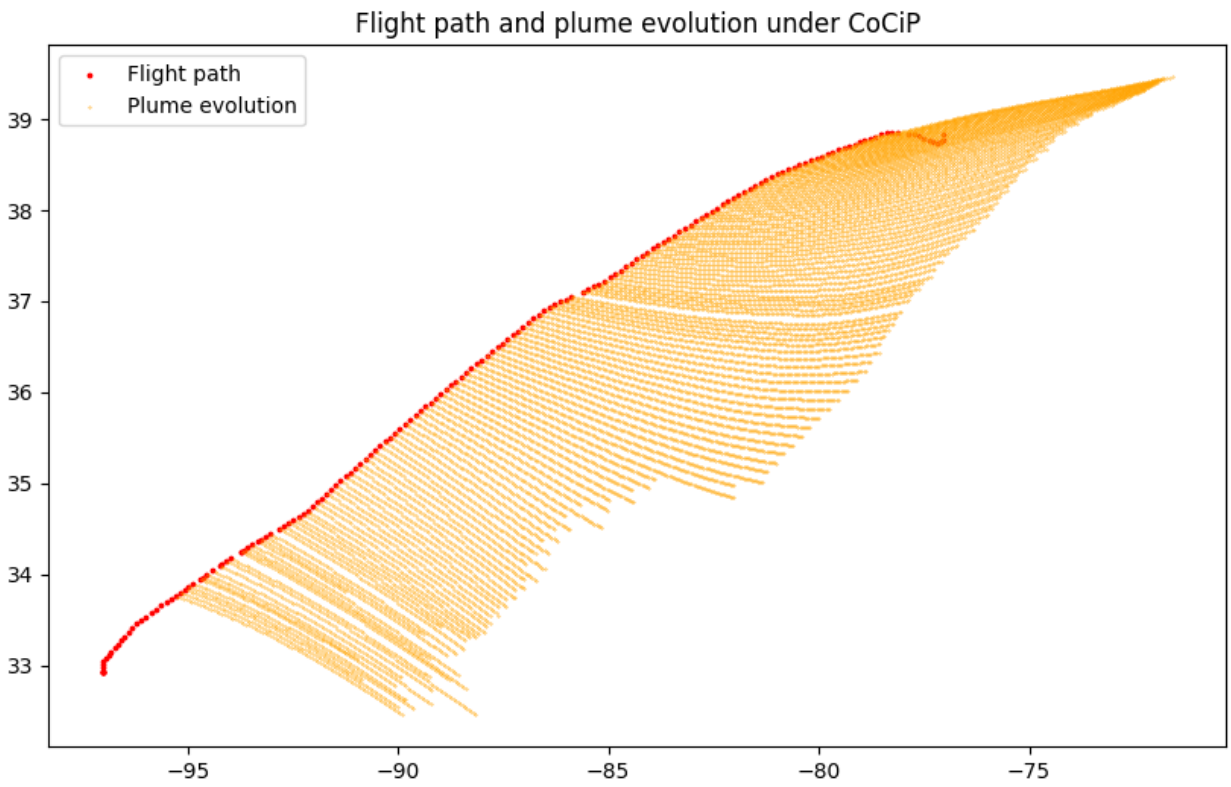
[6]:
```
params = {
    "dt_integration": dt_integration,
    "max_age": max_age,
    "humidity_scaling": ConstantHumidityScaling(rhi_adj=0.9),
    "aircraft_performance": PSFlight(),
}
cocip = Cocip(met, rad, params)
cocip.eval(fl)
cocip_df = cocip.contrail
```

[7]:
```
ax = plt.axes()

ax.scatter(fl["longitude"], fl["latitude"], s=3, color="red", label="Flight path")
ax.scatter(
    cocip_df["longitude"], cocip_df["latitude"], s=0.1, color="orange", label="Plume␣
↪evolution"
)
```

(continues on next page)

(continued from previous page)

```
ax.legend()
ax.set_title("Flight path and plume evolution under CoCiP");
```



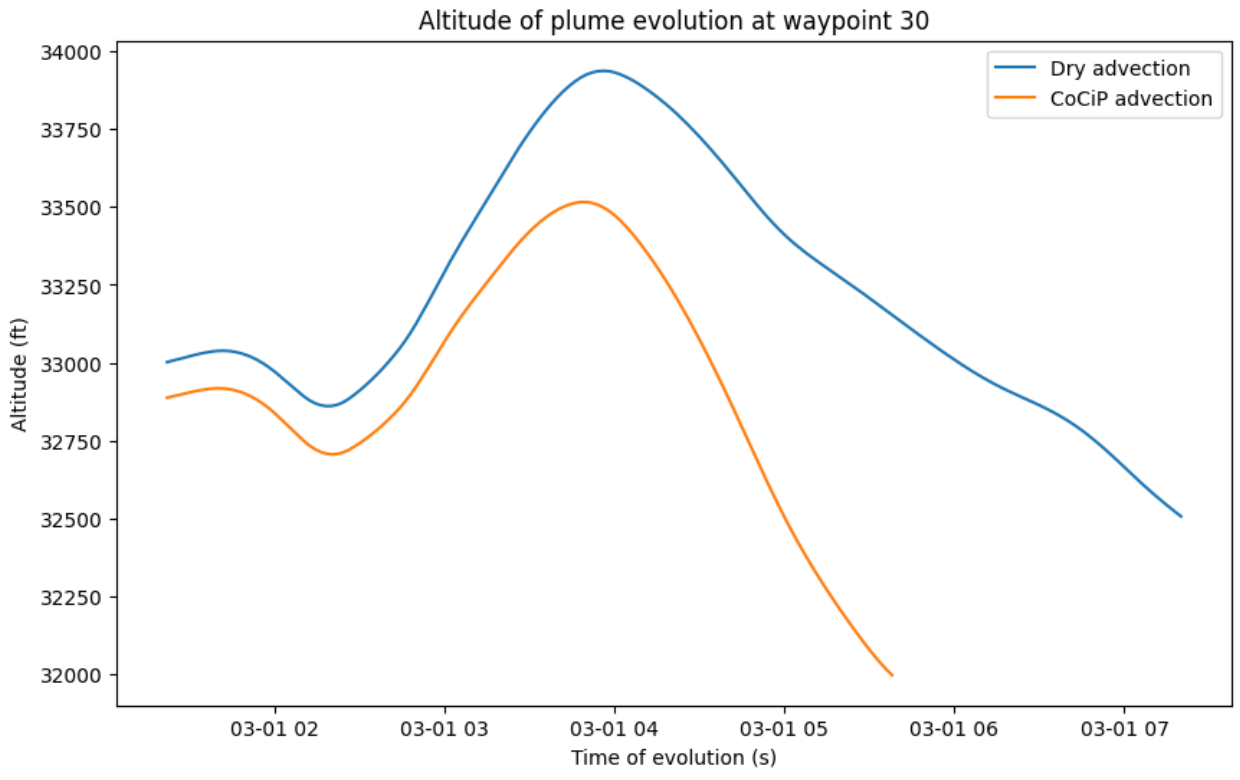
6.5.5 Comparison with CoCiP

	Dry Advection	CoCiP
Horizontal advection	Yes	Yes
Vertical advection	Yes	Yes
Assumes elliptical plume	Yes	Yes
Evolves plume geometry (width, depth)	Yes	Yes
Possible to run in pointwise mode (no width nor depth)	Yes	No
Considers ice growth and sedimentation	No	Yes
Calculates radiative forcing	No	Yes
Models cloud microphysics	No	Yes
Calculates end of life condition at each time step	No	Yes
Requires aircraft performance model	No	Yes
Allows arbitrary initial ellipse dimensions (width, depth)	Yes	No

Visualize advected altitude

While the horizontal position of the advected waypoints are relatively similar in both simulations, the altitude of the evolved plume can substantially differ. In CoCiP, the initial contrail is calculated based on an assumed downwash simulation. This typically lowers the contrail depth several hundred meters below the altitude of the flight path. In addition, because CoCiP simulates ice crystal growth and sedimentation, the contrail altitude typically decreases as ice crystals grow and sediment. The dry advection model does not consider sedimentation, and the plume altitude is initially assumed to be the same as the flight altitude. The following figure shows the altitude of the advected waypoints in both simulations.

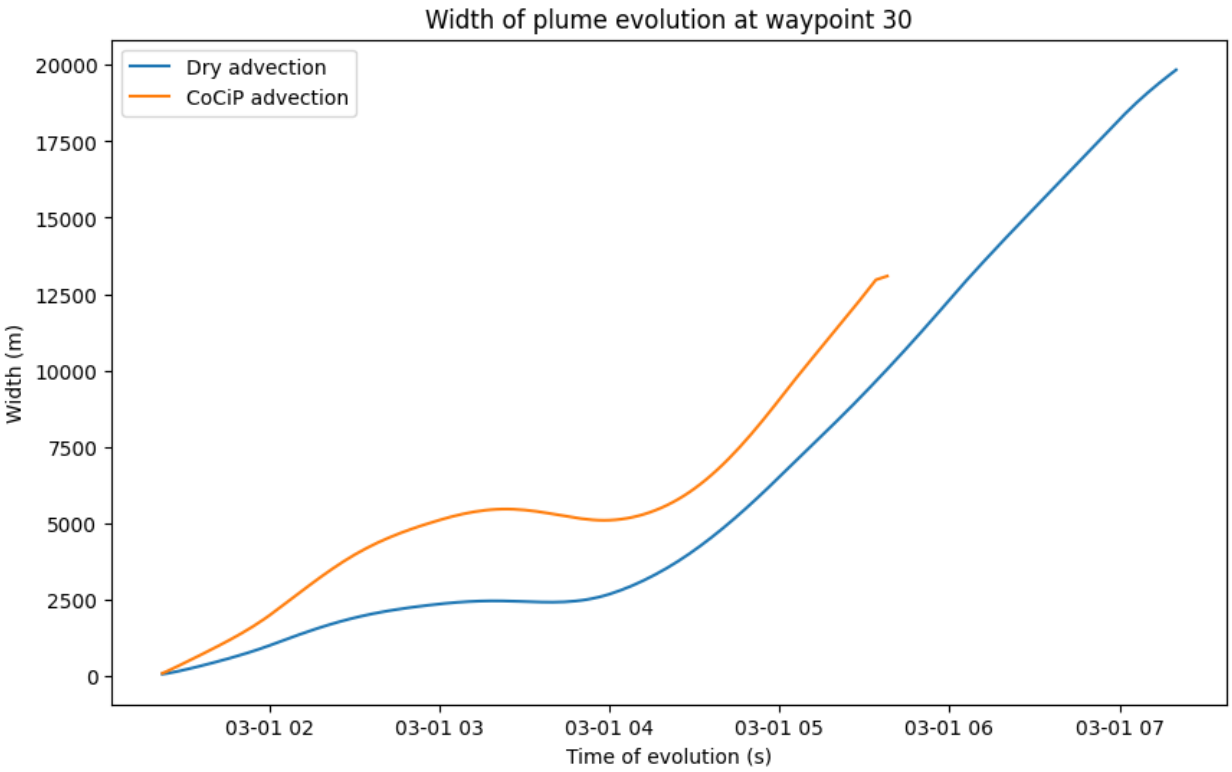
[8]:
```
# Consider a sample waypoint
waypoint = 30
tmp1 = dry_adv_df.query(f"waypoint == {waypoint}")
tmp2 = cocip_df.query(f"waypoint == {waypoint}")

ax = plt.axes()
ax.plot(tmp1["time"], units.pl_to_ft(tmp1["level"]), label="Dry advection")
ax.plot(tmp2["time"], units.pl_to_ft(tmp2["level"]), label="CoCiP advection")
ax.set_xlabel("Time of evolution (s)")
ax.set_ylabel("Altitude (ft)")
ax.set_title(f"Altitude of plume evolution at waypoint {waypoint}")
ax.legend();
```



Visualize advected width

Both CoCiP and dry advection assume the geometry of the plume has an elliptical cross section. The width and depth of this plume evolve according to wind shear effects. Presently, the wind shear enhancement in `Cocip` is not implemented in `DryAdvection`. This explains why the `Cocip`-derived plume is typically wider than the `DryAdvection`-derived plume.

[9]:
```
ax = plt.axes()
ax.plot(tmp1["time"], tmp1["width"], label="Dry advection")
ax.plot(tmp2["time"], tmp2["width"], label="CoCiP advection")
ax.set_xlabel("Time of evolution (s)")
ax.set_ylabel("Width (m)")
ax.set_title(f"Width of plume evolution at waypoint {waypoint}")
ax.legend();
```



6.5.6 Animate the two evolutions

[10]:
```
from matplotlib.animation import FuncAnimation, PillowWriter
from IPython.display import Image
```

[11]:
```
fig, ax = plt.subplots()
ax.set_xlim(dry_adv_df["longitude"].min() - 2, dry_adv_df["longitude"].max() + 2)
ax.set_ylim(dry_adv_df["latitude"].min() - 2, dry_adv_df["latitude"].max() + 2)

scat1 = ax.scatter([], [], s=2, color="red", label="Flight path")
scat2 = ax.scatter([], [], color="orange", label="CoCiP advection")
```

(continues on next page)

(continued from previous page)

```
scat3 = ax.scatter([], [], color="purple", label="Dry advection")
ax.legend(loc="upper left")

source_frames = fl.dataframe.groupby(fl.dataframe["time"].dt.ceil(dt_integration))
dry_adv_frames = dry_adv_df.groupby(dry_adv_df["time"].dt.ceil(dt_integration))
cocip_frames = cocip_df.groupby(cocip_df["time"].dt.ceil(dt_integration))


times = cocip_frames.indices


def animate(time):
    ax.set_title(time)

    try:
        group = source_frames.get_group(time)
    except KeyError:
        offsets = [[None, None]]
    else:
        offsets = group[["longitude", "latitude"]]
    scat1.set_offsets(offsets)

    group = cocip_frames.get_group(time)
    offsets = group[["longitude", "latitude"]]
    width = 1e-3 * group["width"]
    scat2.set_offsets(offsets)
    scat2.set_sizes(width)

    group = dry_adv_frames.get_group(time)
    offsets = group[["longitude", "latitude"]]
    width = 1e-3 * group["width"]
    scat3.set_offsets(offsets)
    scat3.set_sizes(width)

    return scat1, scat2


plt.close()
ani = FuncAnimation(fig, animate, frames=times)
filename = pathlib.Path("evo.gif")
ani.save(filename, dpi=300, writer=PillowWriter(fps=10))

# Show the gif
display(Image(data=filename.read_bytes(), format="png"))

# Cleanup
filename.unlink()
```

notebooks_advection_20_0.png

6.6 ACCF

Interface to the ACCF (algorithmic climate change functions) from the DLR library climaccf.

6.6.1 References

- Dietmüller, Simone. "Dlr-Pa/Climaccf: Dataset Update for GMDD." Zenodo, September 13, 2022. https://doi.org/10.5281/zenodo.7074582.
- Dietmüller, Simone, Sigrun Matthes, Katrin Dahlmann, Hiroshi Yamashita, Abolfazl Simorgh, Manuel Soler, Florian Linke, et al. "A Python Library for Computing Individual and Merged Non-CO2 Algorithmic Climate Change Functions: CLIMaCCF V1.0." Preprint. Atmospheric sciences, October 17, 2022. https://doi.org/10.5194/gmd-2022-203.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.accf import ACCF
from pycontrails.physics import units
```

6.6.2 Set Domain

time / altitude

[2]:
```
time_bounds = ("2022-01-01 12:00:00", "2022-01-01 15:00:00")
pressure_levels = [200, 225, 250]
```

6.6.3 Download meteorology data from ECMWF

This demo uses ERA5 via the Copernicus Data Store (CDS) for met data. This requires account with Copernicus Data Portal and local `~/.cdsapirc` file with credentials.

[3]:
```
# pressure level data
era5_pl = ERA5(
    time=time_bounds,
    variables=[
        "air_temperature",
        "specific_humidity",
        "potential_vorticity",
        "geopotential",
        "relative_humidity",
        "northward_wind",
        "eastward_wind",
    ],
    pressure_levels=pressure_levels,
)

# single level data (radiation)
era5_sl = ERA5(
    time=time_bounds,
    variables=["surface_solar_downward_radiation", "top_net_thermal_radiation"],
)

pl = era5_pl.open_metdataset()
sl = era5_sl.open_metdataset()
```

6.6.4 Create example flight

[4]:
```
# Note: ACCF output is in K/kg of fuel so you will need to provide fuel burn of the␣
↪flight to get impact in K
n = 10000
longitude = np.linspace(45, 58, n) + np.linspace(0, 1, n)
latitude = np.linspace(45, 54, n) - np.linspace(0, 1, n)
altitude = units.pl_to_m(225) * np.ones(n)

start = np.datetime64(time_bounds[0])
end = start + np.timedelta64(90, "m")  # 90 minute flight
time = pd.date_range(start, end, periods=n)
fl = Flight(
    longitude=longitude,
```

(continues on next page)

(continued from previous page)

```
    latitude=latitude,
    altitude=altitude,
    time=time,
    aircraft_type="B737",
    flight_id=17,
)
fl["fuel_flow"] = np.linspace(2.1, 2.4, n)  # kg/s
```

[5]:
```
fl
```

[5]:
```
Flight [5 keys x 10000 length, 3 attributes]
    Keys: longitude, latitude, time, altitude, fuel_flow
    Attributes:
    time                [2022-01-01 12:00:00, 2022-01-01 13:30:00]
    longitude           [45.0, 59.0]
    latitude            [45.0, 53.0]
    altitude            [11037.0, 11037.0]
    aircraft_type       B737
    flight_id           17
    crs                 EPSG:4326
```

6.6.5 Initialize ACCF model and evaluate on flight

[7]:
```
ac = ACCF(met=pl, surface=sl)
```

[8]:
```
# Evaluate ACCFs over this flight
fl = ac.eval(fl)
```

```
selected forecast step:  6.0 HR
 UserWarning: For this configuration formation of persistent contrails is possible, if␣
↪temperatures are low enough (below 235K) and relative humidity (with respect to ice)␣
↪is above or at 90.0%. However keep in mind that the threshold value for the relative␣
↪humidity varies with the used forecast model and its resolution. In order to choose␣
↪the appropriate threshold value, you should read the details given in Section 5.1 of␣
↪the connected publication of Dietmueller et al. 2022
```

6.6.6 Explore the output

Raw output:

[9]:
```
fl
```

[9]:
```
Flight [27 keys x 10000 length, 7 attributes]
    Keys: longitude, latitude, time, altitude, fuel_flow, ..., Fin
    Attributes:
    time                [2022-01-01 12:00:00, 2022-01-01 13:30:00]
    longitude           [45.0, 59.0]
    latitude            [45.0, 53.0]
```

(continues on next page)

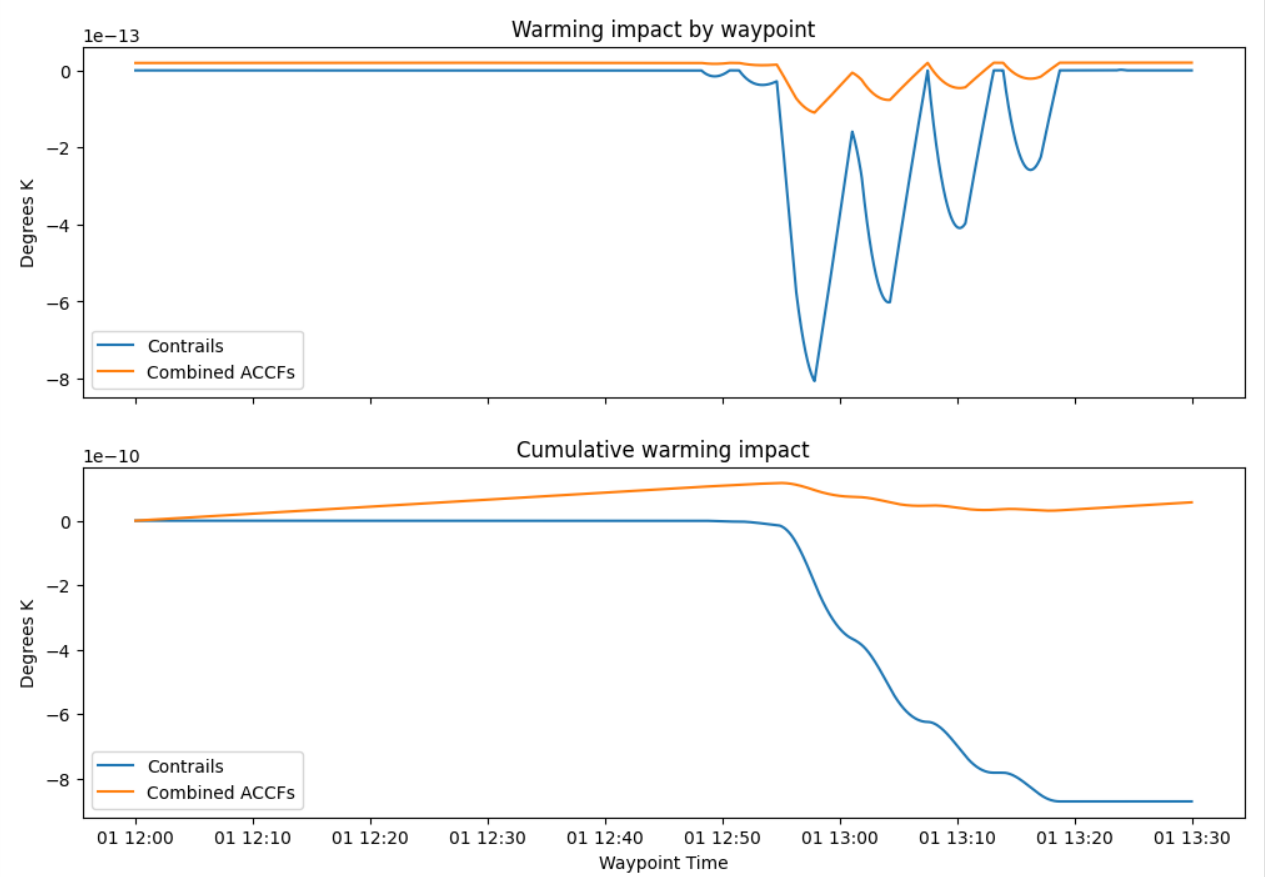
(continued from previous page)

```
    altitude            [11037.0, 11037.0]
    aircraft_type       B737
    flight_id           17
    crs                 EPSG:4326
    met_source_provider ECMWF
    met_source_dataset  ERA5
    met_source_product  reanalysis
    pycontrails_version 0.47.3.dev86
```

Now convert fuel flow to total fuel burned by waypoint, and get the impact of each flight waypoint and plot

[11]:
```
# Waypoint duration in seconds
dt_sec = fl.segment_duration()

# kg fuel per contrail
fuel_burn = fl["fuel_flow"] * dt_sec

# Get impacts in degrees K per waypoint
warming_contrails = fuel_burn * fl["aCCF_Cont"]
warming_merged = fuel_burn * fl["aCCF_merged"]
```

[12]:
```
f, (ax0, ax1) = plt.subplots(2, 1, sharex=True, figsize=(12, 8))
ax0.plot(fl["time"], warming_contrails, label="Contrails")
ax0.plot(fl["time"], warming_merged, label="Combined ACCFs")
ax0.set_ylabel("Degrees K")
ax0.set_title("Warming impact by waypoint")
ax0.legend()

ax1.plot(fl["time"], np.cumsum(warming_contrails), label="Contrails")
ax1.plot(fl["time"], np.cumsum(warming_merged), label="Combined ACCFs")
ax1.legend()
ax1.set_xlabel("Waypoint Time")
ax1.set_ylabel("Degrees K")
ax1.set_title("Cumulative warming impact");
```



6.6.7 Grid ACCF with Parameters

[13]:
```
ac = ACCF(pl, sl, {"emission_scenario": "future_scenario", "accf_v": "V1.1"})
ds = ac.eval()  # This will evaluate over the met grid
```

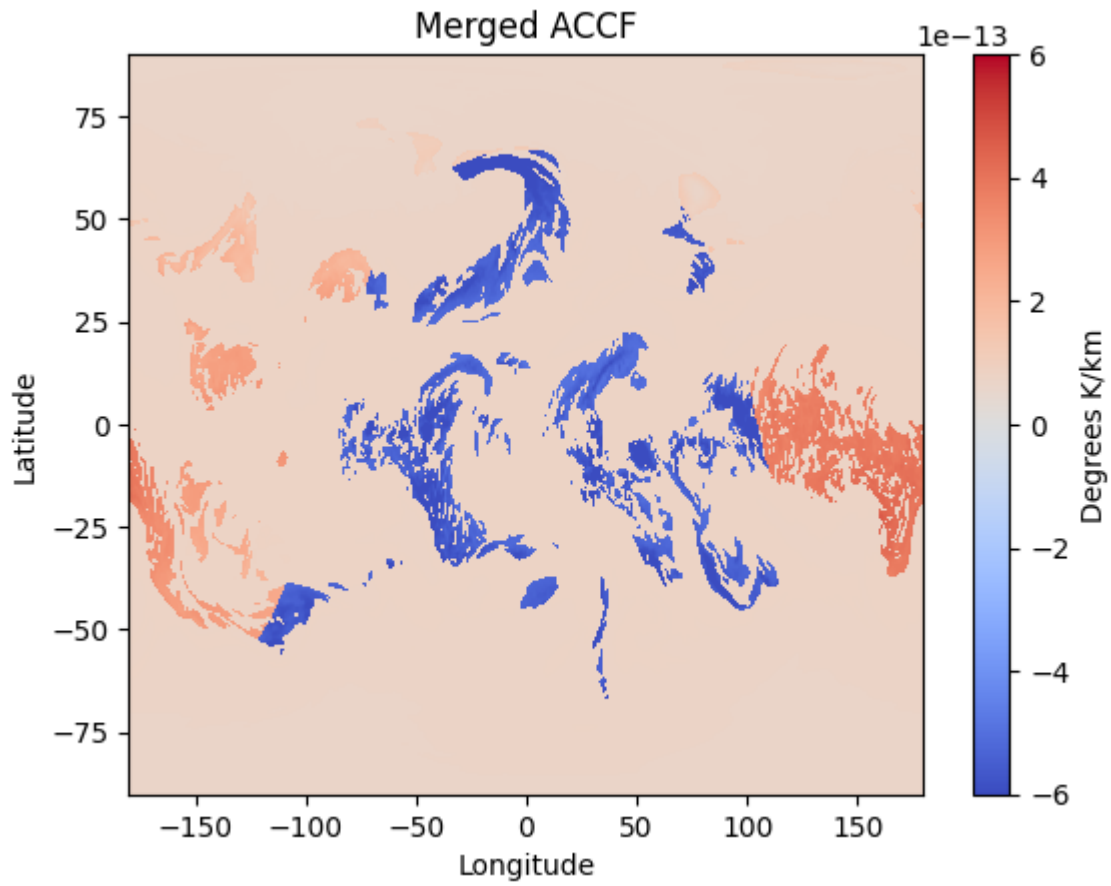
```
selected forecast step:  6.0 HR
 UserWarning: For this configuration formation of persistent contrails is possible, if␣
↪temperatures are low enough (below 235K) and relative humidity (with respect to ice)␣
↪is above or at 90.0%. However keep in mind that the threshold value for the relative␣
↪humidity varies with the used forecast model and its resolution. In order to choose␣
↪the appropriate threshold value, you should read the details given in Section 5.1 of␣
↪the connected publication of Dietmueller et al. 2022
```

[14]:
```
fig, ax = plt.subplots()
p = ax.pcolor(
    ds["longitude"].data,
    ds["latitude"].data,
    ds["aCCF_merged"].data[:, :, 0, 0].T,
    cmap="coolwarm",
    vmin=-6e-13,
    vmax=6e-13,
```

(continues on next page)

(continued from previous page)

```
)
ax.set_xlabel("Longitude")
ax.set_ylabel("Latitude")
ax.set_title("Merged ACCF")
cbar = plt.colorbar(p)
cbar.set_label("Degrees K/km");
```



CHAPTER

SEVEN

OBSERVATIONS

See *Running Notebooks* to interact with these notebooks

7.1 Load GOES imagery

The ability to load and visualize GOES-16 was added in pycontrails 0.47.2. This notebook demonstrates basic usage.

[1]:
```
import cartopy.crs as ccrs
import cartopy.feature as cfeature
import matplotlib.pyplot as plt

from pycontrails.datalib import goes
```

7.1.1 Download GOES data

By default, any GOES data will be cached on disk. Set `cachestore=None` to disable caching when defining the `GOES` object.

We download data for channels 1, 2, and 3. These are needed for creating a true color image.

[2]:
```
handler = goes.GOES(region="conus", channels=("C01", "C02", "C03"))

# Download the data
da = handler.get("2023-02-05T18:00:00")
da
```

[2]:
```
<xarray.DataArray 'CMI' (band_id: 3, y: 3000, x: 5000)>
dask.array<getitem, shape=(3, 3000, 5000), dtype=float32, chunksize=(1, 3000, 5000),␣
↪chunktype=numpy.ndarray>
Coordinates:
    t                (band_id) datetime64[ns] 2023-02-05T18:02:35.739931008 ...
  * y                (y) float64 0.1282 0.1282 0.1282 ... 0.04428 0.04425
  * x                (x) float64 -0.1013 -0.1013 -0.1013 ... 0.0386 0.03863
    y_image          float32 0.08624
    x_image          float32 -0.03136
    band_wavelength  (band_id) float32 dask.array<chunksize=(1,), meta=np.ndarray>
  * band_id          (band_id) int32 1 2 3
Attributes:
    long_name:                  ABI L2+ Cloud and Moisture Imagery reflectanc...
    standard_name:              toa_lambertian_equivalent_albedo_multiplied_b...
```

(continues on next page)

(continued from previous page)

```
    sensor_band_bit_depth:      10
    valid_range:                [   0 4095]
    units:                      1
    resolution:                 y: 0.000028 rad x: 0.000028 rad
    grid_mapping:               goes_imager_projection
    cell_methods:               t: point area: point
    ancillary_variables:        DQF
    goes_imager_projection:     {'long_name': 'GOES-R ABI fixed grid projecti...
    geospatial_lat_lon_extent:  {'long_name': 'geospatial latitude and longit...
```

7.1.2 True color

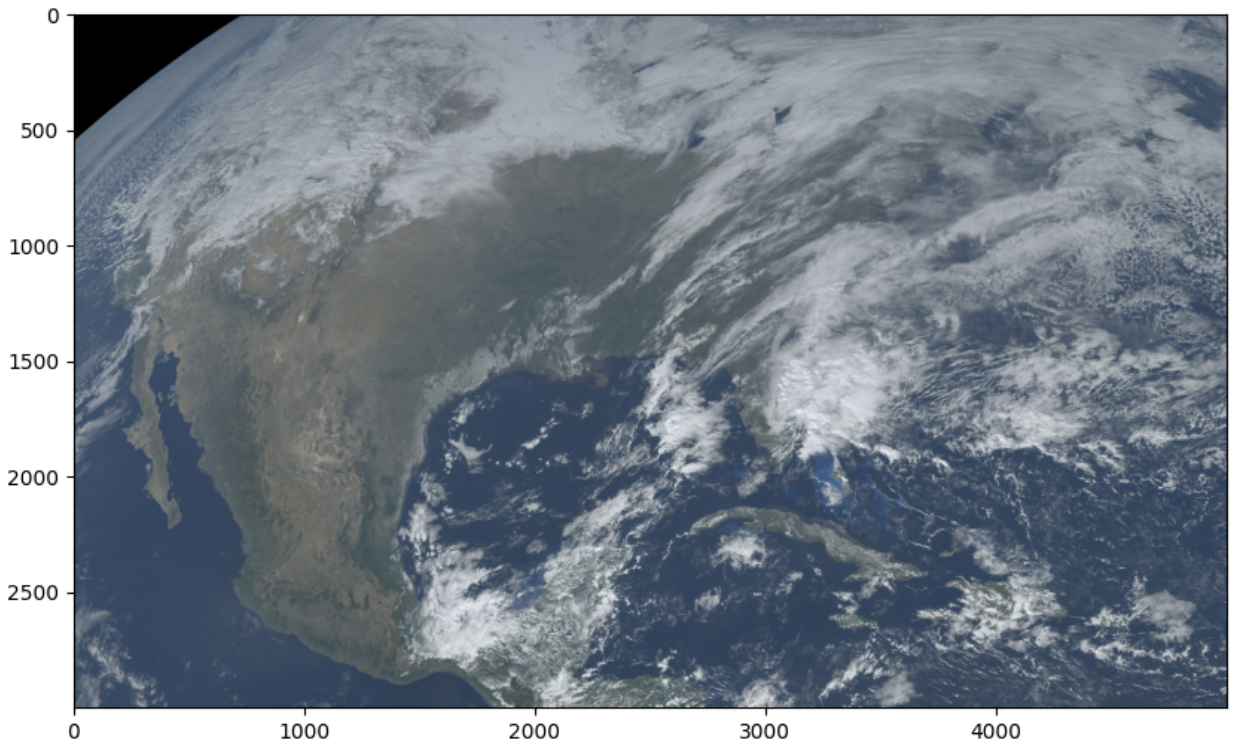
First we plot a true color image of the CONUS region.

[3]:
```
# Extract artifacts for plotting
rgb, src_crs, src_extent = goes.extract_goes_visualization(da, color_scheme="true")
```

RGB array

The rgb array is a 3D array with shape `(3, y, x)`. The first dimension contains the red, green, and blue channels, respectively. The second and third dimensions contain the y and x coordinates. The plot below shows the earth as seen by the satellite.

[4]:
```
fig, ax = plt.subplots(figsize=(10, 10))
ax.imshow(rgb, origin="upper");
```



Projection artifacts

The `src_crs` is a `cartopy.crs.Projection` describing the coordinate reference system of the satellite data. The src_extent is a tuple containing the extent of the data in the source coordinate reference system. These can be used to plot the data on a map.

[5]:
```
src_crs
```

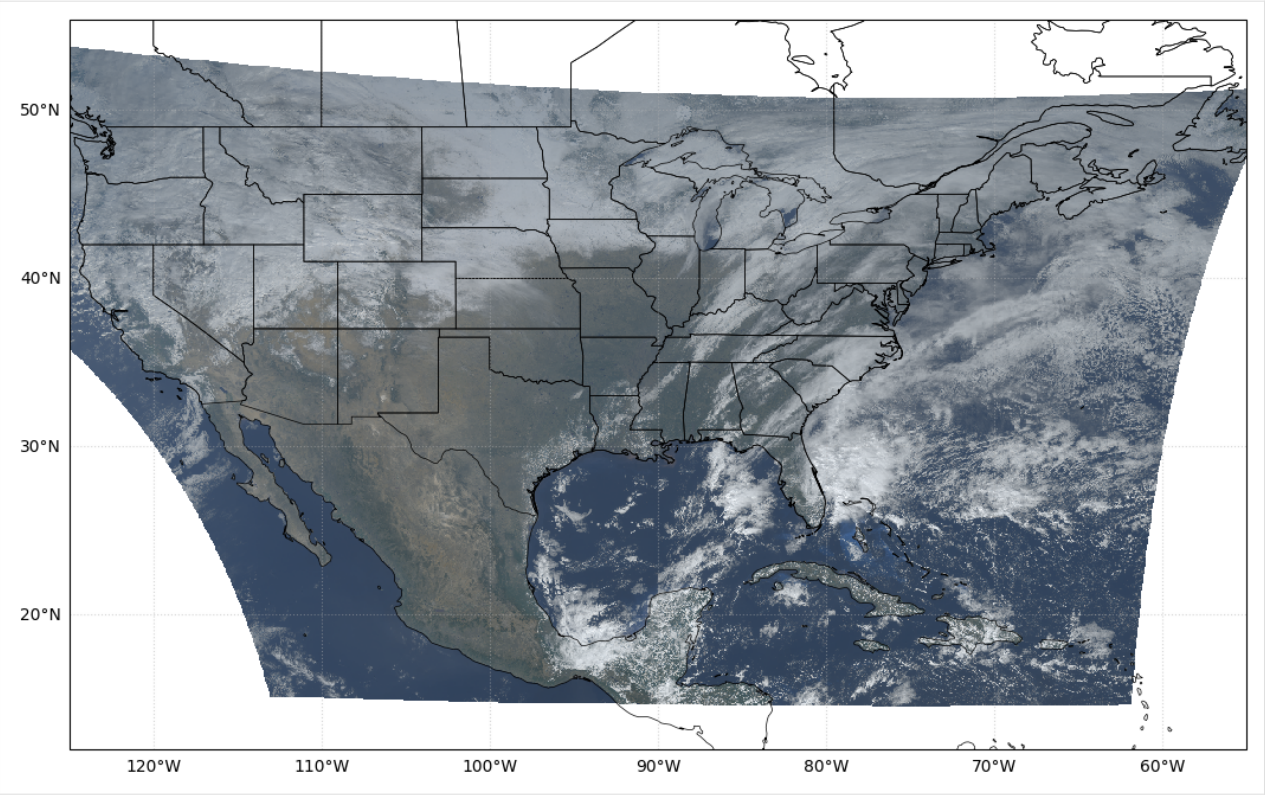
[5]:
```
<Projected CRS: +proj=geos +ellps=WGS84 +lon_0=-75.0 +lat_0=0.0 +h ...>
Name: unknown
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- undefined
Coordinate Operation:
- name: unknown
- method: Geostationary Satellite (Sweep X)
Datum: Unknown based on WGS 84 ellipsoid
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Visualize

We reproject the image to a Plate Carree projection and add state boundaries and coastlines to the plot.

[6]:
```
dst_crs = ccrs.PlateCarree()
```

[7]:
```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(projection=dst_crs, extent=(-125, -55, 12, 50))
ax.coastlines(resolution="50m", color="black", linewidth=0.5)

# add state boundaries to plot
ax.add_feature(cfeature.STATES, edgecolor="black", linewidth=0.5)

ax.imshow(rgb, extent=src_extent, transform=src_crs, origin="upper", interpolation="none
↪")

# Set the x and y ticks to use latitude and longitude labels
gl = ax.gridlines(draw_labels=True, alpha=0.5, linestyle=":")
gl.top_labels = False
gl.right_labels = False
```



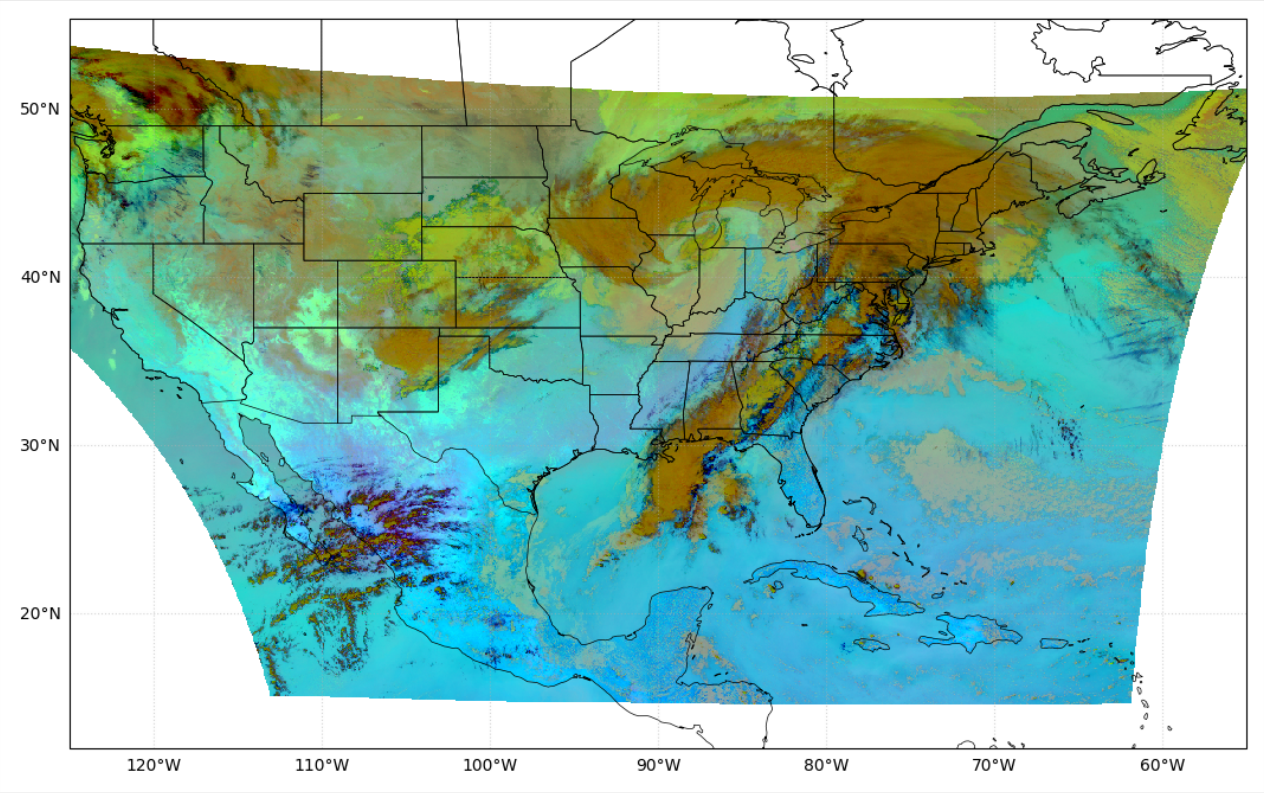
7.1.3 Ash color scheme

The ash color scheme was originally developed to visualize volcanic ash. It is also useful for visualizing contrails.

We download data for channels 11, 14, and 15 to create an ash color image.

[8]:
```
handler = goes.GOES(region="conus", channels=("C11", "C14", "C15"))

# Download the data
da = handler.get("2023-02-09T18:00:00")

rgb, src_crs, src_extent = goes.extract_goes_visualization(da, color_scheme="ash")
```

[9]:
```
fig = plt.figure(figsize=(16, 8))
ax = fig.add_subplot(projection=dst_crs, extent=(-125, -55, 12, 50))
ax.coastlines(resolution="50m", color="black", linewidth=0.5)

# add state boundaries to plot
ax.add_feature(cfeature.STATES, edgecolor="black", linewidth=0.5)

ax.imshow(rgb, extent=src_extent, transform=src_crs, origin="upper", interpolation="none
↪")

# Set the x and y ticks to use latitude and longitude labels
gl = ax.gridlines(draw_labels=True, alpha=0.5, linestyle=":")
gl.top_labels = False
gl.right_labels = False
```


50°N
40°N
30°N
20°N
120°W
110°W
100°W
90°W
80°W
70°W
60°W

CHAPTER

EIGHT

UTILITIES

See *Running Notebooks* to interact with these notebooks

8.1 Set up a Cache store

The `DiskCacheStore` and `GCPCacheStore` module enables connections to backend storage sources for caching large files.

By default, `pycontrails` uses the `DiskCacheStore` which stores files in the user cache directory or the current hard drive.

[1]:
```
from pycontrails import DiskCacheStore, GCPCacheStore
```

8.1.1 Disk Cache

Store and retrieve files from the local hard drive.

[2]:
```
disk = DiskCacheStore()
```

[3]:
```
disk.exists("test/cache/path.nc")
```

[3]:
```
False
```

[4]:
```
# Total cache size, in MB
disk.size
```

[4]:
```
23709.123237
```

8.1.2 GCP Cache

Requires `[gcp]` optional dependencies:

```
$ pip install pycontrails[gcp]
```

Store and retrieve files from Google Cloud Storage.

[5]:
```
gcp = GCPCacheStore(bucket="contrails-301217-unit-test", cache_dir="test/objects")
```

[6]:
```
gcp.bucket
```
[6]:
```
'contrails-301217-unit-test'
```

[7]:
```
gcp.path("path.nc")
```
[7]:
```
'test/objects/path.nc'
```

[8]:
```
gcp.gs_path("met/ecmwf/some-path.nc")
```
[8]:
```
'gs://contrails-301217-unit-test/test/objects/met/ecmwf/some-path.nc'
```

[ ]:
```
gcp.exists("some-path.nc")
```
```
False
```

Put objects

By default, GCP cache is read only. Pass `read_only=False` to constructor to allow writing

[9]:
```
gcp = GCPCacheStore(bucket="contrails-301217-unit-test", cache_dir="test/objects", read_
↪only=False)
```

[ ]:
```
gcp.put("cache.ipynb", "cache.ipynb")
```
```
'cache.ipynb'
```

CHAPTER

NINE

TUTORIALS

See *Running Notebooks* to interact with these notebooks

9.1 Run CoCiP over a flight

This tutorial walks through an example of running the Contrail Cirrus Predicition (CoCiP) model evaluation along a flight trajectory.

9.1.1 References

- Schumann, U. "A Contrail Cirrus Prediction Model." Geoscientific Model Development 5, no. 3 (May 3, 2012): 543–80. https://doi.org/10.5194/gmd-5-543-2012.
- Schumann, U., B. Mayer, K. Graf, and H. Mannstein. "A Parametric Radiative Forcing Model for Contrail Cirrus." Journal of Applied Meteorology and Climatology 51, no. 7 (July 2012): 1391–1406. https://doi.org/10.1175/JAMC-D-11-0242.1.
- Teoh, Roger, Ulrich Schumann, Arnab Majumdar, and Marc E. J. Stettler. "Mitigating the Climate Forcing of Aircraft Contrails by Small-Scale Diversions and Technology Adoption." Environmental Science & Technology 54, no. 5 (March 3, 2020): 2941–50. https://doi.org/10.1021/acs.est.9b05608.
- Teoh, Roger, Ulrich Schumann, Edward Gryspeerdt, Marc Shapiro, Jarlath Molloy, George Koudis, Christiane Voigt, and Marc Stettler. "Aviation Contrail Climate Effects in the North Atlantic from 2016–2021." Atmospheric Chemistry and Physics Discussions, March 30, 2022, 1–27. https://doi.org/10.5194/acp-2022-169.

[1]:
```
import pandas as pd
import xarray as xr

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.cocip import Cocip
from pycontrails.models.humidity_scaling import ConstantHumidityScaling
from pycontrails.physics import units
```

9.1.2 Load Flight

Load flight trajectory from dataset prepared by Roger Teoh in https://doi.org/10.5194/acp-2022-169

[2]:
```
# load flight waypoints
df_flight = pd.read_csv("data/flight-cocip.csv")
df_flight.head()
```

[2]:
```
   Longitude (degrees)  Latitude (degrees)  Altitude (feet)    UTC time  \
0               -10.070              55.185            36000  1546651185
1               -10.273              55.222            36000  1546651245
2               -10.476              55.258            36000  1546651305
3               -10.680              55.295            36000  1546651365
4               -10.883              55.331            36000  1546651425

   True airspeed (m s-1)  Mach Number  Aircraft mass (kg)  \
0                230.858        0.791          236479.000
1                230.682        0.790          236379.755
2                230.563        0.789          236280.355
3                230.501        0.789          236180.791
4                230.476        0.789          236081.128

   Fuel mass flow rate (kg s-1)  Overall propulsion efficiency  \
0                         1.654                            0.4
1                         1.657                            0.4
2                         1.659                            0.4
3                         1.661                            0.4
4                         1.662                            0.4

   nvPM number emissions index (kg-1) ICAO Aircraft Type  Wingspan (m)
0                   1500000000000000                A359         64.75
1                   1500000000000000                A359         64.75
2                   1500000000000000                A359         64.75
3                   1500000000000000                A359         64.75
4                   1500000000000000                A359         64.75
```

[3]:
```
# constant properties along the length of the flight
attrs = {
    "flight_id": "fid",
    "aircraft_type": df_flight["ICAO Aircraft Type"].values[0],
    "wingspan": df_flight["Wingspan (m)"].values[0],
}
```

Process the flight into a format expected by `pycontrails`. See pycontrails.Flight for interface details.

[4]:
```
# convert UTC timestamp to np.datetime64
df_flight["time"] = pd.to_datetime(df_flight["UTC time"], origin="unix", unit="s")

# set altitude in m
df_flight["altitude"] = units.ft_to_m(df_flight["Altitude (feet)"])

# rename a few columns for compatibility with `Flight` requirements
df_flight = df_flight.rename(
    columns={
```

(continues on next page)

(continued from previous page)

```
        "Longitude (degrees)": "longitude",
        "Latitude (degrees)": "latitude",
        "True airspeed (m s-1)": "true_airspeed",
        "Mach Number": "mach_number",
        "Aircraft mass (kg)": "aircraft_mass",
        "Fuel mass flow rate (kg s-1)": "fuel_flow",
        "Overall propulsion efficiency": "engine_efficiency",
        "nvPM number emissions index (kg-1)": "nvpm_ei_n",
    }
)

# clean up a few columns before building Flight class
df_flight = df_flight.drop(
    columns=["ICAO Aircraft Type", "Wingspan (m)", "UTC time", "Altitude (feet)"]
)

fl = Flight(data=df_flight, attrs=attrs)
fl
```

[4]:
```
Flight [10 keys x 162 length, 4 attributes]
    Keys: longitude, latitude, true_airspeed, mach_number, aircraft_mass, ..., time
    Attributes:
    time                [2019-01-05 01:19:45, 2019-01-05 04:00:21]
    longitude           [-50.0, -10.07]
    latitude            [55.185, 61.089]
    altitude            [10972.8, 10972.8]
    flight_id           fid
    aircraft_type       A359
    wingspan            64.75
    crs                 EPSG:4326
```

9.1.3 Load meteorology from ECMWF

[5]:
```
# get met domain from Flight
time = (
    pd.to_datetime(fl["time"][0]).floor("H"),
    pd.to_datetime(fl["time"][-1]).ceil("H") + pd.Timedelta("10H"),
)

# select pressure levels
pressure_levels = [
    400,
    350,
    300,
    250,
    225,
    200,
    175,
    150,
]
```

[6]:
```
# downloads met data from CDS
era5pl = ERA5(time=time, variables=Cocip.met_variables, pressure_levels=pressure_levels)
era5sl = ERA5(
    time=time,
    variables=Cocip.rad_variables,
)
```

[7]:
```
# create `MetDataset` from sources
met = era5pl.open_metdataset()
rad = era5sl.open_metdataset()
```

9.1.4 Set up model

[8]:
```
params = {
    "process_emissions": False,
    "verbose_outputs": True,
    "humidity_scaling": ConstantHumidityScaling(rhi_adj=0.98),
}
cocip = Cocip(met=met, rad=rad, params=params)
```

9.1.5 Run model

[9]:
```
fl_out = cocip.eval(source=fl)
```

```
/Users/marcshapiro/computing/contrailcirrus/pycontrails/pycontrails/models/cocip/cocip.
↪py:2189: UserWarning: At time 2019-01-05T15:30:00.000000, the contrail has no␣
↪intersection with the met data. This is likely due to the contrail being advected␣
↪outside the met domain.
  warnings.warn(
```

9.1.6 Review output

The output flight has the original flight data with many new variables added from the evaluation.

[10]:
```
fl_out
```

[10]:
```
Flight [66 keys x 162 length, 9 attributes]
    Keys: waypoint, longitude, latitude, true_airspeed, mach_number, ..., cocip
    Attributes:
    time                [2019-01-05 01:19:45, 2019-01-05 04:00:21]
    longitude           [-50.0, -10.07]
    latitude            [55.185, 61.089]
    altitude            [10972.8, 10972.8]
    flight_id           fid
    aircraft_type       A359
    wingspan            64.75
    crs                 EPSG:4326
    rhi_adj             0.98
    humidity_scaling_nameconstant_scale
```

(continues on next page)

(continued from previous page)

```
    humidity_scaling_formularhi -> rhi / rhi_adj
    humidity_scaling_rhi_adj0.98
    pycontrails_version 0.49.3.dev50
```

[11]:
```
fl_out.dataframe.columns
```

[11]:
```
Index(['waypoint', 'longitude', 'latitude', 'true_airspeed', 'mach_number',
       'aircraft_mass', 'fuel_flow', 'engine_efficiency', 'nvpm_ei_n',
       'altitude', 'time', 'flight_id', 'level', 'air_pressure',
       'air_temperature', 'specific_humidity', 'u_wind', 'v_wind', 'rhi',
       'tau_cirrus', 'specific_cloud_ice_water_content', 'rho_air', 'sdr',
       'segment_length', 'sin_a', 'cos_a', 'G', 'T_sat_liquid', 'rh',
       'rh_critical_sac', 'sac', 'T_critical_sac', 'width', 'depth', 'rhi_1',
       'air_temperature_1', 'specific_humidity_1', 'altitude_1',
       'persistent_1', 'dT_dz', 'ds_dz', 'dz_max', 'rho_air_1', 'iwc_1',
       'n_ice_per_m_1', 'ef', 'contrail_age', 'sdr_mean', 'sdr_min', 'sdr_max',
       'rsr_mean', 'rsr_min', 'rsr_max', 'olr_mean', 'olr_min', 'olr_max',
       'rf_sw_mean', 'rf_sw_min', 'rf_sw_max', 'rf_lw_mean', 'rf_lw_min',
       'rf_lw_max', 'rf_net_mean', 'rf_net_min', 'rf_net_max', 'cocip'],
      dtype='object')
```

The `cocip` variable describes where persistent contrails form. It can take on values:

- 1: Persistent contrails form
- 0: No persistent contrails form

[12]:
```
fl_out["cocip"]
```

[12]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

The model class contains information about the contrail created:

- `cocip.source` the original input flight
- `cocip.contrail` will be defined as a `pandas` DataFrame if a contrail is created.
- `cocip.contrail_dataset` is the same data but formatted as an `xarray` Dataset.

[13]:
```
cocip.contrail
```

[13]:
```
       waypoint flight_id      formation_time                time  \
index
0            29       fid 2019-01-05 01:48:44 2019-01-05 02:00:00
1            30       fid 2019-01-05 01:49:44 2019-01-05 02:00:00
2            31       fid 2019-01-05 01:50:44 2019-01-05 02:00:00
3            32       fid 2019-01-05 01:51:44 2019-01-05 02:00:00
```

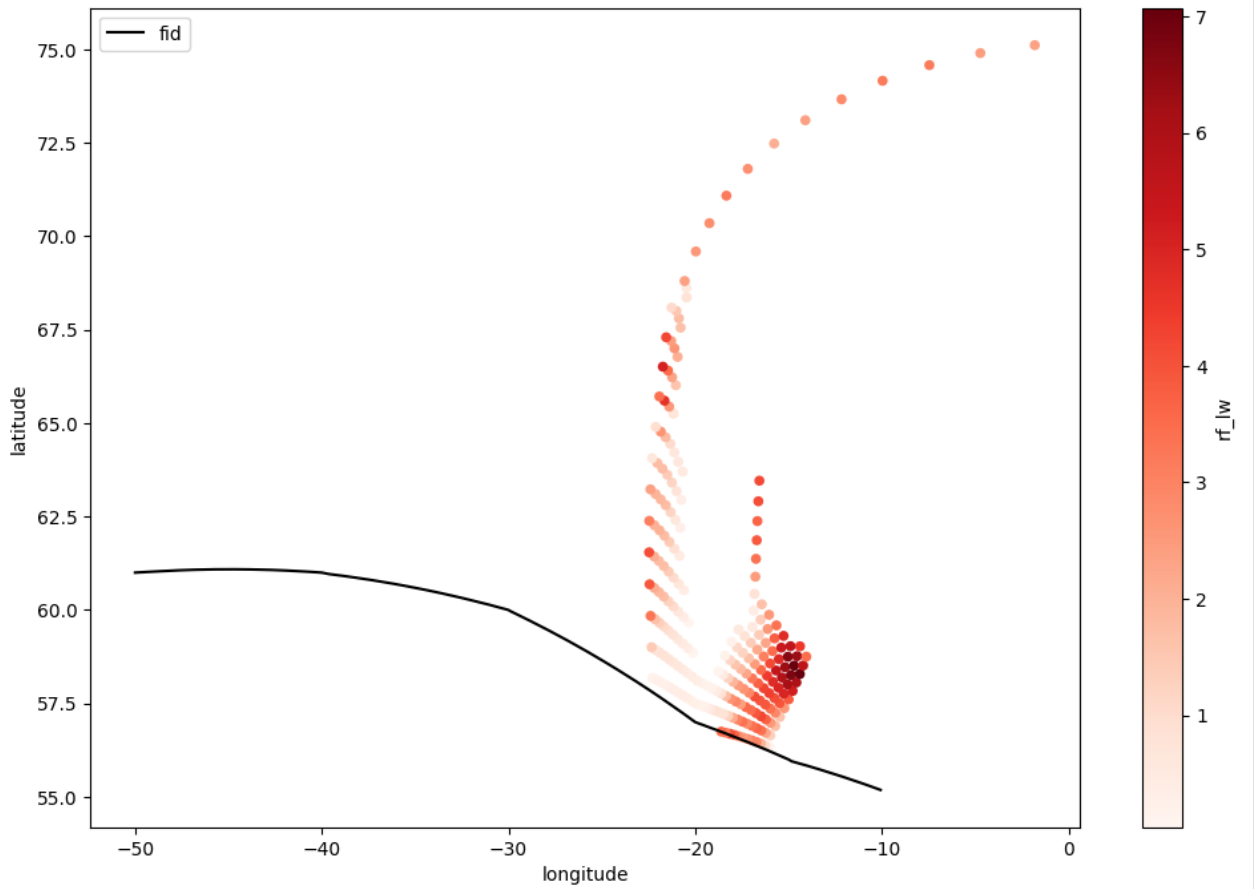
(continues on next page)

(continued from previous page)

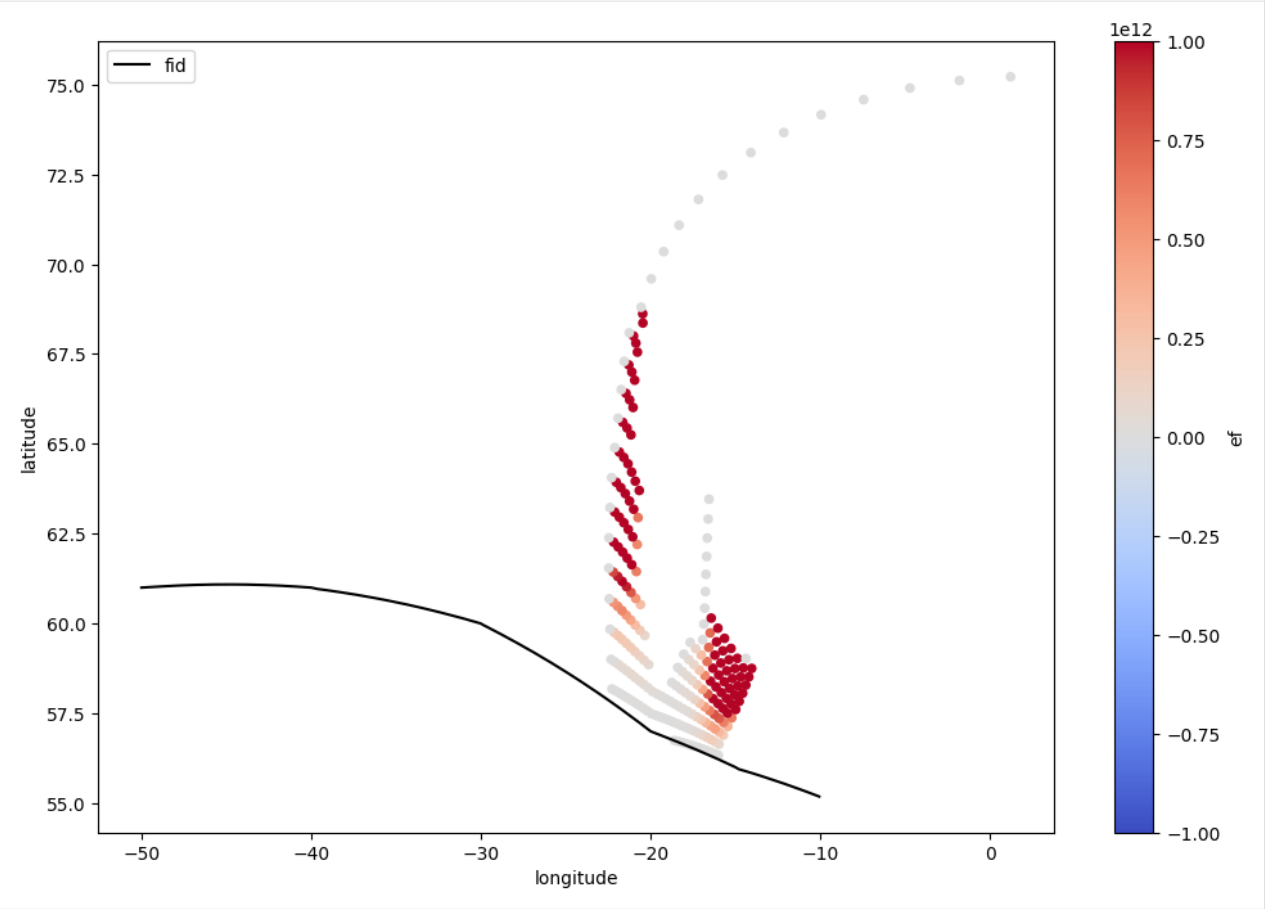
```
4            33       fid 2019-01-05 01:52:44 2019-01-05 02:00:00
...         ...       ...                 ...                 ...
0            55       fid 2019-01-05 02:14:48 2019-01-05 13:00:00
0            55       fid 2019-01-05 02:14:48 2019-01-05 13:30:00
0            55       fid 2019-01-05 02:14:48 2019-01-05 14:00:00
0            55       fid 2019-01-05 02:14:48 2019-01-05 14:30:00
0            55       fid 2019-01-05 02:14:48 2019-01-05 15:00:00

                  age  longitude   latitude      altitude       level  \
index
0     0 days 00:11:16 -16.010456  56.334371  10929.200527  228.855145
1     0 days 00:10:16 -16.243723  56.380602  10926.894766  228.938191
2     0 days 00:09:16 -16.478084  56.425542  10924.667135  229.018447
3     0 days 00:08:16 -16.714148  56.469048  10922.745458  229.087698
4     0 days 00:07:16 -16.951516  56.510025  10921.141514  229.145512
...                ...        ...        ...           ...         ...
0     0 days 00:00:00  -9.957466  74.162396  10938.604760  228.516685
0     0 days 00:00:00  -7.452062  74.581339  10914.404661  229.388471
0     0 days 00:00:00  -4.724895  74.903333  10874.310228  230.838762
0     0 days 00:00:00  -1.807077  75.115034  10854.571581  231.555468
0     0 days 00:00:00   1.202617  75.218719  10856.300880  231.492606

       continuous  ...     dn_dt_agg  dn_dt_turb  rf_sw     rf_lw    rf_net  \
index              ...
0            True  ...  1.567664e-20    0.000025    0.0  0.092565  0.092565
1            True  ...  3.985451e-19    0.000027    0.0  1.043019  1.043019
2            True  ...  6.609394e-19    0.000029    0.0  1.955334  1.955334
3            True  ...  8.881788e-19    0.000030    0.0  2.946739  2.946739
4            True  ...  9.416210e-19    0.000031    0.0  3.009561  3.009561
...           ...  ...           ...         ...    ...       ...       ...
0           False  ...  1.933380e-18    0.000002    0.0  3.184593  3.184593
0           False  ...  1.674621e-18    0.000001    0.0  3.146509  3.146509
0           False  ...  1.218293e-18    0.000001    0.0  2.435764  2.435764
0           False  ...  1.016172e-18    0.000001    0.0  2.323749  2.323749
0           False  ...  1.791175e-18    0.000001    0.0       NaN       NaN

       persistent            ef  timestep  age_hours  dt_integration
index
0            True  2.118821e+09         0   0.187778 0 days 00:11:16
1            True  5.918572e+09         0   0.171111 0 days 00:10:16
2            True  8.176894e+09         0   0.154444 0 days 00:09:16
3            True  9.090643e+09         0   0.137778 0 days 00:08:16
4            True  6.849343e+09         0   0.121111 0 days 00:07:16
...           ...           ...       ...        ...             ...
0            True  0.000000e+00        22   0.000000 0 days 00:30:00
0            True  0.000000e+00        23   0.000000 0 days 00:30:00
0            True  0.000000e+00        24   0.000000 0 days 00:30:00
0            True  0.000000e+00        25   0.000000 0 days 00:30:00
0            True  0.000000e+00        26   0.000000 0 days 00:30:00

[230 rows x 57 columns]
```

We can visualize the contrail on top of the original flight trajectory using pandas plotting capabilities

[14]:
```
ax = cocip.source.dataframe.plot(
    "longitude", "latitude", color="k", label=fl.attrs["flight_id"], figsize=(12, 8)
)
cocip.contrail.plot.scatter("longitude", "latitude", c="rf_lw", cmap="Reds", ax=ax);
```



[15]:
```
ax = cocip.source.dataframe.plot(
    "longitude", "latitude", color="k", label=fl.attrs["flight_id"], figsize=(12, 8)
)
cocip.contrail.plot.scatter(
    "longitude", "latitude", c="ef", cmap="coolwarm", vmin=-1e12, vmax=1e12, ax=ax
);
```



9.2 Run CoCiP with FDR or QAR data

9.2.1 References

- Schumann, U. “A Contrail Cirrus Prediction Model.” Geoscientific Model Development 5, no. 3 (May 3, 2012): 543–80. https://doi.org/10.5194/gmd-5-543-2012.
- EASA: ICAO Aircraft Engine Emissions Databank (07/2021), 2021.
- ICAO: Annex 16: Environmental Protection - Volume II - Aircraft Engine Emissions: https://store.icao.int/en/annex-16-environmental-protection-volume-ii-aircraft-engine-emissions, 2008. Last access: 18 August 2022.

[1]:
```
import numpy as np
import xarray as xr
import pandas as pd
from matplotlib import pyplot as plt
import scipy

from pycontrails import Flight
from pycontrails.datalib.ecmwf import ERA5
from pycontrails.models.cocip import Cocip
from pycontrails.models.humidity_scaling import HistogramMatching
```

(continues on next page)

(continued from previous page)

```
from pycontrails.models.ps_model import PSFlight
from pycontrails.models.emissions import Emissions
```

9.2.2 Download met data

[2]:
```
time_bounds = ("2022-03-01 00:00:00", "2022-03-01 23:00:00")
pressure_levels = (300, 250, 200)  # 30,000 ft to 38,000 ft

era5pl = ERA5(
    time=time_bounds,
    variables=Cocip.met_variables + Cocip.optional_met_variables,
    pressure_levels=pressure_levels,
)
era5sl = ERA5(time=time_bounds, variables=Cocip.rad_variables)

# download data from ERA5 (or open from cache)
met = era5pl.open_metdataset()
rad = era5sl.open_metdataset()
```

9.2.3 Read in data and resample

In this example, the sample data provides TAS, aircraft mass, and fuel flow per engine. Any values provided as input to the CoCiP model will not be overwritten when CoCiP is run, and so we should make sure we are saving the most accurate portions of the FDR data. Typically, thurst values provided by FDRs are noisy and innaccurate and so it is recommended to drop the thrust value if provided and recompute thrust through an aircraft performance model (below).

In this case, fuel flow is provided per engine, and so total fuel flow must be computed by summing the two values together.

Here we are also resampling the FDR data to a one minute sampling period, which is recommended when running CoCiP. Note that the `resample_and_fill` function will interpolate time, position, and altitude, but for the remaining columns, it will simply choose the nearsest value. There are much more accurate ways to resample fuel flow data, but this should generally be sufficient to estimate contrail impacts.

[3]:
```
attrs = {
    "flight_id": "test",
    "aircraft_type": "B77W",
    "engine_uid": "01P21GE217",  # 01P21GE217 -> GE90-115B
    # "n_engine":2 # This shouldn't be needed?
}

df = pd.read_csv("data/flight-fdr.csv")
df["fuel_flow"] = df["fuel_flow_1"] + df["fuel_flow_2"]  # Checked
fl = Flight(df, attrs=attrs)

fl = fl.resample_and_fill(freq="60s", drop=False)
fl
```

[3]:
```
Flight [10 keys x 136 length, 4 attributes]
    Keys: longitude, latitude, altitude, flight_id, true_airspeed, ..., time
    Attributes:
```

(continues on next page)

(continued from previous page)

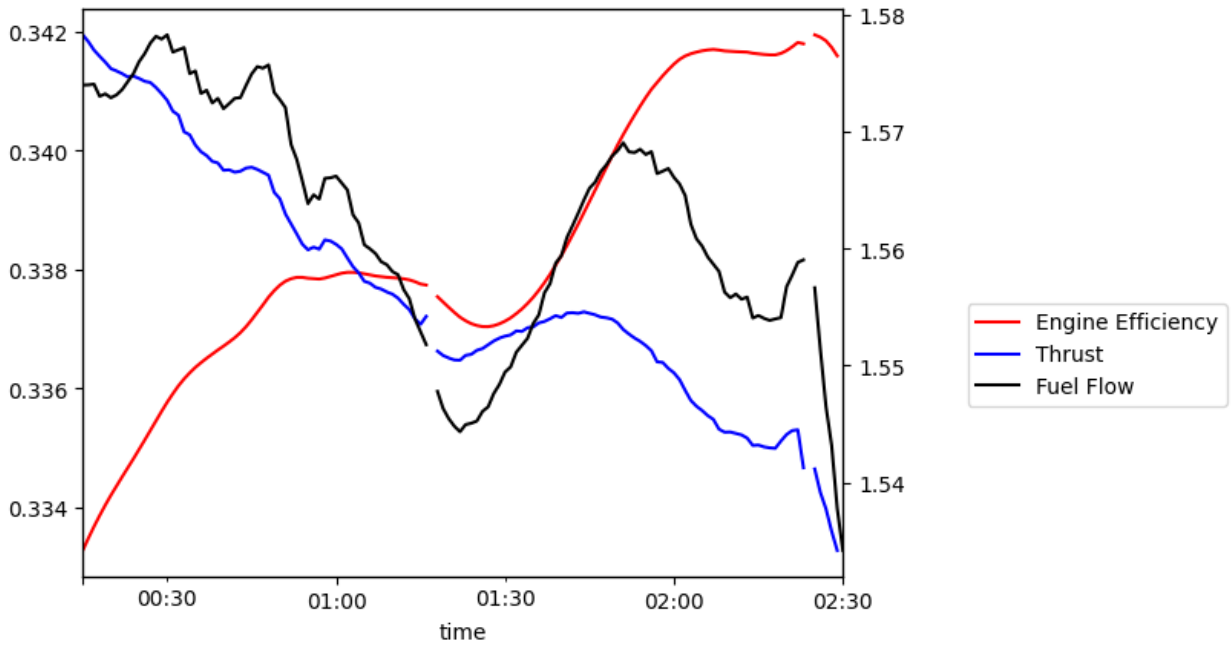
```
    time                [2022-03-01 00:15:00, 2022-03-01 02:30:00]
    longitude           [-39.926, -25.0]
    latitude            [34.0, 39.97]
    altitude            [10900.0, 10900.0]
    flight_id           test
    aircraft_type       B77W
    engine_uid          01P21GE217
    crs                 EPSG:4326
```

9.2.4 (Optional) Run Aircraft Performance Model

The CoCiP module will automatically run an aircraft performance model to compute any missing values needed. In this case, we still need to compute engine efficiency and estimated thrust force. For completeness, we show how this can be computed directly from the an aircraft performance model.

[4]:
```
perf = PSFlight(met=met)
fp = perf.eval(fl)
```

[5]:
```
fig, ax = plt.subplots()
ax2 = ax.twinx()
ax3 = ax2.twinx()
ax2.set_yticks([])
fp.dataframe.plot(ax=ax, x="time", y="engine_efficiency", style="r", legend=False)
fp.dataframe.plot(ax=ax2, x="time", y="thrust", style="b", legend=False)
fp.dataframe.plot(ax=ax3, x="time", y="fuel_flow", style="k", legend=False)
_ = ax3.legend(
    [ax.get_lines()[0], ax2.get_lines()[0], ax3.get_lines()[0]],
    ["Engine Efficiency", "Thrust", "Fuel Flow"],
    bbox_to_anchor=(1.15, 0.5),
)
```



9.2.5 Compute Aircraft Emissions

Given fuel flow data, met data, aircraft, and engine type, we have sufficient information to run the emissions module and compute nvPM estimates needed as input for CoCiP. We have not specified engine type, so in the case, the default engine type will be assumed, which for the `B77W` is the `GE-90 115B`. To specific a different engine type, add the key `engine_uid` to the `attrs` dict when creating the `Flight` object.

Note that the emissions module estimates the aircraft thrust setting by comparing the fuel flow to the maximum fuel flow. This is the perferred way to estimate emissions using the ICAO emissions inventory and is the default behavior of the emissions module even when aircraft thrust is provided in the input.

[6]:
```
emissions = Emissions(met=met, humidity_scaling=HistogramMatching())
fl = emissions.eval(fl)
fl
```

[6]:
```
Flight [31 keys x 136 length, 17 attributes]
    Keys: longitude, latitude, altitude, flight_id, true_airspeed, ..., nvpm_number
    Attributes:
    time                [2022-03-01 00:15:00, 2022-03-01 02:30:00]
    longitude           [-39.926, -25.0]
    latitude            [34.0, 39.97]
    altitude            [10900.0, 10900.0]
    flight_id           test
    aircraft_type       B77W
    engine_uid          01P21GE217
    crs                 EPSG:4326
    n_engine            2
    gaseous_data_source FFM2
    nvpm_data_source    ICAO EDB
    total_co2           39390.76441156896
    total_h2o           15337.3346711712
```

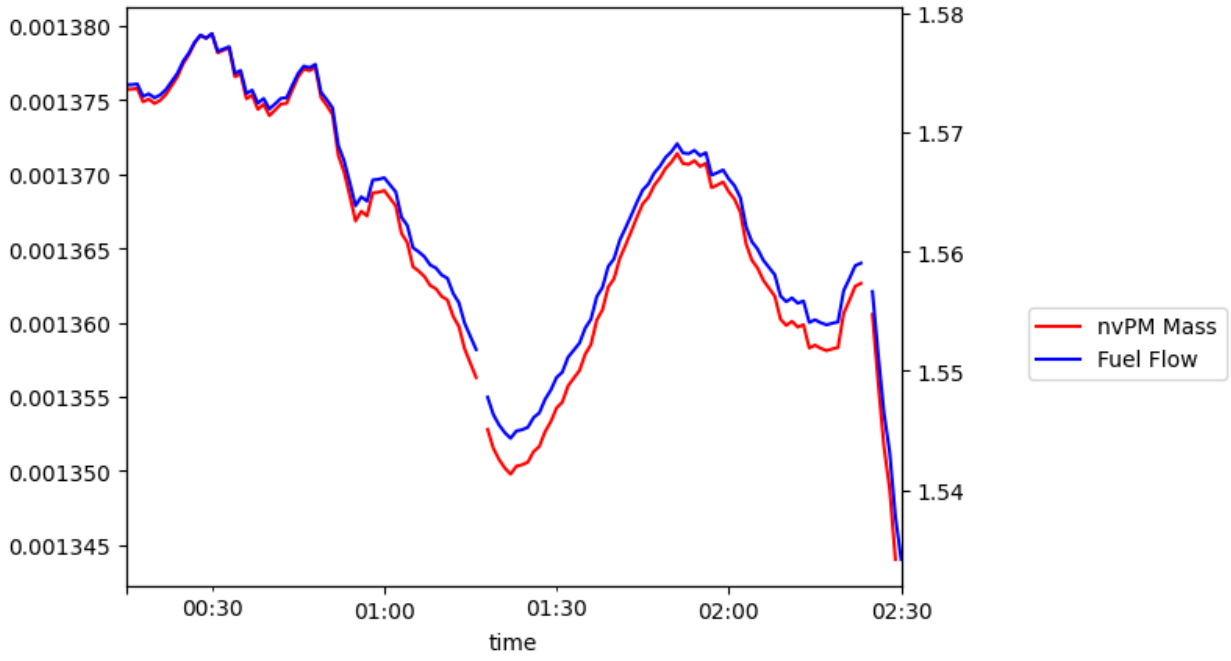
(continues on next page)

(continued from previous page)

```
    total_so2           14.963253337727998
    total_sulphates     0.30537251709648977
    total_oc            0.2493875556288
    total_nox           192.979281298076
    total_co            2.915210297723201
    total_hc            0.6230983842461807
    total_nvpm_mass     0.18164446517898938
    total_nvpm_number   3.54753797881968e+18
```

[7]:
```
fig, ax = plt.subplots()
ax2 = ax.twinx()
fl.dataframe.plot(ax=ax, x="time", y="nvpm_mass", style="r", legend=False)
fl.dataframe.plot(ax=ax2, x="time", y="fuel_flow", style="b", legend=False)
ax2.legend(
    [
        ax.get_lines()[0],
        ax2.get_lines()[0],
    ],
    ["nvPM Mass", "Fuel Flow"],
    bbox_to_anchor=(1.15, 0.5),
)
```

[7]:
```
<matplotlib.legend.Legend at 0x176d15010>
```



9.2.6 Run CoCiP over the flight

In order to predict contrail impact, we still need an estimate of engine efficiency, which is needed to determine if the Schmidt-Appleman Criteria is satisfied. If we provide the `CoCiP` module with an aircraft performance model, in this case the Poll-Schumann model, then this value will be estimated for us.

In the code below, the `CoCiP` module will run the Poll-Schmann model over the flight using the provided aircraft mass in order to estimate the thrust force required for the plane to fly the specified trajectory. This value will then be used along with the provided fuel flow data to estimate engine efficiency. Fuel flow and emissions numbers will not be recomputed here, and the warning from `aircraft_performance.py` can be ignored — three iterations of the performance model are only needed to estimate aircraft mass, which in this case is taken from the FDR data.

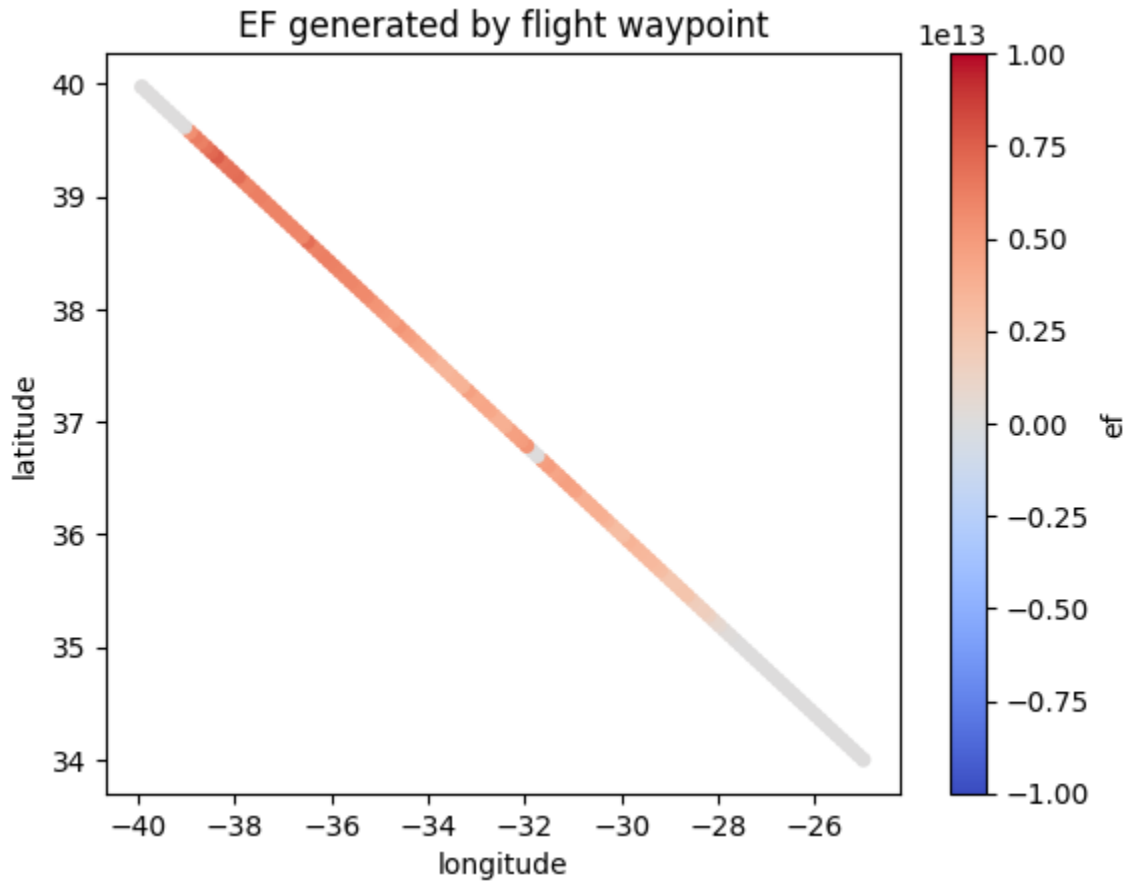
[8]:
```
cocip = Cocip(
    met=met, rad=rad, aircraft_performance=PSFlight(), humidity_
↪scaling=HistogramMatching()
)
fl = cocip.eval(fl)
fl
```

[8]:
```
Flight [67 keys x 136 length, 25 attributes]
    Keys: waypoint, longitude, latitude, altitude, flight_id, ..., cocip
    Attributes:
    time                [2022-03-01 00:15:00, 2022-03-01 02:30:00]
    longitude           [-39.926, -25.0]
    latitude            [34.0, 39.97]
    altitude            [10900.0, 10900.0]
    flight_id           test
    aircraft_type       B77W
    engine_uid          01P21GE217
    crs                 EPSG:4326
    n_engine            2
    gaseous_data_source FFM2
    nvpm_data_source    ICAO EDB
    total_co2           39390.76441156896
    total_h2o           15337.3346711712
    total_so2           14.963253337727998
    total_sulphates     0.30537251709648977
    total_oc            0.2493875556288
    total_nox           192.979281298076
    total_co            2.915210297723201
    total_hc            0.6230983842461807
    total_nvpm_mass     0.18164446517898938
    total_nvpm_number   3.54753797881968e+18
    aircraft_performance_modelPSFlight
    wingspan            64.8
    max_mach            0.89
    max_altitude        13136.880000000001
    total_fuel_burn     12469.37778144
    humidity_scaling_namehistogram_matching
    humidity_scaling_formulaera5_quantiles -> iagos_quantiles
    pycontrails_version 0.49.3.dev50
```

9.2.7 Visualize Contrail Impact

First, we plot cumulative EF by flight waypoint (one minute sample period)

[9]:
```
fl.dataframe.plot.scatter(
    x="longitude",
    y="latitude",
    c="ef",
    cmap="coolwarm",
    vmin=-1e13,
    vmax=1e13,
    title="EF generated by flight waypoint",
);
```



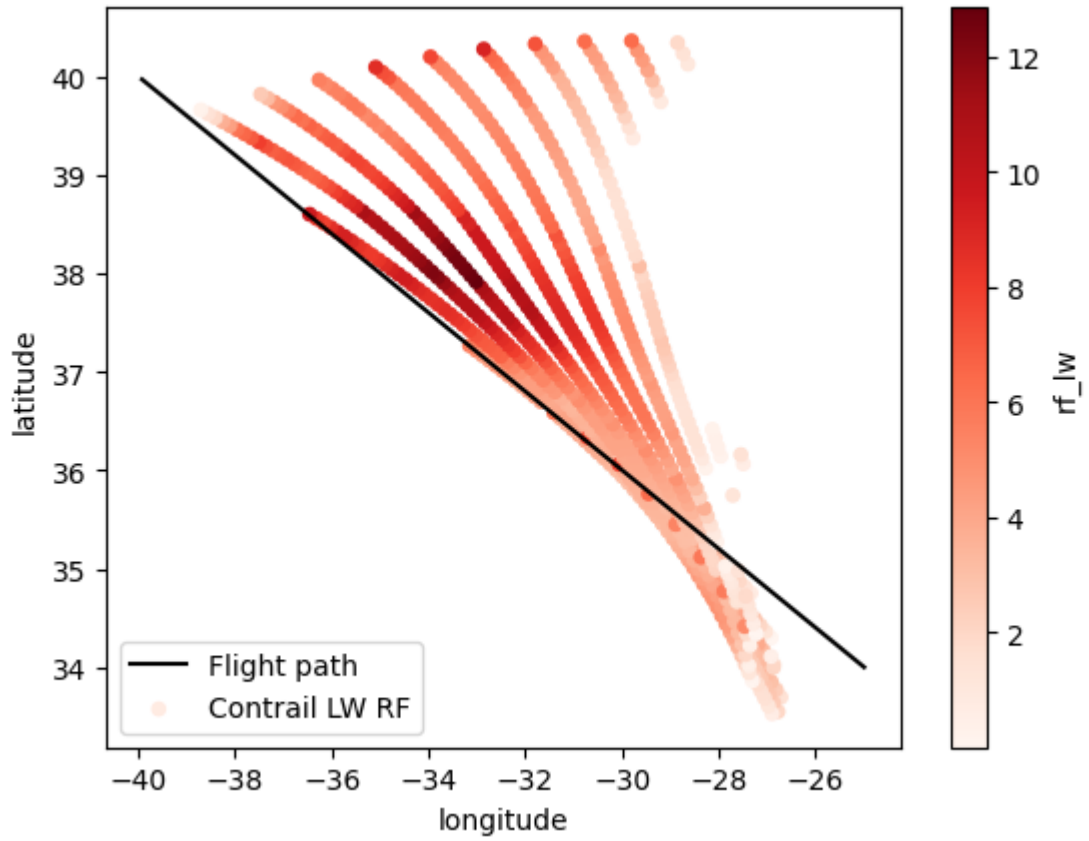
Next, we plot the evolution of the contrail as it advects along each flight segment

[10]:
```
ax = plt.axes()

cocip.source.dataframe.plot(
    "longitude",
    "latitude",
    color="k",
    ax=ax,
    label="Flight path",
```

(continues on next page)

(continued from previous page)

```
)
cocip.contrail.plot.scatter(
    "longitude",
    "latitude",
    c="rf_lw",
    cmap="Reds",
    ax=ax,
    label="Contrail LW RF",  # Contrail age?
)
ax.legend();
```



9.3 Working with Model Level Met

Meteorology providers may choose to compute their data on different vertical levels. For example, the ECMWF IFS product is computed on 137 model levels. The pressure of a given model level is not constant, but varies with time and horizontal position. The pressure at a given point can be computed following ECMWF guidelines. This notebook uses the `metview` library to convert from ECMWF model levels to pressure levels.

The metview package can be difficult to install. Installation with conda may be the easiest way to get started. This notebook assumes a working installation of metview.

[1]:
```
import os
```

(continues on next page)

(continued from previous page)

```
import cdsapi
import matplotlib.pyplot as plt
import metview as mv
import numpy as np
import pandas as pd
import xarray as xr

from pycontrails import Fleet, Flight, MetDataset
from pycontrails.datalib.ecmwf import PRESSURE_LEVEL_VARIABLES, SURFACE_VARIABLES
from pycontrails.models.cocip import Cocip
from pycontrails.models.humidity_scaling import HistogramMatching
from pycontrails.models.ps_model import PSFlight
```

9.3.1 Define CDS model level request

We make three requests to the CDS API.

- `ml_request` is a request for the model levels meteorology data
- `lnsp_request` is a request for the surface pressure data needed for interpolating from model levels to pressure levels
- `sl_request` is a request for the single level top of atmosphere radiation data needed for CoCiP

[2]:
```
date = "2024-01-15"
time = list(range(18))
grid = "1/1"

ml_variables = ["t", "q", "u", "v", "w", "ciwc"]
sl_variables = ["tsr", "ttr"]

ml_request = {
    "levtype": "ml",
    "levellist": "70/to/90/by/1",
    "param": ml_variables,
    "date": date,
    "time": time,
    "grid": grid,
}

lnsp_request = {
    "levtype": "ml",
    "levelist": "1",
    "param": "lnsp",
    "date": date,
    "time": time,
    "grid": grid,
}

sl_request = {
    "levtype": "sfc",
    "product_type": "reanalysis",
```

(continues on next page)

(continued from previous page)

```
    "param": sl_variables,
    "date": date,
    "time": time,
    "grid": grid,
    "format": "netcdf",
}
```

9.3.2 Make the request

Download the data if it is not already sitting on disk. Depending on the load on the Copernicus server, we may get queued here for a while.

[3]:
```
client = cdsapi.Client()

target = "ml.grib"
if not os.path.isfile(target):
    client.retrieve("reanalysis-era5-complete", ml_request, target)

target = "lnsp.grib"
if not os.path.isfile(target):
    client.retrieve("reanalysis-era5-complete", lnsp_request, target)

target = "sl.nc"
if not os.path.isfile(target):
    client.retrieve("reanalysis-era5-single-levels", sl_request, target)
```

9.3.3 Target pressure levels

In order to use the `metview` library to convert from model levels to pressure levels, we must specify target pressure levels for the re-interpolation.

Assuming a constant surface pressure of 1013.25 hPa, the table below shows that ECMWF model levels occur roughly every 10 hPa at altitudes for which contrails form. This informs how we set our target pressure levels: We define our target pressure levels ranging from 170 hPa to 390 hPa, spaced every 10 hPa. A finer spacing than this may be redundant (`pycontrails` models perform linear interpolation between pressure levels), and a coarser spacing would result in loss of information.

[4]:
```
url = "https://confluence.ecmwf.int/display/UDOC/L137+model+level+definitions"
df = pd.read_html(url, na_values="-", index_col="n")[0].rename_axis("hybrid")
df.loc[70:90]  # model levels 70 - 90 agree with our ml_request
```

[4]:

hybrid	a [Pa]	b	ph [hPa]	pf [hPa]	Geopotential Altitude [m]
70	15508.256836	0.011806	167.0450	163.0927	13077.79
71	16026.115234	0.014816	175.2731	171.1591	12771.64
72	16527.322266	0.018318	183.8344	179.5537	12467.99
73	17008.789063	0.022355	192.7389	188.2867	12166.81
74	17467.613281	0.026964	201.9969	197.3679	11868.08
75	17901.621094	0.032176	211.6186	206.8078	11571.79
76	18308.433594	0.038026	221.6146	216.6166	11277.92
77	18685.718750	0.044548	231.9954	226.8050	10986.70

(continues on next page)

(continued from previous page)

```
78      19031.289063  0.051773  242.7719  237.3837                  10696.22
79      19343.511719  0.059728  253.9549  248.3634                  10405.61
80      19620.042969  0.068448  265.5556  259.7553                  10114.89
81      19859.390625  0.077958  277.5852  271.5704                   9824.08
82      20059.931641  0.088286  290.0548  283.8200                   9533.20
83      20219.664063  0.099462  302.9762  296.5155                   9242.26
84      20337.863281  0.111505  316.3607  309.6684                   8951.30
85      20412.308594  0.124448  330.2202  323.2904                   8660.32
86      20442.078125  0.138313  344.5663  337.3932                   8369.35
87      20425.718750  0.153125  359.4111  351.9887                   8078.41
88      20361.816406  0.168910  374.7666  367.0889                   7787.51
89      20249.511719  0.185689  390.6450  382.7058                   7496.68
90      20087.085938  0.203491  407.0583  398.8516                   7205.93

        Geometric Altitude [m]  Temperature [K]  Density [kg/m^3]
hybrid
70                   13104.70           216.65          0.262244
71                   12797.30           216.65          0.275215
72                   12492.44           216.65          0.288713
73                   12190.10           216.65          0.302755
74                   11890.24           216.65          0.317357
75                   11592.86           216.65          0.332536
76                   11297.93           216.65          0.348308
77                   11005.69           216.74          0.364545
78                   10714.22           218.62          0.378253
79                   10422.64           220.51          0.392358
80                   10130.98           222.40          0.406868
81                    9839.26           224.29          0.421790
82                    9547.49           226.18          0.437130
83                    9255.70           228.08          0.452897
84                    8963.90           229.97          0.469097
85                    8672.11           231.86          0.485737
86                    8380.36           233.75          0.502825
87                    8088.67           235.64          0.520367
88                    7797.04           237.53          0.538370
89                    7505.51           239.42          0.556842
90                    7214.09           241.31          0.575790
```

9.3.4 Interpolate from model levels to pressure levels

We use the mvl_ml2hPa metview function to interpolate from model levels to pressure levels. This function has some constraints:

- Only a single met parameter can be interpolated in a given function call
- Only a single time can be interpolated in a given function call
- The target pressure levels are assumed to be integers

We loop over all parameters and times to interpolate each parameter at each time. We then concatenate the results into a single `xr.Dataset`.

[5]:
```
pressures = np.arange(170, 400, 10)

ds_dict = {}
for param in ml_variables:
    da_list = []
    for t in time:
        lnsp = mv.read(source="lnsp.grib", time=t)
        fsml = mv.read(source="ml.grib", param=param, time=t)
        fspl = mv.mvl_ml2hPa(lnsp, fsml, pressures)
        ds = fspl.to_dataset()
        da = ds.expand_dims("time")[param]
        da_list.append(da)

    da = xr.concat(da_list, dim="time")
    ds_dict[param] = da

ds = xr.Dataset(ds_dict)
```

[6]:
```
# Ready the Dataset for pycontrails
met = MetDataset(
    ds.rename(isobaricInhPa="level"),
    provider="ECMWF",
    dataset="ERA5",
    product="reanalysis",
)
met.standardize_variables(PRESSURE_LEVEL_VARIABLES)
met
```

[6]:
```
MetDataset with data:

<xarray.Dataset> Size: 647MB
Dimensions:                                (time: 18, level: 23, latitude: 181,
                                            longitude: 360)
Coordinates:
  * time                                   (time) datetime64[ns] 144B 2024-01-1...
    step                                   timedelta64[ns] 8B 00:00:00
  * level                                  (level) float64 184B 170.0 ... 390.0
  * latitude                               (latitude) float64 1kB -90.0 ... 90.0
    valid_time                             (time) datetime64[ns] 144B 2024-01-1...
  * longitude                              (longitude) float64 3kB -180.0 ... 1...
    air_pressure                           (level) float32 92B 1.7e+04 ... 3.9e+04
    altitude                               (level) float32 92B 1.281e+04 ... 7...
Data variables:
    air_temperature                        (longitude, latitude, level, time) float32␣
↪108MB ...
    specific_humidity                      (longitude, latitude, level, time) float32␣
↪108MB ...
    eastward_wind                          (longitude, latitude, level, time) float32␣
↪108MB ...
    northward_wind                         (longitude, latitude, level, time) float32␣
↪108MB ...
    lagrangian_tendency_of_air_pressure    (longitude, latitude, level, time) float32␣
↪108MB ...
```

(continues on next page)

(continued from previous page)

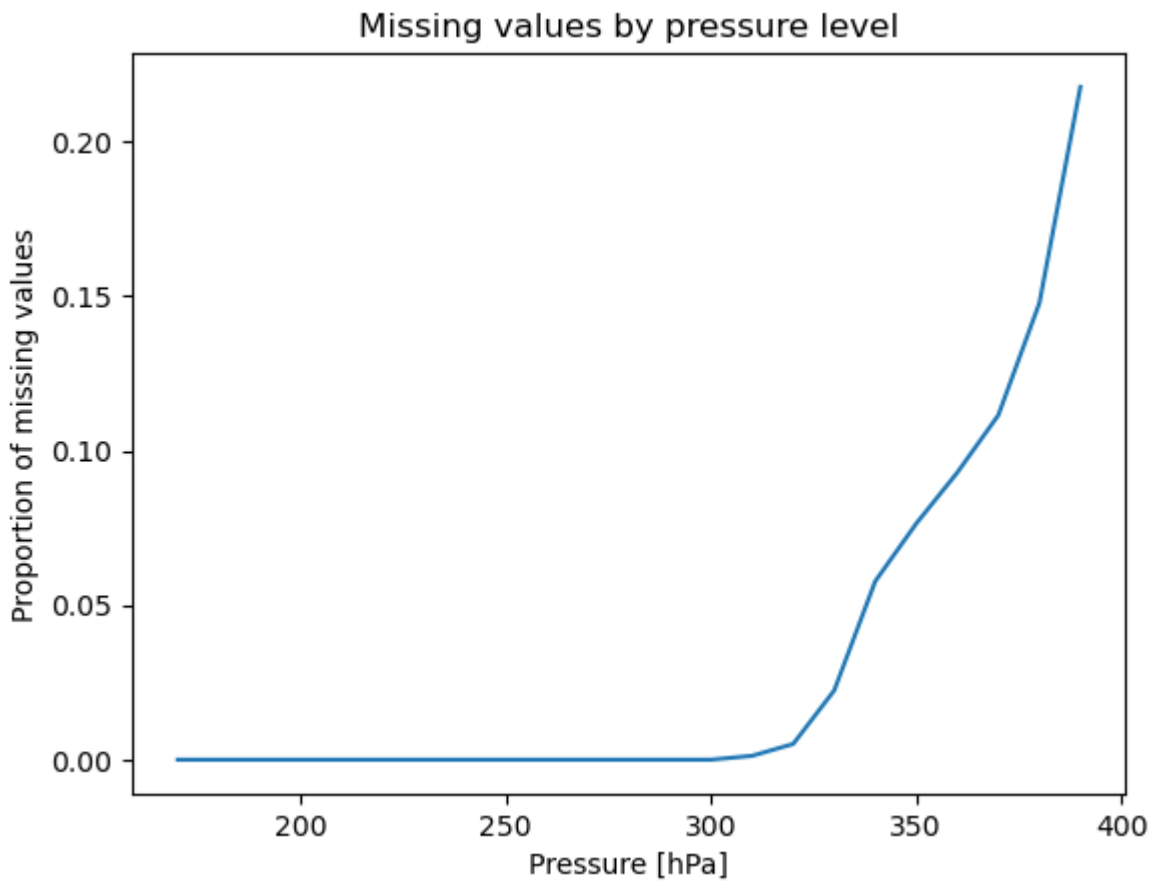
```
    specific_cloud_ice_water_content     (longitude, latitude, level, time) float32␣
↪108MB ...
Attributes:
    provider:  ECMWF
    dataset:   ERA5
    product:   reanalysis
```

9.3.5 Missing values

Converting from model levels to pressure levels will result in some missing values. These are plotted below.

Points at which the surface pressure is significantly lower than the 1013.25 hPa reference surface pressure will result in missing values at the lowest-altitude target pressure levels. Additional model levels can be downloaded to decrease the number of missing values. Missing values do not occur at higher-altitude target pressure levels.

[7]:
```
da = met.data["air_temperature"]
da.isnull().groupby("level").mean(...).plot()
plt.ylabel("Proportion of missing values")
plt.title("Missing values by pressure level");
```

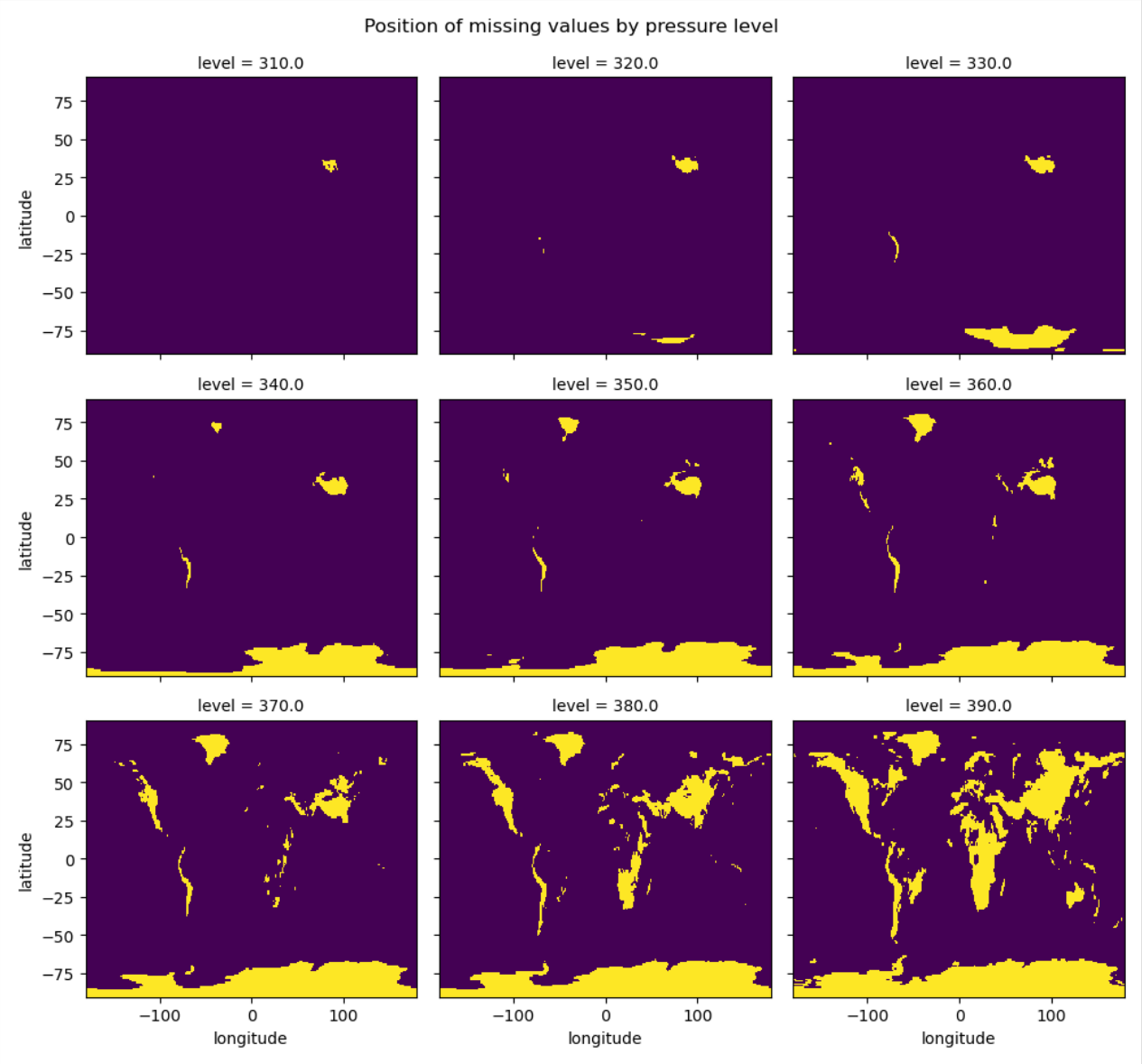


[8]:
```
# There are no missing values above 310 hPa
tmp = da.isnull().isel(time=0).sel(level=slice(310, None))
```

(continues on next page)

(continued from previous page)

```
tmp.plot(x="longitude", y="latitude", add_colorbar=False, col="level", col_wrap=3)
plt.gcf().suptitle("Position of missing values by pressure level", y=1.02);
```



9.3.6 Single level data

The single level data is much easier to work with. We can read it directly with `xarray` using the `netcdf4` engine.

[9]:
```
# Ready the single level data for pycontrails
ds = xr.open_dataset("sl.nc")
ds = ds.expand_dims(level=[-1])

rad = MetDataset(ds, provider="ECMWF", dataset="ERA5", product="reanalysis")
rad.standardize_variables(SURFACE_VARIABLES)
rad
```

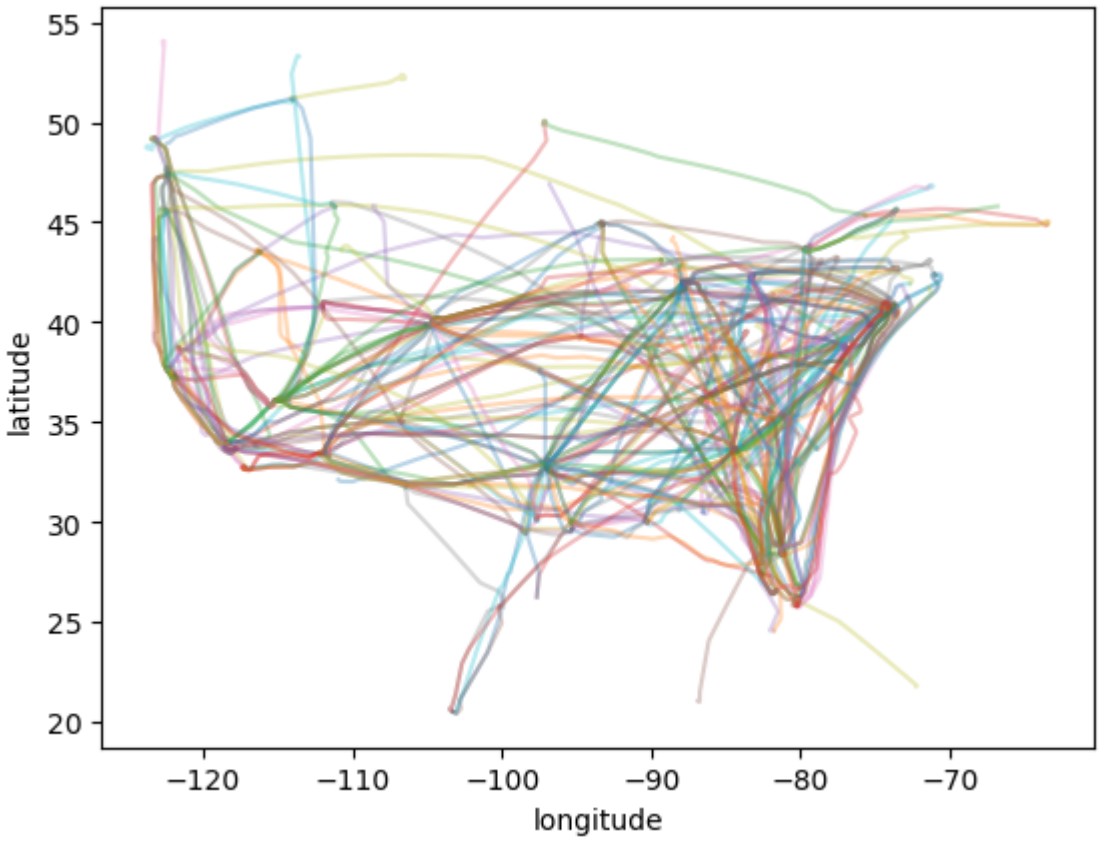
[9]:
```
MetDataset with data:

<xarray.Dataset> Size: 9MB
Dimensions:                    (level: 1, latitude: 181, time: 18,
                                longitude: 360)
Coordinates:
  * level                      (level) float64 8B -1.0
  * latitude                   (latitude) float64 1kB -90.0 -89.0 ... 89.0 90.0
  * time                       (time) datetime64[ns] 144B 2024-01-15 ... 2024...
  * longitude                  (longitude) float64 3kB -180.0 -179.0 ... 179.0
Data variables:
    top_net_solar_radiation    (longitude, latitude, level, time) float32 5MB ...
    top_net_thermal_radiation  (longitude, latitude, level, time) float32 5MB ...
Attributes:
    Conventions:  CF-1.6
    history:      2024-03-05 06:57:34 GMT by grib_to_netcdf-2.25.1: /opt/ecmw...
    provider:     ECMWF
    dataset:      ERA5
    product:      reanalysis
```

9.3.7 Download ADS-B

For flight trajectories, we use a sample from Contrails API. We artificially shift the times of the flight trajectories to overlap the times of the ERA5 data.

[10]:
```
df = pd.read_csv("https://apidocs.contrails.org/_static/fleet_sample.csv", parse_dates=[
↪"time"])
df["time"] = df["time"].dt.tz_convert(None)
df = df.rename(columns={"altitude": "altitude_ft"})

# Shift time to the date of interest
df["time"] = df["time"] + (pd.Timestamp(date) - df["time"].min())

# Convert to a pycontrails Fleet instance, keeping only aircraft type covered by the PS␣
↪model
ps_flight = PSFlight()
flights = []
for flight_id, group in df.groupby("flight_id"):
    aircraft_type = group["aircraft_type"].iloc[0]
    if not ps_flight.check_aircraft_type_availability(aircraft_type, raise_error=False):
        continue

    engine_uid = group["engine_uid"].iloc[0]
    group = group.drop(columns=["aircraft_type", "engine_uid", "flight_id"])
    flight = Flight(group, aircraft_type=aircraft_type, engine_uid=engine_uid, flight_
↪id=flight_id)
    flights.append(flight)

fleet = Fleet.from_seq(flights)
```

[11]:
```
ax = plt.subplot()
for flight in flights:
    flight.plot(ax=ax, alpha=0.3)
```



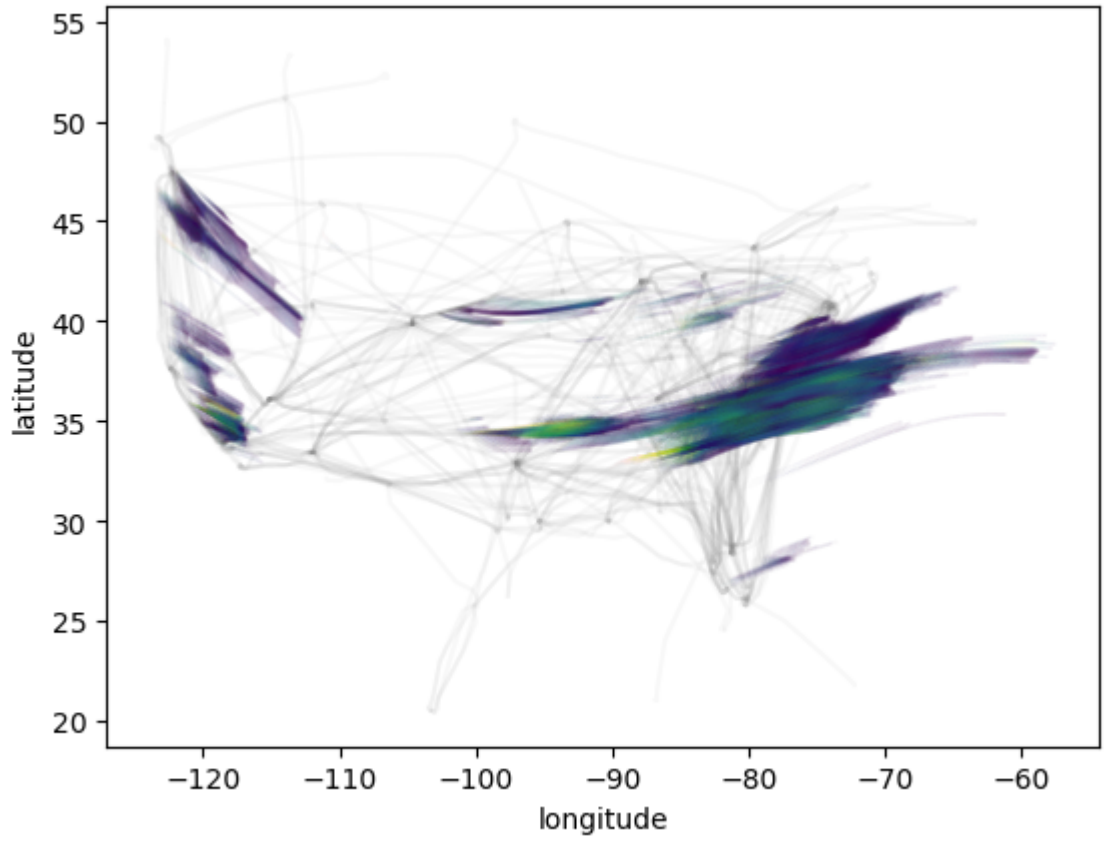
9.3.8 Run CoCiP

We run CoCiP on the model-level ERA5 data.

[12]:
```
cocip = Cocip(
    met=met,
    rad=rad,
    dt_integration="5 min",
    max_age="12 hours",
    aircraft_performance=PSFlight(),
    humidity_scaling=HistogramMatching(),
)
cocip_pred = cocip.eval(fleet)
contrail = cocip.contrail
```

9.3.9 Visualize output

[13]:
```
ax = plt.subplot()
for flight in flights:
    flight.plot(ax=ax, color="gray", alpha=0.05)

contrail.plot.scatter(
    x="longitude",
    y="latitude",
    s=contrail["width"] / 100000,
    c=contrail["tau_contrail"],
    vmin=0,
    vmax=0.3,
    alpha=0.05,
    zorder=2,
    ax=ax,
);
```



CHAPTER

TEN

SPECIFIC HUMIDITY INTERPOLATION

This document discusses best practices for interpolating gridded specific humidity. The focus here is on ECMWF data, but the principles apply to other datasets as well.

Among all meteorology variables, humidity plays a special role in contrail prediction. Generally, contrails will not persist outside of ice super-saturated regions. The energy forcing predictions from CoCiP are themselves extremely sensitive to humidity. Small perturbations in humidity can lead to large changes in radiative forcing.

10.1 ERA5 data

In this notebook, we use the `ARCOERA5` interface to the publicly available ARCO ERA5 "model-level" data.

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
import xarray as xr
from scipy.interpolate import PchipInterpolator

from pycontrails.core import models
from pycontrails.datalib.ecmwf import ARCOERA5
from pycontrails.physics import thermo
```

10.1.1 Regrid to pressure levels

ERA5 data is often provided at discrete pressure levels. Below we regrid the "model level" data from ARCO ERA5 to discrete pressure levels. The spacing of these vertical pressure levels has been chosen to match common spacing of the ERA5 data.

While gridded forecast and reanalysis data is far from perfect, the coarseness of the gridded product can introduce additional errors when interpolating between pressure levels. A model such as CoCiP will proceed with higher precision as additional pressure levels are included in the met data. **We recommend using the highest resolution data available that you can afford to process.** Many of the interpolation artifacts discussed below become negligible when using pressure levels spaced at 10 hPa or less. At cruising altitudes, this is roughly a vertical resolution of 1000 ft or less.

Below we create both a coarsely and finely spaced pressure level dataset for comparison purposes.

[2]:
```
pl_coarse = [100, 150, 200, 250, 300, 350, 400]
pl_fine = [200, 210, 220, 230, 240, 250, 260, 270, 280, 290, 300]

time = "2021-03-14T15"
```

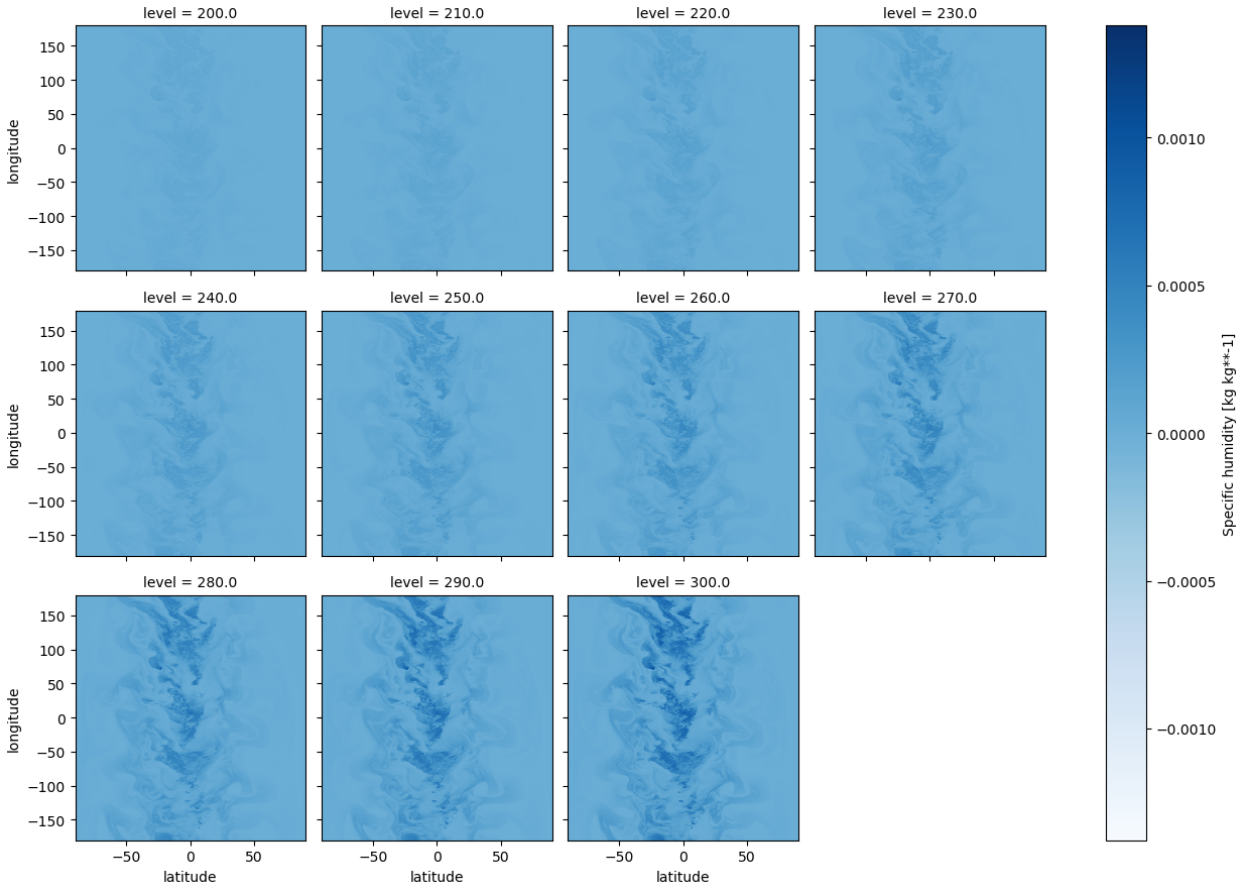
(continues on next page)

(continued from previous page)

```
variables = ["t", "q", "ciwc"]

arco_coarse = ARCOERA5(time, variables, pl_coarse)
ds_coarse = arco_coarse.open_metdataset().data
ds_coarse.load()

arco_fine = ARCOERA5(time, variables, pl_fine)
ds_fine = arco_fine.open_metdataset().data
ds_fine.load();
```

[3]:
```
# Add RHi to the datasets
for ds in (ds_coarse, ds_fine):
    ds["rhi"] = thermo.rhi(ds["specific_humidity"], ds["air_temperature"], ds["level"] *␣
↪100.0)
```

[4]:
```
ds_fine["specific_humidity"].squeeze().plot(col="level", col_wrap=4, cmap="Blues");
```



10.2 Lapse rates

Lapse rate typically refers to the rate at which temperature decreases with atmospheric height. Similarly, we can also consider the rate at which specific humidity or cloud ice water content decreases with height. The lapse rate of specific humidity exhibits important differences from the classic temperature lapse rate.

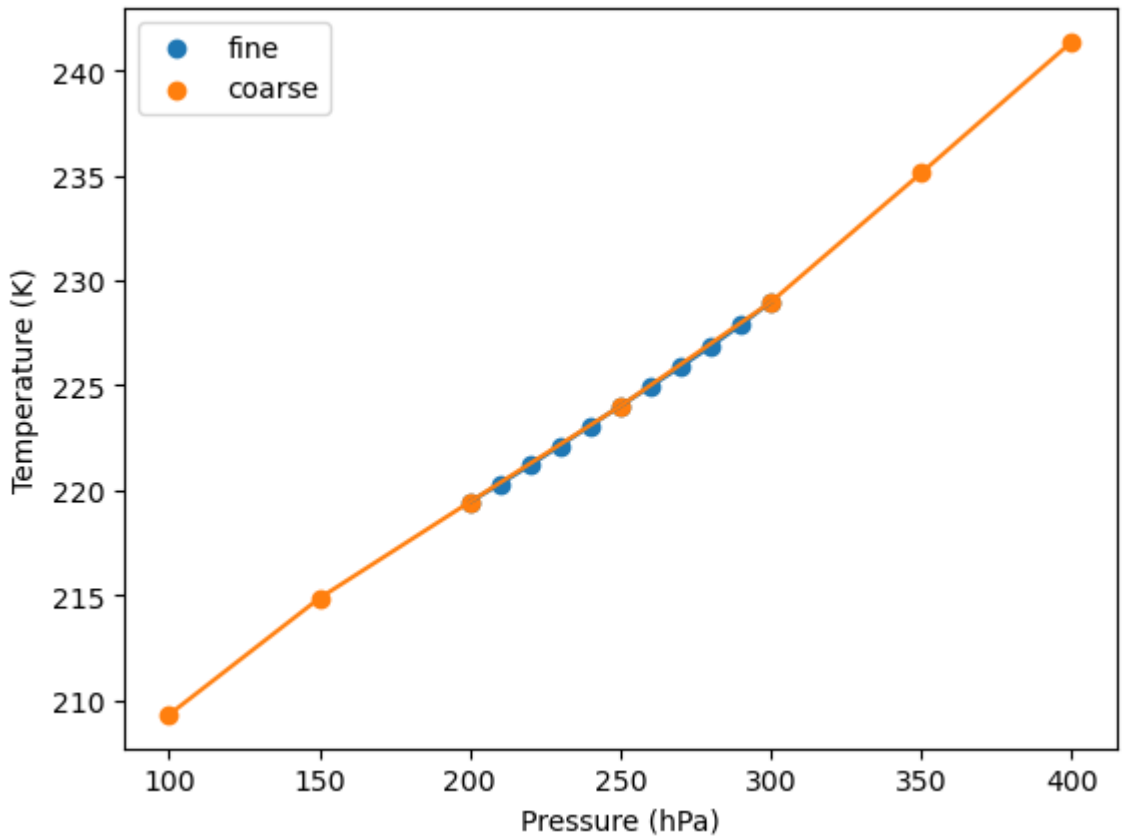
One can use thermodynamics to derive a theoretical temperature lapse rate. Below, we compute the average temperature of the ERA5 data at each pressure level as another way of thinking about the temperature lapse rate.

[5]:
```
ds_coarse_mean = ds_coarse.groupby("level").mean(...)
ds_fine_mean = ds_fine.groupby("level").mean(...)
```

10.2.1 Temperature lapse rate

Below, we show the mean temperature profile over the entire domain. Importantly, the **temperature profile is roughly linear over the pressure levels** used here.

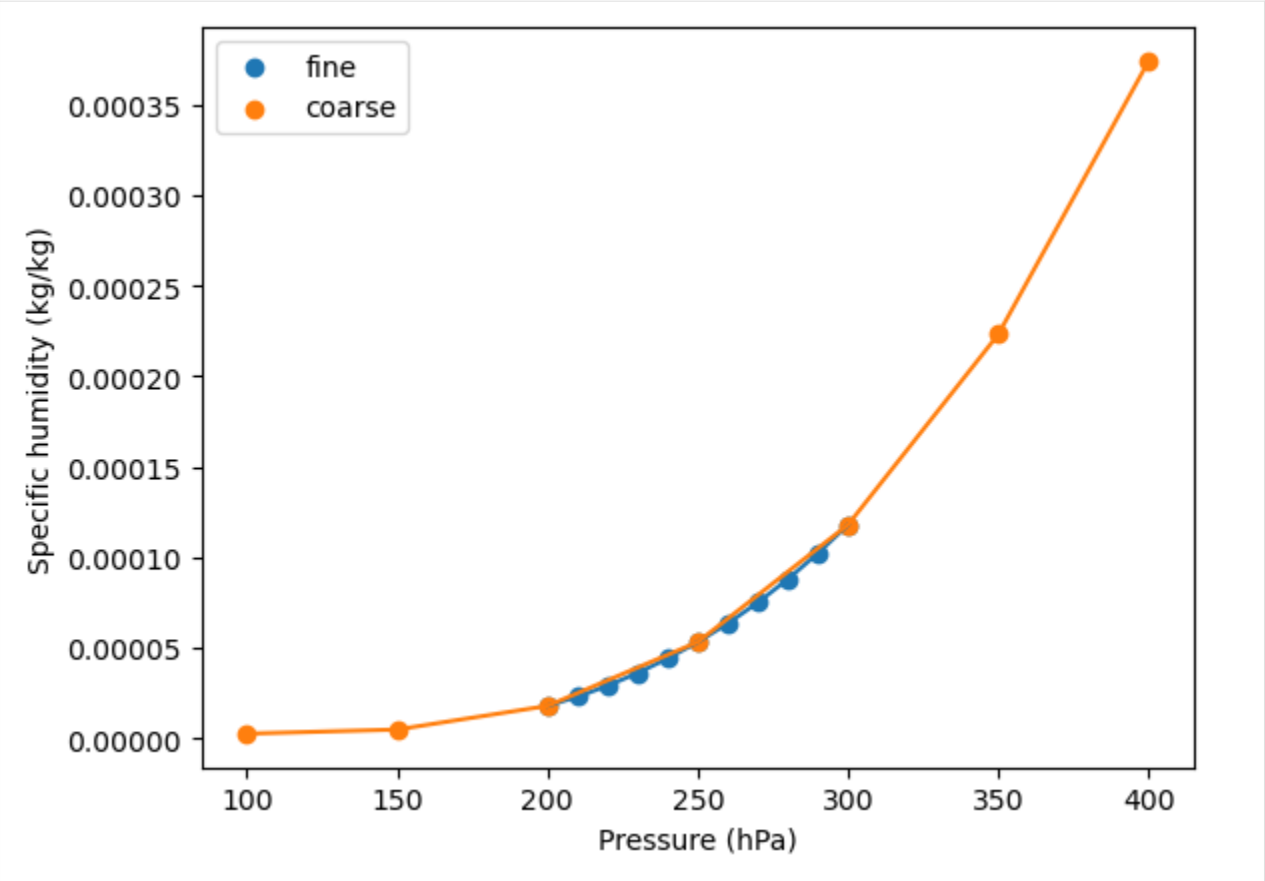
[6]:
```
fig, ax = plt.subplots()

ax.scatter(ds_fine_mean["air_temperature"].level, ds_fine_mean["air_temperature"], label=
↪"fine")
ax.plot(ds_fine_mean["air_temperature"].level, ds_fine_mean["air_temperature"])

ax.scatter(
    ds_coarse_mean["air_temperature"].level, ds_coarse_mean["air_temperature"], label=
↪"coarse"
)
ax.plot(ds_coarse_mean["air_temperature"].level, ds_coarse_mean["air_temperature"])

ax.legend()
ax.set_ylabel("Temperature (K)")
ax.set_xlabel("Pressure (hPa)");
```



10.2.2 Specific humidity lapse rate

Make the same plot for specific humidity. Importantly, the specific humidity profile is **nonlinear**. In particular, the plot shows a pronounced upward concavity, primarily because saturation vapor pressure varies exponentially with temperature.

[7]:
```
fig, ax = plt.subplots()

ax.scatter(ds_fine_mean["specific_humidity"].level, ds_fine_mean["specific_humidity"],␣
↪label="fine")
ax.plot(ds_fine_mean["specific_humidity"].level, ds_fine_mean["specific_humidity"])

ax.scatter(
    ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"],␣
↪label="coarse"
)
ax.plot(ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"])

ax.legend()
ax.set_ylabel("Specific humidity (kg/kg)")
ax.set_xlabel("Pressure (hPa)");
```



10.2.3 Cloud ice water content lapse rate

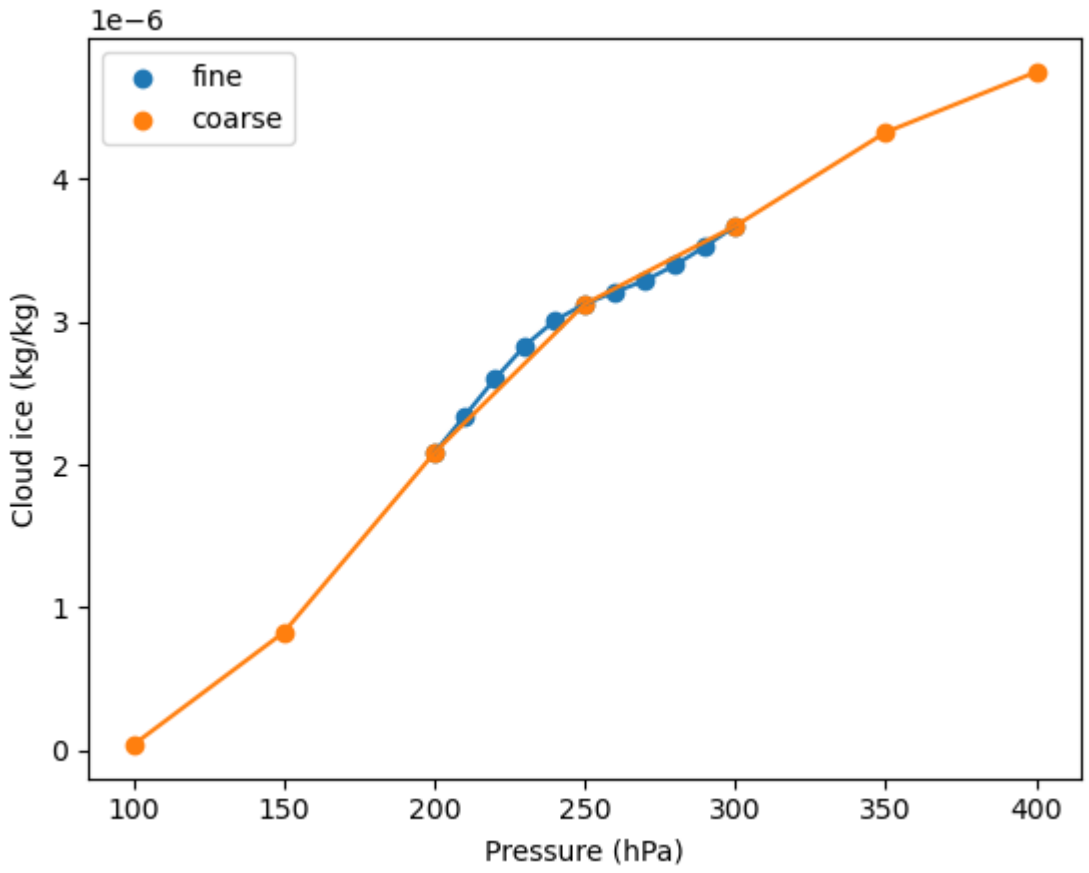
Showing the same profile for cloud ice water content. This variable is somewhat less critical because CoCiP does not have a high sensitivity on it. We include it here as a point of comparison.

[8]:
```
fig, ax = plt.subplots()

ax.scatter(
    ds_fine_mean["specific_cloud_ice_water_content"].level,
    ds_fine_mean["specific_cloud_ice_water_content"],
    label="fine",
)
ax.plot(
    ds_fine_mean["specific_cloud_ice_water_content"].level,
    ds_fine_mean["specific_cloud_ice_water_content"],
)

ax.scatter(
    ds_coarse_mean["specific_cloud_ice_water_content"].level,
    ds_coarse_mean["specific_cloud_ice_water_content"],
    label="coarse",
)
ax.plot(
```

(continues on next page)

(continued from previous page)

```
    ds_coarse_mean["specific_cloud_ice_water_content"].level,
    ds_coarse_mean["specific_cloud_ice_water_content"],
)

ax.legend()
ax.set_ylabel("Cloud ice (kg/kg)")
ax.set_xlabel("Pressure (hPa)");
```



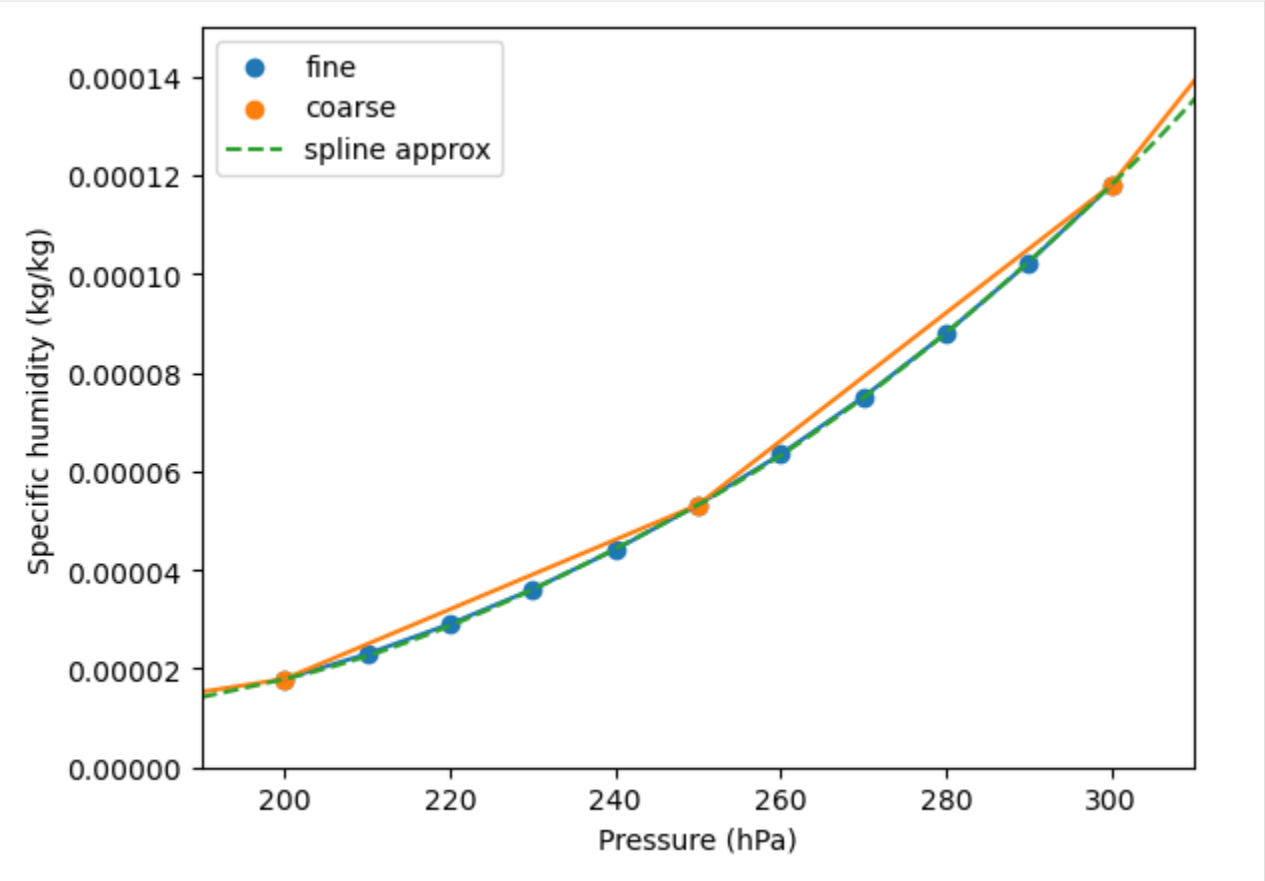
10.3 The problem

A nonlinear lapse rate may lead to bias if using linear interpolation.

Because the specific humidity lapse rate has a positive second derivative, a chord connecting two points on the lapse rate curve will lie above the curve. This means that linear interpolation will tend to overestimate the specific humidity on average at intermediate points.

We demonstrate this below by comparing the coarse and fine pressure level datasets. We also construct a monotonic cubic spline from the coarse pressure level lapse rate as a means of approximating the fine pressure level lapse rate. In particular, this spline approximation is much closer to the fine pressure level lapse rate than a naive linear approximation of the coarse pressure level lapse rate.

[9]:
```
fig, ax = plt.subplots()

ax.scatter(ds_fine_mean["specific_humidity"].level, ds_fine_mean["specific_humidity"],␣
↪label="fine")
ax.plot(ds_fine_mean["specific_humidity"].level, ds_fine_mean["specific_humidity"])

ax.scatter(
    ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"],␣
↪label="coarse"
)
ax.plot(ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"])

pchip = PchipInterpolator(
    ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"]
)
p = np.linspace(100, 400, 100)
ax.plot(p, pchip(p), label="spline approx", linestyle="--")
ax.legend()


ax.legend()
ax.set_ylabel("Specific humidity (kg/kg)")
ax.set_xlabel("Pressure (hPa)")

ax.set_xlim(190, 310)
ax.set_ylim(0, 0.00015);
```



Below, we ignore the fine pressure level dataset lapse rate and only use the coarse pressure level dataset lapse rate to compute an interpolated specific humidity value. In particular, we compare two interpolation methods:

- naive linear interpolation
- monotonic cubic spline interpolation

In light of the previous plot, we expect the monotonic cubic spline to better approximate the lapse rate associated to the fine pressure level dataset. The plot below shows that the two interpolation methods can differ by as much as 5%. This error is a significant, and suggests that linear interpolation in specific humidity may lead to a bias in CoCiP when using coarse pressure level met data.

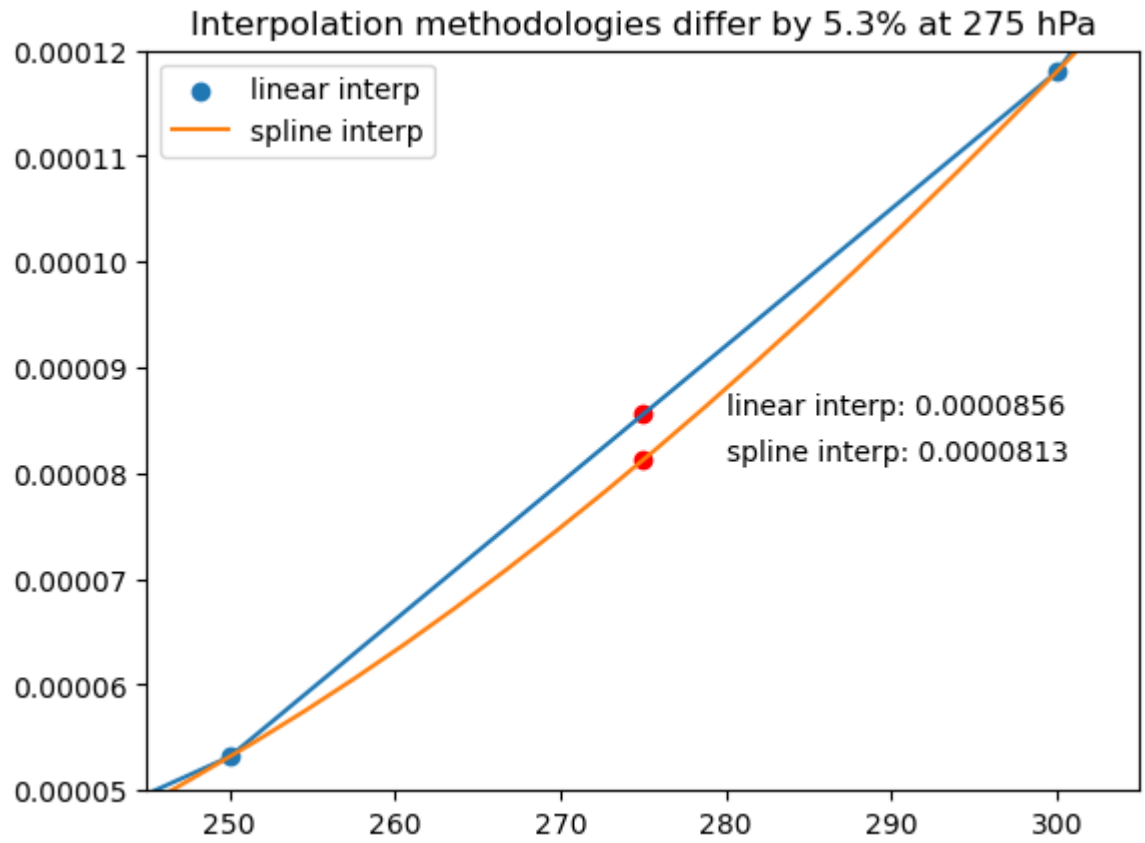
This sort of data omission experiment is pursued more *rigorously below*.

[10]:
```
fig, ax = plt.subplots()

ax.scatter(
    ds_coarse_mean["specific_humidity"].level,
    ds_coarse_mean["specific_humidity"],
    label="linear interp",
)
ax.plot(ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"])

ax.plot(p, pchip(p), label="spline interp")
ax.legend()
ax.set_xlim(245, 305)
ax.set_ylim(0.00005, 0.00012)
```

(continues on next page)

(continued from previous page)

```
p0 = 275
y0 = ds_coarse_mean["specific_humidity"].interp(level=p0).values
y1 = pchip(p0)
pct = (y0 - y1) / y1 * 100.0

ax.scatter([p0, p0], [y0, y1], color="red")
ax.annotate(f"linear interp: {y0:.7f}", (p0, y0), (p0 + 5, y0))
ax.annotate(f"spline interp: {y1:.7f}", (p0, y1), (p0 + 5, y1))
ax.set_title(f"Interpolation methodologies differ by {pct:.1f}% at {p0} hPa");
```



10.4 Solutions

10.4.1 Interpolate in the log-log domain

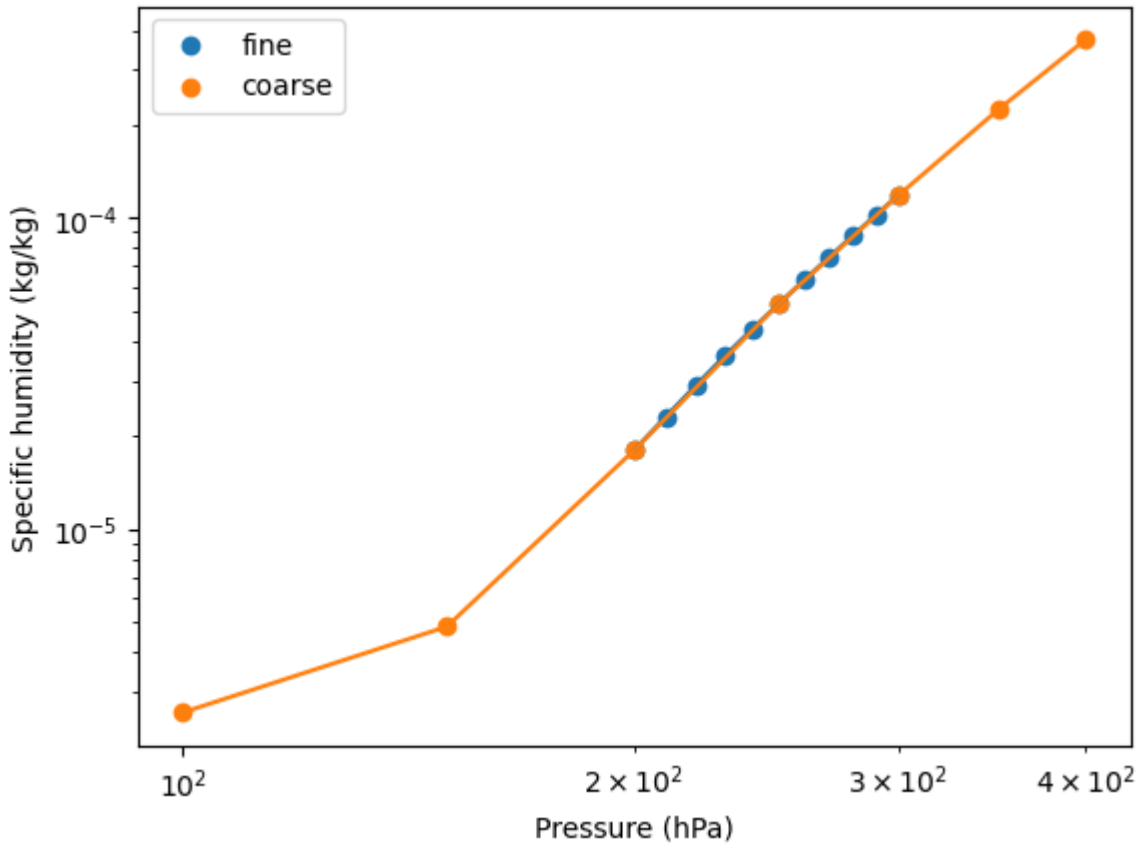
Because both pressure and specific humidity are positive quantities that span several orders of magnitude, it is natural to interpolate in the logarithm of each. Below we plot the lapse rate in the log-log domain. Visibly, the log-log lapse rate is more linear than the lapse rate in the original domain.

[11]:
```
fig, ax = plt.subplots()

ax.scatter(ds_fine_mean["specific_humidity"].level, ds_fine_mean["specific_humidity"],␣
```

(continues on next page)

(continued from previous page)

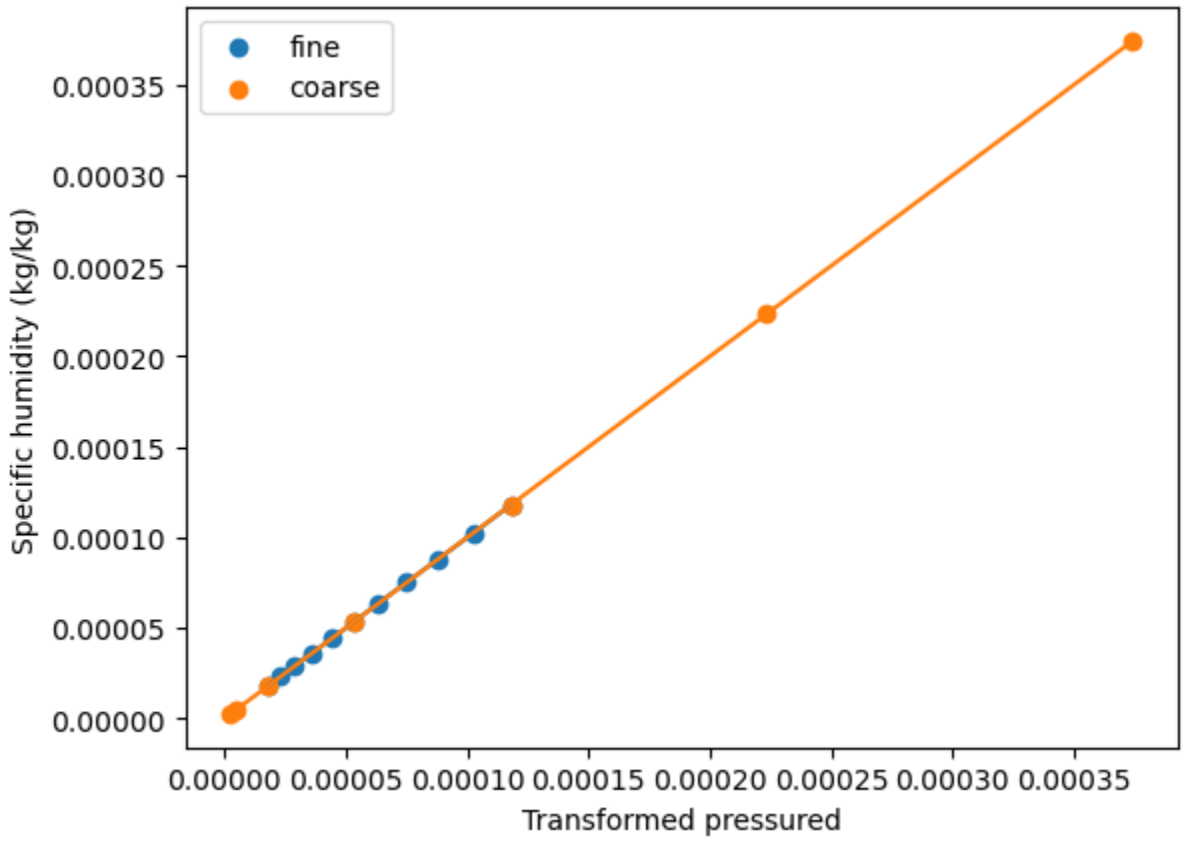
```
↪label="fine")
ax.plot(ds_fine_mean["specific_humidity"].level, ds_fine_mean["specific_humidity"])

ax.scatter(
    ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"],␣
↪label="coarse"
)
ax.plot(ds_coarse_mean["specific_humidity"].level, ds_coarse_mean["specific_humidity"])

ax.set_xscale("log")
ax.set_yscale("log")

ax.legend()
ax.set_ylabel("Specific humidity (kg/kg)")
ax.set_xlabel("Pressure (hPa)");
```



10.4.2 Use the cubic-spline to linearize the lapse rate

We can manually transform the lapse rate curve by normalizing the pressure domain by the lapse rate. We use the previously computed cubic spline to achieve this. For this approach to be applicable generally, the lapse rate must be monotonic.

[12]:
```
fig, ax = plt.subplots()

ax.scatter(
    pchip(ds_fine_mean["specific_humidity"].level), ds_fine_mean["specific_humidity"],␣
↪label="fine"
)
ax.plot(pchip(ds_fine_mean["specific_humidity"].level), ds_fine_mean["specific_humidity
↪"])

ax.scatter(
    pchip(ds_coarse_mean["specific_humidity"].level),
    ds_coarse_mean["specific_humidity"],
    label="coarse",
)
ax.plot(pchip(ds_coarse_mean["specific_humidity"].level), ds_coarse_mean["specific_
↪humidity"])

ax.legend()
ax.set_ylabel("Specific humidity (kg/kg)")
ax.set_xlabel("Transformed pressured");
```



10.4.3 Interpolate in the gridded RHi space

Gridded relative humidity over ice (RHi) does not exhibit a clear nonlinear lapse rate. Instead of interpolating specific humidity, we could instead interpolate gridded RHi directly. This approach is intuitive to explain, but also has some drawbacks. In particular, the RHi field is not defined in regions where the temperature is above freezing. This means that we would need to use a different interpolation method in these regions.

10.5 Data Omission Experiment

To evaluate each interpolation method, we perform a data omission experiment. For each pressure level in the fine pressure level dataset, we compute the interpolated specific humidity using the coarse pressure level dataset. We then compare this interpolated value to the value in the fine pressure level dataset.

We run this experiment for the four pressure levels 260, 270, 280, and 290 hPa. These are situated between the coarse pressure level dataset levels at 250 and 300 hPa.

10.5.1 Error metrics

We look at mean squared error (MSE) and mean absolute error (MAE) for each interpolation method. Instead of working with the full dataset, we compute error metrics only near the ISSR threshold. This is the region of the atmosphere where the specific humidity is most important for contrail prediction.

[13]:
```
def run_experiment(level):
    """Compute RHi using different interpolation methods."""

    rhi_true = ds_fine["rhi"].sel(level=level)
    t_true = ds_fine["air_temperature"].sel(level=level)

    q_top_bottom = ds_coarse["specific_humidity"].sel(level=[250, 300])
    rhi_top_bottom = ds_coarse["rhi"].sel(level=[250, 300])

    # linear-q interpolation
    tmp = q_top_bottom.interp(level=level)
    linear_q = thermo.rhi(tmp, t_true, level * 100.0)

    # log-q-log-p interpolation
    tmp = q_top_bottom.where(q_top_bottom > 0)
    tmp = np.log(tmp)
    tmp = tmp.assign_coords(level=np.log(tmp["level"]))
    tmp = tmp.interp(level=np.log(level))
    tmp = np.exp(tmp)
    log_q_log_p = thermo.rhi(tmp, t_true, level * 100.0)

    # cubic-spline; using the pre-computed spline in pycontrails
    # the `models._load_spline()` spline was computed in a similar way to the
    # spline created earlier in this notebook
    pchip = models._load_spline()
    tmp = q_top_bottom.assign_coords(level=pchip(q_top_bottom["level"]))
    tmp = tmp.interp(level=pchip(level))
    cubic_spline = thermo.rhi(tmp, t_true, level * 100.0)
```

(continues on next page)

(continued from previous page)

```
    # interp in rhi space
    linear_rhi = rhi_top_bottom.interp(level=level)

    return rhi_true, linear_q, log_q_log_p, cubic_spline, linear_rhi


rhi_true_list = []
linear_q_list = []
log_q_log_p_list = []
cubic_spline_list = []
linear_rhi_list = []

for level in (260, 270, 280, 290):
    rhi_true, linear_q, log_q_log_p, cubic_spline, linear_rhi = run_experiment(level)
    rhi_true_list.append(rhi_true.expand_dims("level"))
    linear_q_list.append(linear_q.expand_dims("level"))
    log_q_log_p_list.append(log_q_log_p.expand_dims("level"))
    cubic_spline_list.append(cubic_spline.expand_dims("level"))
    linear_rhi_list.append(linear_rhi.expand_dims("level"))

rhi_true = xr.concat(rhi_true_list, "level")
linear_q = xr.concat(linear_q_list, "level")
log_q_log_p = xr.concat(log_q_log_p_list, "level")
cubic_spline = xr.concat(cubic_spline_list, "level")
linear_rhi = xr.concat(linear_rhi_list, "level")
```

10.5.2 Visualize residuals

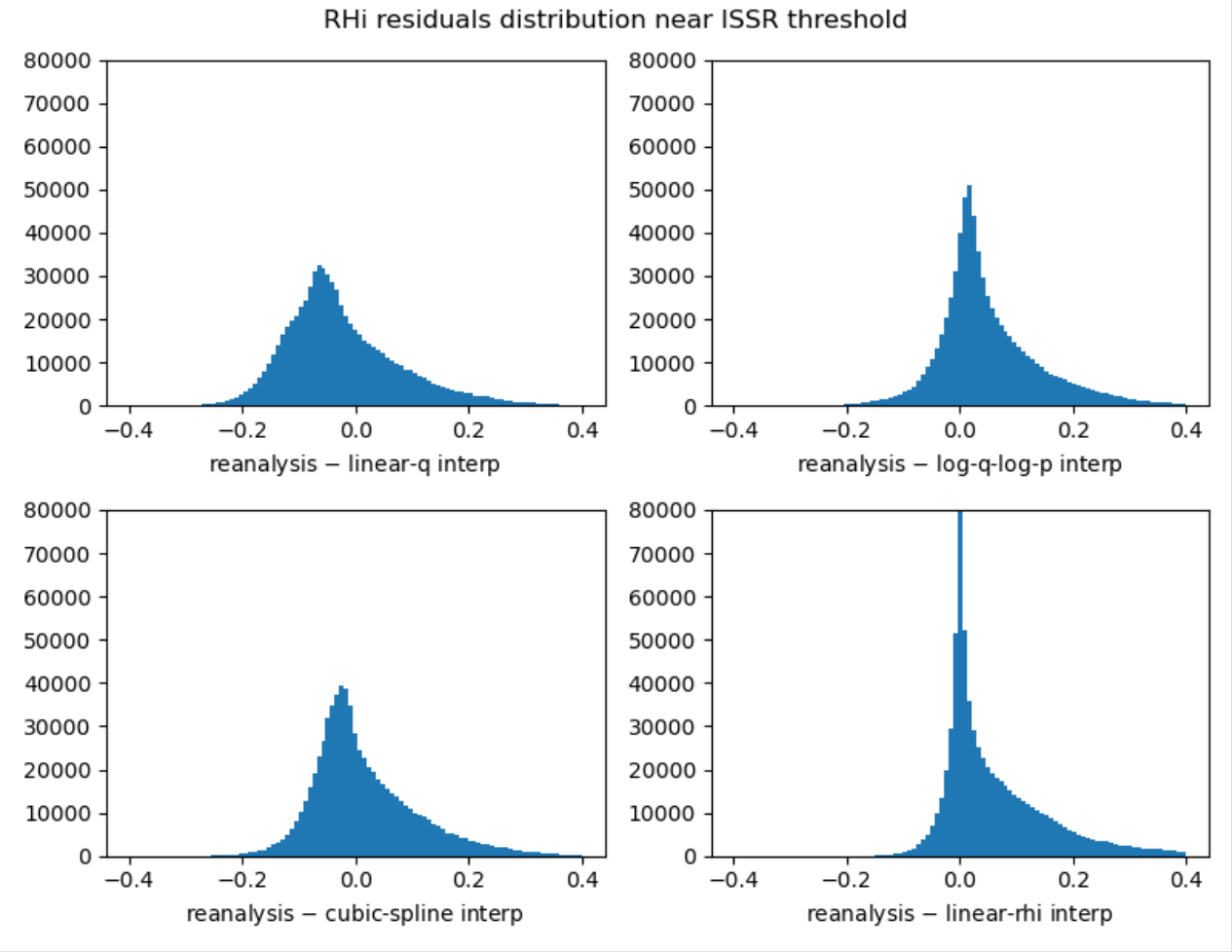
- Linear interpolation in q ("linear-q") shows a clear bias in the residuals. This is the same bias previously discussed arising from the "concave up" lapse rate.
- Linear interpolation in the RHi grid ("linear-rhi") shows some bias as well. In particular, the residual mean is further from zero than the other methods.
- Interpolation in the log-log domain ("log-q-log-p") and the cubic spline method ("cubic-spline") both appear promising.

[14]:
```
fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(figsize=(8, 6), ncols=2, nrows=2)

bins = np.linspace(-0.4, 0.4, 100)
cond = (rhi_true > 0.9) & (rhi_true < 1.3)

ax1.hist((rhi_true - linear_q).values[cond], bins=bins)
ax2.hist((rhi_true - log_q_log_p).values[cond], bins=bins)
ax3.hist((rhi_true - cubic_spline).values[cond], bins=bins)
ax4.hist((rhi_true - linear_rhi).values[cond], bins=bins)

ax1.set_xlabel("reanalysis $-$ linear-q interp")
ax2.set_xlabel("reanalysis $-$ log-q-log-p interp")
ax3.set_xlabel("reanalysis $-$ cubic-spline interp")
ax4.set_xlabel("reanalysis $-$ linear-rhi interp")
```

(continues on next page)

(continued from previous page)

```
ax1.set_ylim(0, 80000)
ax2.set_ylim(0, 80000)
ax3.set_ylim(0, 80000)
ax4.set_ylim(0, 80000)

plt.tight_layout()
fig.suptitle("RHi residuals distribution near ISSR threshold", y=1.02);
```

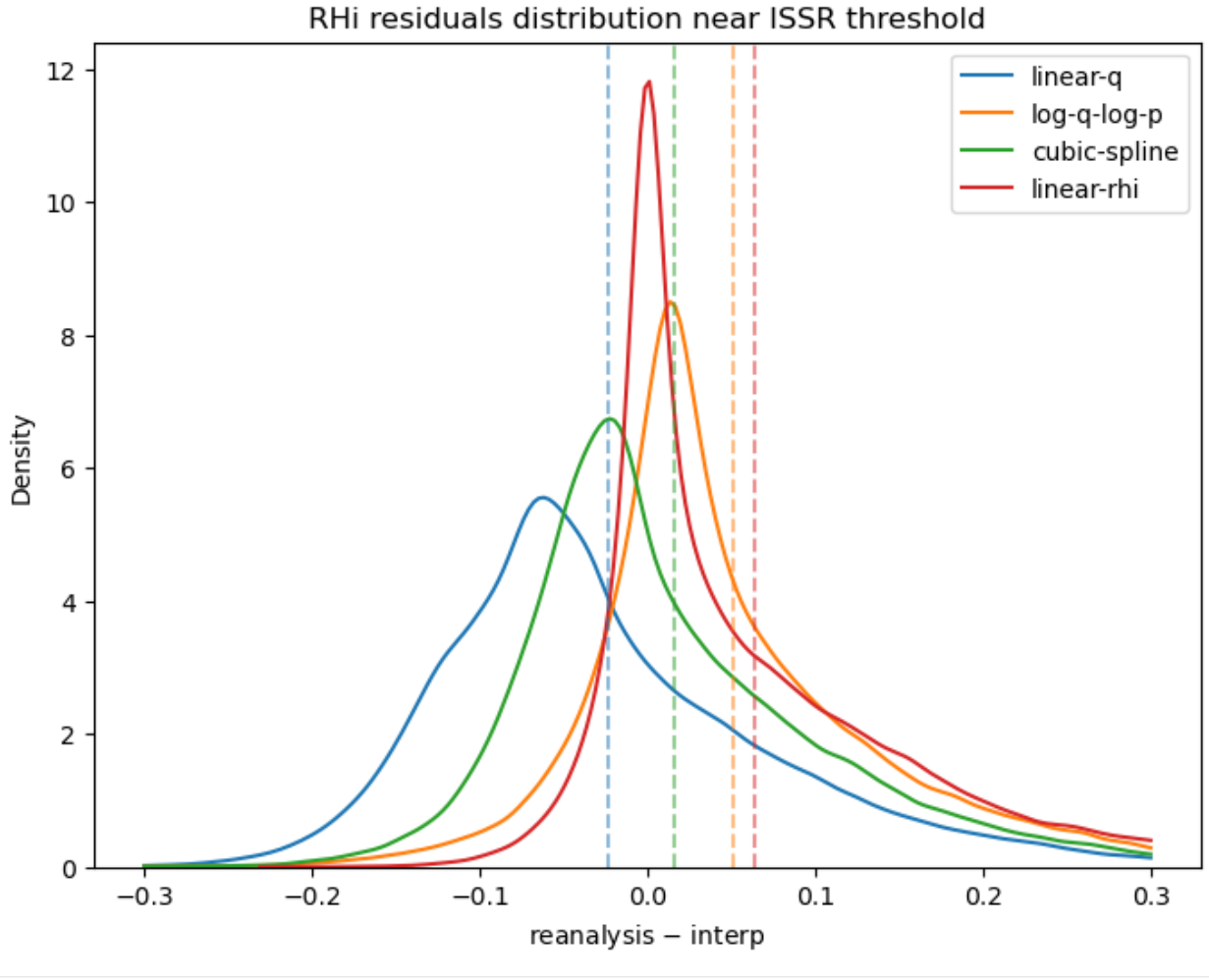


[15]:
```
fig, ax = plt.subplots(figsize=(8, 6))

clip = -0.3, 0.3

sns.kdeplot((rhi_true - linear_q).values[cond], clip=clip, ax=ax, label="linear-q")
sns.kdeplot((rhi_true - log_q_log_p).values[cond], clip=clip, ax=ax, label="log-q-log-p")
sns.kdeplot((rhi_true - cubic_spline).values[cond], clip=clip, ax=ax, label="cubic-spline
↪")
sns.kdeplot((rhi_true - linear_rhi).values[cond], clip=clip, ax=ax, label="linear-rhi")

ax.axvline((rhi_true - linear_q).values[cond].mean(), color="C0", linestyle="--",␣
↪alpha=0.5)
```

(continues on next page)

(continued from previous page)

```
ax.axvline((rhi_true - log_q_log_p).values[cond].mean(), color="C1", linestyle="--",␣
↪alpha=0.5)
ax.axvline((rhi_true - cubic_spline).values[cond].mean(), color="C2", linestyle="--",␣
↪alpha=0.5)
ax.axvline((rhi_true - linear_rhi).values[cond].mean(), color="C3", linestyle="--",␣
↪alpha=0.5)

ax.set_title("RHi residuals distribution near ISSR threshold")
ax.set_xlabel("reanalysis $-$ interp")
ax.legend();
```



10.5.3 Print error metrics

[16]:
```
print(f"MSE linear-q near ISSR:     {np.mean((rhi_true - linear_q).values[cond] ** 2):.
↪4f}")
print(f"MSE log-q-log-p near ISSR:  {np.mean((rhi_true - log_q_log_p).values[cond] ** 2):
↪.4f}")
print(f"MSE cubic-spline near ISSR: {np.mean((rhi_true - cubic_spline).values[cond] **␣
↪2):.4f}")
print(f"MSE linear-rhi near ISSR:   {np.mean((rhi_true - linear_rhi).values[cond] ** 2):.
↪4f}")
print("---")
print(f"MAE linear-q near ISSR:     {np.mean(np.abs(rhi_true - linear_q).values[cond]):.
↪4f}")
print(f"MAE log-q-log-p near ISSR:  {np.mean(np.abs(rhi_true - log_q_log_p).
↪values[cond]):.4f}")
print(f"MAE cubic-spline near ISSR: {np.mean(np.abs(rhi_true - cubic_spline).
↪values[cond]):.4f}")
print(f"MAE linear-rhi near ISSR:   {np.mean(np.abs(rhi_true - linear_rhi).values[cond]):
↪.4f}")
```

```
MSE linear-q near ISSR:     0.0112
MSE log-q-log-p near ISSR:  0.0110
MSE cubic-spline near ISSR: 0.0092
MSE linear-rhi near ISSR:   0.0138
---
MAE linear-q near ISSR:     0.0848
MAE log-q-log-p near ISSR:  0.0722
MAE cubic-spline near ISSR: 0.0697
MAE linear-rhi near ISSR:   0.0753
```

10.6 Conclusions

- When working with gridded met data, use a finer vertical resolution if possible. Using pressure levels spaced at 10 hPa or less at cruising altitudes is recommended for running CoCiP with high precision.
- Linear interpolation in specific humidity appears to introduce bias when working with gridded met data having coarse pressure levels. This bias can be reduced by using more pressure levels.
- Interpolating in the log-log domain appears to be a good solution. An experimental implementation of this is available in `pycontrails` by setting the model parameter `interpolation_q_method="log-q-log-p"`.
- Interpolating with a monotonic cubic spline transformation of the lapse rate also appears to be a good solution. Similarly, this is experimentally available in `pycontrails` by setting a model parameter `interpolation_q_method="cubic-spline"`.
- Interpolating in the gridded RHi space is another possible solution, but it also introduces bias. This is not directly available in `pycontrails`, but could be implemented by the user.

CHAPTER

ELEVEN

API REFERENCE

11.1 Data

11.1.1 Meteorology

MetDataset(data[, cachestore, ...])	Meteorological dataset with multiple variables.
MetDataArray(data[, cachestore, ...])	Meteorological DataArray of single variable.

pycontrails.MetDataset

class pycontrails.**MetDataset**(*data*, *cachestore=None*, *wrap_longitude=False*, *copy=True*, *attrs=None*, ***attrs_kwargs*)

Bases: *MetBase*

Meteorological dataset with multiple variables.

Composition around xr.Dataset to enforce certain variables and dimensions for internal usage

Parameters

- **data** (`xarray.Dataset`) – `xarray.Dataset` containing meteorological variables and coordinates
- **cachestore** (`CacheStore`, *optional*) – Cache datastore for staging intermediates with *save()*. Defaults to None.
- **wrap_longitude** (`bool`, *optional*) – Wrap data along the longitude dimension. If True, duplicate and shift longitude values (ie, -180 -> 180) to ensure that the longitude dimension covers the entire interval `[-180, 180]`. Defaults to False.
- **copy** (`bool`, *optional*) – Copy data on construction. Defaults to True.
- **attrs** (`dict[str, Any]`, *optional*) – Attributes to add to `data.attrs`. Defaults to None. Generally, pycontrails `Models` may use the following attributes:
  - `provider`: Name of the data provider (e.g. "ECMWF").
  - `dataset`: Name of the dataset (e.g. "ERA5").
  - `product`: Name of the product type (e.g. "reanalysis").
- ****attrs_kwargs** (`Any`) – Keyword arguments to add to `data.attrs`. Defaults to None.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import xarray as xr
>>> from pycontrails.datalib.ecmwf import ERA5
```

```
>>> time = ("2022-03-01T00", "2022-03-01T02")
>>> variables = ["air_temperature", "specific_humidity"]
>>> pressure_levels = [200, 250, 300]
>>> era5 = ERA5(time, variables, pressure_levels)
```

```
>>> # Open directly as `MetDataset`
>>> met = era5.open_metdataset()
>>> # Use `data` attribute to access `xarray` object
>>> assert isinstance(met.data, xr.Dataset)
```

```
>>> # Alternatively, open with `xarray` and cast to `MetDataset`
>>> ds = xr.open_mfdataset(era5._cachepaths)
>>> met = MetDataset(ds)
```

```
>>> # Access sub-`DataArrays`
>>> mda = met["t"]  # `MetDataArray` instance, needed for interpolation operations
>>> da = mda.data  # Underlying `xarray` object
```

```
>>> # Check out a few values
>>> da[5:8, 5:8, 1, 1].values
array([[224.08959005, 224.41374427, 224.75945349],
       [224.09456429, 224.42037658, 224.76525676],
       [224.10036756, 224.42617985, 224.77106004]])
```

```
>>> # Mean temperature over entire array
>>> da.mean().load().item()
223.5083
```

__init__(*data*, *cachestore=None*, *wrap_longitude=False*, *copy=True*, *attrs=None*, ***attrs_kwargs*)

Methods

__init__(data[, cachestore, wrap_longitude, ...])	
broadcast_coords(name)	Broadcast coordinates along other dimensions.
copy()	Create a copy of the current class.
downselect(bbox)	Downselect met data within spatial bounding box.
downselect_met(met, *[, longitude_buffer, ...])	Downselect `met` to encompass a spatiotemporal region of the data.
ensure_vars(vars[, raise_error])	Ensure variables exist in xr.Dataset.
from_coords(longitude, latitude, level, time)	Create a *MetDataset* containing a coordinate skeleton from coordinate arrays.
from_zarr(store, **kwargs)	Create a *MetDataset* from a path to a Zarr store.
get(key[, default_value])	Shortcut to `data.get(k, v)()` method.
load(hash[, cachestore, chunks])	Load saved intermediate from `cachestore`.
save(**kwargs)	Save intermediate to `cachestore` as netcdf.
standardize_variables(variables)	Standardize variables **in-place**.
to_vector([transfer_attrs])	Convert a *MetDataset* to a *GeoVectorDataset* by raveling data.
update([other])	Shortcut to `data.update()`.
wrap_longitude()	Wrap longitude coordinates.

Attributes

attrs	Pass through to `self.data.attrs`.
coords	Get coordinates of underlying *data* coordinates.
dataset_attr	Look up the 'dataset' attribute with a custom error message.
dim_order	Default dimension order for DataArray or Dataset (x, y, z, t)
hash	Generate a unique hash for this met instance.
indexes	Low level access to underlying *data* indexes.
is_single_level	Check if instance contains "single level" or "surface level" data.
is_wrapped	Check if the longitude dimension covers the closed interval `[-180, 180]`.
is_zarr	Check if underlying *data* is sourced from a Zarr group.
product_attr	Look up the 'product' attribute with a custom error message.
provider_attr	Look up the 'provider' attribute with a custom error message.
shape	Return the shape of the dimensions.
size	Return the size of (each) array in underlying *data*.
variables	See `indexes`.
data	DataArray or Dataset
cachestore	Cache datastore to use for *save()* or *load()*

broadcast_coords(*name*)

Broadcast coordinates along other dimensions.

Parameters
name (str) – Coordinate/dimension name to broadcast. Can be a dimension or non-dimension coordinates.

Returns
xarray.DataArray – DataArray of the coordinate broadcasted along all other dimensions. The DataArray will have the same shape as the gridded data.

`copy()`

Create a copy of the current class.

Returns
MetDataset – MetDataset copy

`data`

DataArray or Dataset

`property dataset_attr`

Look up the 'dataset' attribute with a custom error message.

Returns
str – Dataset of the data. If not one of 'ERA5', 'HRES', 'IFS', or 'GFS', a warning is issued.

`downselect(`*bbox*`)`

Downselect met data within spatial bounding box.

Parameters
bbox (list[float]) – List of coordinates defining a spatial bounding box in WGS84 coordinates. For 2D queries, list is [west, south, east, north]. For 3D queries, list is [west, south, min-level, east, north, max-level] with level defined in $[hPa]$.

Returns
MetBase – Return downselected data

`ensure_vars(`*vars*, *raise_error=True*`)`

Ensure variables exist in xr.Dataset.

Parameters

- **vars** (MetVariable | str | Sequence[MetVariable | str | list[MetVariable]]) – List of MetVariable (or string key), or individual MetVariable (or string key). If vars contains an element with a list[MetVariable], then only one variable in the list must be present in dataset.
- **raise_error** (bool, *optional*) – Raise KeyError if data does not contain variables. Defaults to True.

Returns
list[str] – List of met keys verified in MetDataset. Returns an empty list if any MetVariable is missing.

Raises
KeyError – Raises when dataset does not contain variable in vars

`classmethod from_coords(`*longitude*, *latitude*, *level*, *time*`)`

Create a *MetDataset* containing a coordinate skeleton from coordinate arrays.

Parameters

- **longitude, latitude** (npt.ArrayLike | float) – Horizontal coordinates, in [deg]
- **level** (npt.ArrayLike | float) – Vertical coordinate, in $[hPa]$

- **time** (`npt.ArrayLike | np.datetime64,`) – Temporal coordinates, in $[UTC]$. Will be sorted.

Returns

MetDataset – MetDataset with no variables.

Examples

```
>>> # Create skeleton MetDataset
>>> longitude = np.arange(0, 10, 0.5)
>>> latitude = np.arange(0, 10, 0.5)
>>> level = [250, 300]
>>> time = np.datetime64("2019-01-01")
>>> met = MetDataset.from_coords(longitude, latitude, level, time)
>>> met
MetDataset with data:
<xarray.Dataset> Size: 360B
Dimensions:       (longitude: 20, latitude: 20, level: 2, time: 1)
Coordinates:
  * longitude     (longitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * latitude      (latitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * level         (level) float64 16B 250.0 300.0
  * time          (time) datetime64[ns] 8B 2019-01-01
    air_pressure  (level) float32 8B 2.5e+04 3e+04
    altitude      (level) float32 8B 1.036e+04 9.164e+03
Data variables:
    *empty*
```

```
>>> met.shape
(20, 20, 2, 1)
```

```
>>> met.size
800
```

```
>>> # Fill it up with some constant data
>>> met["temperature"] = xr.DataArray(np.full(met.shape, 234.5), coords=met.
↪coords)
>>> met["humidity"] = xr.DataArray(np.full(met.shape, 0.5), coords=met.coords)
>>> met
MetDataset with data:
<xarray.Dataset> Size: 13kB
Dimensions:       (longitude: 20, latitude: 20, level: 2, time: 1)
Coordinates:
  * longitude     (longitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * latitude      (latitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * level         (level) float64 16B 250.0 300.0
  * time          (time) datetime64[ns] 8B 2019-01-01
    air_pressure  (level) float32 8B 2.5e+04 3e+04
    altitude      (level) float32 8B 1.036e+04 9.164e+03
Data variables:
    temperature   (longitude, latitude, level, time) float64 6kB 234.5 ... 234.5
    humidity      (longitude, latitude, level, time) float64 6kB 0.5 0.5 ... 0.5
```

```
>>> # Convert to a GeoVectorDataset
>>> vector = met.to_vector()
>>> vector.dataframe.head()
longitude  latitude  level       time  temperature  humidity
0        0.0       0.0  250.0 2019-01-01        234.5       0.5
1        0.0       0.0  300.0 2019-01-01        234.5       0.5
2        0.0       0.5  250.0 2019-01-01        234.5       0.5
3        0.0       0.5  300.0 2019-01-01        234.5       0.5
4        0.0       1.0  250.0 2019-01-01        234.5       0.5
```

classmethod from_zarr(*store*, ***kwargs*)

Create a *MetDataset* from a path to a Zarr store.

Parameters

- **store** (Any) – Path to Zarr store. Passed into xarray.open_zarr().
- ****kwargs** (Any) – Other keyword only arguments passed into xarray.open_zarr().

Returns

MetDataset – MetDataset with data from Zarr store.

get(*key*, *default_value=None*)

Shortcut to data.get(k, v)() method.

Parameters

- **key** (str) – Key to get from *data*
- **default_value** (Any, *optional*) – Return *default_value* if *key* not in *data*, by default *None*

Returns

Any – Values returned from data.get(key, default_value)

classmethod load(*hash*, *cachestore=None*, *chunks=None*)

Load saved intermediate from cachestore.

Parameters

- **hash** (str) – Saved hash to load.
- **cachestore** (CacheStore, *optional*) – Cache datastore to use for sourcing files. Defaults to DiskCacheStore.
- **chunks** (dict[str: int], *optional*) – Chunks kwarg passed to xarray.open_mfdataset() when opening files.

Returns

MetDataset – New MetDataArray with loaded data.

property product_attr

Look up the 'product' attribute with a custom error message.

Returns

str – Product of the data. If not one of 'forecast', 'ensemble', or 'reanalysis', a warning is issued.

property provider_attr

Look up the 'provider' attribute with a custom error message.

Returns

str – Provider of the data. If not one of 'ECMWF' or 'NCEP', a warning is issued.

save(***kwargs*)

Save intermediate to `cachestore` as netcdf.

Load and restore using *load()*.

Parameters
****kwargs** (`Any`) – Keyword arguments passed directly to `xarray.Dataset.to_netcdf()`

Returns
`list[str]` – Returns filenames saved

property shape

Return the shape of the dimensions.

Returns
`tuple[int, int, int, int]` – Shape of underlying data

property size

Return the size of (each) array in underlying *data*.

Returns
`int` – Total number of grid points in underlying data

standardize_variables(*variables*)

Standardize variables **in-place**.

Parameters
variables (`Iterable[MetVariable]`) – Data source variables

See also:

standardize_variables()

to_vector(*transfer_attrs=True*)

Convert a *MetDataset* to a *GeoVectorDataset* by raveling data.

If *data* is lazy, it will be loaded.

Parameters
transfer_attrs (`bool`, *optional*) – Transfer attributes from *data* to output *GeoVectorDataset*. By default, True, meaning that attributes are transferred.

Returns
GeoVectorDataset – Converted *GeoVectorDataset*. The variables on the returned instance include all of those on the input instance, plus the four core spatial temporal variables.

Examples

```
>>> from pycontrails.datalib.ecmwf import ERA5
>>> times = "2022-03-01",  "2022-03-01T01"
>>> variables = ["air_temperature", "specific_humidity"]
>>> levels = [250, 200]
>>> era5 = ERA5(time=times, variables=variables, pressure_levels=levels)
>>> met = era5.open_metdataset()
>>> met.to_vector(transfer_attrs=False)
GeoVectorDataset [6 keys x 4152960 length, 1 attributes]
    Keys: longitude, latitude, level, time, air_temperature, ..., specific_
↪humidity
```

(continues on next page)

(continued from previous page)

```
    Attributes:
    time                [2022-03-01 00:00:00, 2022-03-01 01:00:00]
    longitude           [-180.0, 179.75]
    latitude            [-90.0, 90.0]
    altitude            [10362.8, 11783.9]
    crs                 EPSG:4326
```

update(*other=None*, ***kwargs*)

Shortcut to `data.update()`.

See `xarray.Dataset.update()` for reference.

Parameters

- **other** (`MutableMapping`) – Variables with which to update this dataset
- ****kwargs** (`Any`) – Variables defined by keyword arguments. If a variable exists both in `other` and as a keyword argument, the keyword argument takes precedence.

See also:

-

meth:*xarray.Dataset.update*

wrap_longitude()

Wrap longitude coordinates.

Returns

MetDataset – Copy of MetDataset with wrapped longitude values. Returns copy of current MetDataset when longitude values are already wrapped

pycontrails.MetDataArray

class pycontrails.**MetDataArray**(*data*, *cachestore=None*, *wrap_longitude=False*, *copy=True*, *validate=True*, *name=None*, ***kwargs*)

Bases: *MetBase*

Meteorological DataArray of single variable.

Wrapper around xr.DataArray to enforce certain variables and dimensions for internal usage.

Parameters

- **data** (*ArrayLike*) – xr.DataArray or other array-like data source. When array-like input is provided, input `**kwargs` passed directly to xr.DataArray constructor.
- **cachestore** (`CacheStore`, *optional*) – Cache datastore for staging intermediates with *save()*. Defaults to DiskCacheStore.
- **wrap_longitude** (`bool`, *optional*) – Wrap data along the longitude dimension. If True, duplicate and shift longitude values (ie, -180 -> 180) to ensure that the longitude dimension covers the entire interval `[-180, 180]`. Defaults to False.
- **copy** (`bool`, *optional*) – Copy *data* parameter on construction, by default *True*. If *data* is lazy-loaded via *dask*, this parameter has no effect. If *data* is already loaded into memory, a copy of the data (rather than a view) may be created if *True*.

- **validate** (bool, *optional*) – Confirm that the parameter *data* has correct specification. This automatically handled in the case that *copy=True*. Validation only introduces a very small overhead. This parameter should only be set to *False* if working with data derived from an existing MetDataset or :class`MetDataArray`. By default *True*.
- **name** (Hashable, *optional*) – Name of the data variable. If not specified, the name will be set to "met".
- ****kwargs** – To be removed in future versions. Passed directly to xr.DataArray constructor.

Examples

```
>>> import numpy as np
>>> import xarray as xr
>>> rng = np.random.default_rng(seed=456)
```

```
>>> # Cook up random xarray object
>>> coords = {
...     "longitude": np.arange(-20, 20),
...     "latitude": np.arange(-30, 30),
...     "level": [220, 240, 260, 280],
...     "time": [np.datetime64("2021-08-01T12", "ns"), np.datetime64("2021-08-01T16
↪", "ns")]
...     }
>>> da = xr.DataArray(rng.random((40, 60, 4, 2)), dims=coords.keys(), coords=coords)
```

```
>>> # Cast to `MetDataArray` in order to interpolate
>>> from pycontrails import MetDataArray
>>> mda = MetDataArray(da)
>>> mda.interpolate(-11.4, 5.7, 234, np.datetime64("2021-08-01T13"))
array([0.52358215])
```

```
>>> mda.interpolate(-11.4, 5.7, 234, np.datetime64("2021-08-01T13"), method='nearest
↪')
array([0.4188465])
```

```
>>> da.sel(longitude=-11, latitude=6, level=240, time=np.datetime64("2021-08-01T12
↪")).item()
0.41884649899766946
```

__init__(*data*, *cachestore=None*, *wrap_longitude=False*, *copy=True*, *validate=True*, *name=None*, ***kwargs*)

Methods

__init__(data[, cachestore, wrap_longitude, ...])	
broadcast_coords(name)	Broadcast coordinates along other dimensions.
copy()	Create a copy of the current class.
downselect(bbox)	Downselect met data within spatial bounding box.
downselect_met(met, *[, longitude_buffer, ...])	Downselect `met` to encompass a spatiotemporal region of the data.
find_edges()	Find edges of regions.
interpolate(longitude, latitude, level, time, *)	Interpolate values over underlying DataArray.
load(hash[, cachestore, chunks])	Load saved intermediate from `cachestore`.
save(**kwargs)	Save intermediate to `cachestore` as netcdf.
to_polygon_feature([level, time, ...])	Create GeoJSON Feature artifact from spatial array on a single level and time slice.
to_polygon_feature_collection([time, ...])	Create GeoJSON FeatureCollection artifact from spatial array at time slice.
to_polyhedra(*[, time, iso_value, ...])	Create a collection of polyhedra from spatial array corresponding to a single time slice.
wrap_longitude()	Wrap longitude coordinates.

Attributes

attrs	Pass through to `self.data.attrs`.
binary	Determine if all data is a binary value (0, 1).
coords	Get coordinates of underlying *data* coordinates.
dim_order	Default dimension order for DataArray or Dataset (x, y, z, t)
hash	Generate a unique hash for this met instance.
in_memory	Check if underlying *data* is loaded into memory.
indexes	Low level access to underlying *data* indexes.
is_single_level	Check if instance contains "single level" or "surface level" data.
is_wrapped	Check if the longitude dimension covers the closed interval `[-180, 180]`.
is_zarr	Check if underlying *data* is sourced from a Zarr group.
name	Return the DataArray name.
proportion	Compute proportion of points with value 1.
shape	Return the shape of the dimensions.
size	Return the size of (each) array in underlying *data*.
values	Return underlying numpy array.
variables	See `indexes`.
data	DataArray or Dataset
cachestore	Cache datastore to use for *save()* or *load()*

property binary

Determine if all data is a binary value (0, 1).

Returns

bool – True if all data values are binary value (0, 1)

broadcast_coords(*name*)

Broadcast coordinates along other dimensions.

Parameters
name (str) – Coordinate/dimension name to broadcast. Can be a dimension or non-dimension coordinates.

Returns
xarray.DataArray – DataArray of the coordinate broadcasted along all other dimensions. The DataArray will have the same shape as the gridded data.

copy()

Create a copy of the current class.

Returns
MetDataArray – MetDataArray copy

data

DataArray or Dataset

downselect(*bbox*)

Downselect met data within spatial bounding box.

Parameters
bbox (list[float]) – List of coordinates defining a spatial bounding box in WGS84 coordinates. For 2D queries, list is [west, south, east, north]. For 3D queries, list is [west, south, min-level, east, north, max-level] with level defined in [hPa].

Returns
MetBase – Return downselected data

find_edges()

Find edges of regions.

Returns
MetDataArray – MetDataArray with a binary field, 1 on the edge of the regions, 0 outside and inside the regions.

Raises
NotImplementedError – If the instance is not binary.

property in_memory

Check if underlying *data* is loaded into memory.

This method uses protected attributes of underlying *xarray* objects, and may be subject to deprecation.

Changed in version 0.26.0: Rename from is_loaded to in_memory.

Returns
bool – If underlying data exists as an *np.ndarray* in memory.

interpolate(*longitude*, *latitude*, *level*, *time*, *, *method='linear'*, *bounds_error=False*, *fill_value=nan*, *localize=False*, *indices=None*, *return_indices=False*)

Interpolate values over underlying DataArray.

Zero dimensional coordinates are reshaped to 1D arrays.

Method automatically loads underlying *data* into memory.

If method == "nearest", the out array will have the same dtype as the underlying *data*.

If method == "linear", the out array will be promoted to the most precise dtype of:

- underlying *data*
- `data.longitude`
- `data.latitude`
- `data.level`
- `longitude`
- `latitude`

New in version 0.24: This method can now handle singleton dimensions with `method == "linear"`. Previously these degenerate dimensions caused nan values to be returned.

Parameters

- **longitude** (`float | npt.NDArray[np.float64]`) – Longitude values to interpolate. Assumed to be 0 or 1 dimensional.
- **latitude** (`float | npt.NDArray[np.float64]`) – Latitude values to interpolate. Assumed to be 0 or 1 dimensional.
- **level** (`float | npt.NDArray[np.float64]`) – Level values to interpolate. Assumed to be 0 or 1 dimensional.
- **time** (`np.datetime64 | npt.NDArray[np.datetime64]`) – Time values to interpolate. Assumed to be 0 or 1 dimensional.
- **method** (`str`, *optional*) – Additional keyword arguments to pass to `scipy.interpolate.RegularGridInterpolator`. Defaults to "linear".
- **bounds_error** (`bool`, *optional*) – Additional keyword arguments to pass to `scipy.interpolate.RegularGridInterpolator`. Defaults to `False`.
- **fill_value** (`float | np.float64`, *optional*) – Additional keyword arguments to pass to `scipy.interpolate.RegularGridInterpolator`. Set to None to extrapolate outside the boundary when `method` is `nearest`. Defaults to `np.nan`.
- **localize** (`bool`, *optional*) – Experimental. If True, downselect gridded data to smallest bounding box containing all points. By default False.
- **indices** (`tuple | None`, *optional*) – Experimental. See `interpolation.interp()`. None by default.
- **return_indices** (`bool`, *optional*) – Experimental. See `interpolation.interp()`. False by default.

Returns

`numpy.ndarray` – Interpolated values

See also:

GeoVectorDataset.intersect_met()

Examples

```
>>> from datetime import datetime
>>> import numpy as np
>>> import pandas as pd
>>> from pycontrails.datalib.ecmwf import ERA5
```

```
>>> times = (datetime(2022, 3, 1, 12),  datetime(2022, 3, 1, 15))
>>> variables = "air_temperature"
>>> levels = [200, 250, 300]
>>> era5 = ERA5(times, variables, levels)
>>> met = era5.open_metdataset()
>>> mda = met["air_temperature"]
```

```
>>> # Interpolation at a grid point agrees with value
>>> mda.interpolate(1, 2, 300, np.datetime64('2022-03-01T14:00'))
array([241.91972984])
```

```
>>> da = mda.data
>>> da.sel(longitude=1, latitude=2, level=300, time=np.datetime64('2022-03-01T14
↪')).item()
241.9197298421629
```

```
>>> # Interpolation off grid
>>> mda.interpolate(1.1, 2.1, 290, np.datetime64('2022-03-01 13:10'))
array([239.83793798])
```

```
>>> # Interpolate along path
>>> longitude = np.linspace(1, 2, 10)
>>> latitude = np.linspace(2, 3, 10)
>>> level = np.linspace(200, 300, 10)
>>> time = pd.date_range("2022-03-01T14", periods=10, freq="5min")
>>> mda.interpolate(longitude, latitude, level, time)
array([220.44347694, 223.08900738, 225.74338924, 228.41642088,
       231.10858599, 233.54857391, 235.71504913, 237.86478872,
       239.99274623, 242.10792167])
```

classmethod load(*hash*, *cachestore=None*, *chunks=None*)

Load saved intermediate from `cachestore`.

Parameters

- **hash** (`str`) – Saved hash to load.
- **cachestore** (`CacheStore`, *optional*) – Cache datastore to use for sourcing files. Defaults to DiskCacheStore.
- **chunks** (`dict[str, int]`, *optional*) – Chunks kwarg passed to `xarray.open_mfdataset()` when opening files.

Returns

MetDataArray – New MetDataArray with loaded data.

property name

Return the DataArray name.

Returns
Hashable – DataArray name

property proportion

Compute proportion of points with value 1.

Returns
float – Proportion of points with value 1

Raises
NotImplementedError – If instance does not contain binary data.

save(***kwargs*)

Save intermediate to cachestore as netcdf.

Load and restore using *load()*.

Parameters
****kwargs** (Any) – Keyword arguments passed directly to xarray.save_mfdataset()

Returns
list[str] – Returns filenames of saved files

property shape

Return the shape of the dimensions.

Returns
tuple[int, int, int, int] – Shape of underlying data

property size

Return the size of (each) array in underlying *data*.

Returns
int – Total number of grid points in underlying data

to_polygon_feature(*level=None*, *time=None*, *fill_value=nan*, *iso_value=None*, *min_area=0.0*, *epsilon=0.0*, *lower_bound=True*, *precision=None*, *interiors=True*, *convex_hull=False*, *include_altitude=False*, *properties=None*)

Create GeoJSON Feature artifact from spatial array on a single level and time slice.

Computed polygons always contain an exterior linear ring as defined by the *GeoJSON Polygon specification <https://www.rfc-editor.org/rfc/rfc7946.html#section-3.1.6>*. Polygons may also contain interior linear rings (holes). This method does not support nesting beyond the GeoJSON specification. See the *pycontrails.core.polygon* for additional polygon support.

Changed in version 0.25.12: Previous implementation include several additional parameters which have been removed:

- The approximate parameter
- An path parameter to save output as JSON
- Passing arbitrary kwargs to skimage.measure.find_contours().

New implementation includes new parameters previously lacking:

- fill_value
- min_area
- include_altitude

Changed in version 0.38.0: Change default value of `epsilon` from 0.15 to 0.

Changed in version 0.41.0: Convert continuous fields to binary fields before computing polygons. The parameters `max_area` and `epsilon` are now expressed in terms of longitude/latitude units instead of pixels.

Parameters

- **level** (`float`, *optional*) – Level slice to create polygons. If the "level" coordinate is length 1, then the single level slice will be selected automatically.
- **time** (`datetime`, *optional*) – Time slice to create polygons. If the "time" coordinate is length 1, then the single time slice will be selected automatically.
- **fill_value** (`float`, *optional*) – Value used for filling missing data and for padding the underlying data array. Set to `np.nan` by default, which ensures that regions with missing data are never included in polygons.
- **iso_value** (`float`, *optional*) – Value in field to create iso-surface. Defaults to the average of the min and max value of the array. (This is the same convention as used by `skimage`.)
- **min_area** (`float`, *optional*) – Minimum area of each polygon. Polygons with area less than `min_area` are not included in the output. The unit of this parameter is in longitude/latitude degrees squared. Set to 0 to omit any polygon filtering based on a minimal area conditional. By default, 0.0.
- **epsilon** (`float`, *optional*) – Control the extent to which the polygon is simplified. A value of 0 does not alter the geometry of the polygon. The unit of this parameter is in longitude/latitude degrees. By default, 0.0.
- **lower_bound** (`bool`, *optional*) – Whether to use `iso_value` as a lower or upper bound on values in polygon interiors. By default, True.
- **precision** (`int`, *optional*) – Number of decimal places to round coordinates to. If None, no rounding is performed.
- **interiors** (`bool`, *optional*) – If True, include interior linear rings (holes) in the output. True by default.
- **convex_hull** (`bool`, *optional*) – EXPERIMENTAL. If True, compute the convex hull of each polygon. Only implemented for depth=1. False by default. A warning is issued if the underlying algorithm fails to make valid polygons after computing the convex hull.
- **include_altitude** (`bool`, *optional*) – If True, include the array altitude $[m]$ as a z-coordinate in the *GeoJSON output <https://www.rfc-editor.org/rfc/rfc7946#section-3.1.1>*. False by default.
- **properties** (`dict`, *optional*) – Additional properties to include in the GeoJSON output. By default, None.

Returns

`dict[str, Any]` – Python representation of GeoJSON Feature with MultiPolygon geometry.

Notes

`Cocip` and `CocipGrid` set some quantities to 0 and other quantities to `np.nan` in regions where no contrails form. When computing polygons from `Cocip` or `CocipGrid` output, take care that the choice of `fill_value` correctly includes or excludes contrail-free regions. See the `Cocip` documentation for details about `np.nan` in model output.

See also:

to_polyhedra(), `pycontrails.core.polygons.find_multipolygons()`

Examples

```
>>> from pprint import pprint
>>> from pycontrails.datalib.ecmwf import ERA5
>>> era5 = ERA5("2022-03-01", variables="air_temperature", pressure_levels=250)
>>> mda = era5.open_metdataset()["air_temperature"]
>>> mda.shape
(1440, 721, 1, 1)
```

```
>>> pprint(mda.to_polygon_feature(iso_value=239.5, precision=2, epsilon=0.1))
{'geometry': {'coordinates': [[[[167.88, -22.5],
                                [167.75, -22.38],
                                [167.62, -22.5],
                                [167.75, -22.62],
                                [167.88, -22.5]]],
                              [[[43.38, -33.5],
                                [43.5, -34.12],
                                [43.62, -33.5],
                                [43.5, -33.38],
                                [43.38, -33.5]]]],
              'type': 'MultiPolygon'},
 'properties': {},
 'type': 'Feature'}
```

to_polygon_feature_collection(*time=None*, *fill_value=nan*, *iso_value=None*, *min_area=0.0*, *epsilon=0.0*, *lower_bound=True*, *precision=None*, *interiors=True*, *convex_hull=False*, *include_altitude=False*, *properties=None*)

Create GeoJSON FeatureCollection artifact from spatial array at time slice.

See the *to_polygon_feature()* method for a description of the parameters.

Returns

`dict[str, Any]` – Python representation of GeoJSON FeatureCollection. This dictionary is comprised of individual GeoJON Features, one per `self.data["level"]`.

to_polyhedra(***, *time=None*, *iso_value=0.0*, *simplify_fraction=1.0*, *lower_bound=True*, *return_type='geojson'*, *path=None*, *altitude_scale=1.0*, *output_vertex_normals=False*, *closed=True*)

Create a collection of polyhedra from spatial array corresponding to a single time slice.

Parameters

- **time** (`datetime`, *optional*) – Time slice to create mesh. If the "time" coordinate is length 1, then the single time slice will be selected automatically.

- **iso_value** (`float`, *optional*) – Value in field to create iso-surface. Defaults to 0.
- **simplify_fraction** (`float`, *optional*) – Apply *open3d simplify_quadric_decimation* method to simplify the polyhedra geometry. This parameter must be in the half-open interval (0.0, 1.0]. Defaults to 1.0, corresponding to no reduction.
- **lower_bound** (`bool`, *optional*) – Whether to use `iso_value` as a lower or upper bound on values in polyhedra interiors. By default, True.
- **return_type** (`str`, *optional*) – Must be one of "geojson" or "mesh". Defaults to "geojson". If "geojson", this method returns a dictionary representation of a geojson MultiPolygon object whose polygons are polyhedra faces. If "mesh", this method returns an *open3d TriangleMesh* instance.
- **path** (`str`, *optional*) – Output geojson or mesh to file. If *return_type* is "mesh", see Open3D File I/O for Mesh for file type options.
- **altitude_scale** (`float`, *optional*) – Rescale the altitude dimension of the mesh, $[m]$
- **output_vertex_normals** (`bool`, *optional*) – If `path` is defined, write out vertex normals. Defaults to False.
- **closed** (`bool`, *optional*) – If True, pad spatial array along all axes to ensure polyhedra are "closed". This flag often gives rise to cleaner visualizations. Defaults to True.

Returns

`dict | :class:`o3d.geometry.TriangleMesh`` – Python representation of geojson object or Open3D Triangle Mesh depending on the *return_type* parameter.

Raises

- `ModuleNotFoundError` – Method requires the *vis* optional dependencies
- `ValueError` – If input parameters are invalid.

See also:

`to_polygons()` skimage.measure.marching_cubes

Notes

Uses the scikit-image Marching Cubes algorithm to reconstruct a surface from the point-cloud like arrays.

property values

Return underlying numpy array.

This methods loads *data* if it is not already in memory.

Returns

`numpy.ndarray` – Underlying numpy array

See also:

-

meth:*xr.Dataset.load*

-

meth:*xr.DataArray.load*

wrap_longitude()

Wrap longitude coordinates.

Returns

MetDataArray – Copy of MetDataArray with wrapped longitude values. Returns copy of current MetDataArray when longitude values are already wrapped

11.1.2 Vector Data

VectorDataset([data, attrs, copy])	Base class to hold 1D arrays of consistent size.
GeoVectorDataset([data, longitude, ...])	Base class to hold 1D geospatial arrays of consistent size.

pycontrails.VectorDataset

class pycontrails.**VectorDataset**(*data=None*, *, *attrs=None*, *copy=True*, ***attrs_kwargs*)

Bases: object

Base class to hold 1D arrays of consistent size.

Parameters

- **data** (dict[str, npt.ArrayLike] | pd.DataFrame | VectorDataDict | VectorDataset | None, *optional*) – Initial data, by default None
- **attrs** (dict[str, Any] | AttrDict, *optional*) – Dictionary of attributes, by default None
- **copy** (bool, *optional*) – Copy data on class creation, by default True
- ****attrs_kwargs** (Any) – Additional attributes passed as keyword arguments

Raises

ValueError – If "time" variable cannot be converted to numpy array.

__init__(*data=None*, *, *attrs=None*, *copy=True*, ***attrs_kwargs*)

Methods

__init__([data, attrs, copy])	
broadcast_attrs(keys[, overwrite, raise_error])	Attach values from `keys` in *attrs* onto *data*.
broadcast_numeric_attrs([ignore_keys, overwrite])	Attach numeric values in *attrs* onto *data*.
copy(**kwargs)	Return a copy of this VectorDatasetType class.
create_empty(keys[, attrs])	Create instance with variables defined by *keys* and size 0.
ensure_vars(vars[, raise_error])	Ensure variables exist in column of *data* or *attrs*.
filter(mask[, copy])	Filter *data* according to a boolean array `mask`.
from_dict(obj[, copy])	Create instance from dict representation containing data and attrs.
generate_splits(n_splits[, copy])	Split instance into `n_split` sub-vectors.
get(key[, default_value])	Get values from *data* with `default_value` if `key` not in *data*.
get_data_or_attr(key[, default])	Get value from *data* or *attrs*.
select(keys[, copy])	Return new class instance only containing specified keys.
setdefault(key[, default])	Shortcut to `VectorDataDict.setdefault()`.
sort(by)	Sort data by key(s).
sum(vectors[, infer_attrs, fill_value])	Sum a list of *VectorDataset* instances.
to_dataframe([copy])	Create `pd.DataFrame` in which each key-value pair in *data* is a column.
to_dict()	Create dictionary with *data* and *attrs*.
update([other])	Update values in *data* dict without warning if overwriting.

Attributes

data	Vector data with labels as keys and `numpy.ndarray` as values
attrs	Generic dataset attributes
dataframe	Shorthand property to access *to_dataframe()* with `copy=False`.
hash	Generate a unique hash for this class instance.
shape	Shape of each array in *data*.
size	Length of each array in *data*.

attrs

Generic dataset attributes

broadcast_attrs(*keys*, *overwrite=False*, *raise_error=True*)

Attach values from `keys` in *attrs* onto *data*.

If possible, use `dtype = np.float32` when broadcasting. If not possible, use whatever `dtype` is inferred from the data by `numpy.full()`.

Parameters

- **keys** (`str | Iterable[str]`) – Keys to broadcast

- **overwrite** (bool, *optional*) – If True, overwrite existing values in *data*. By default False.
- **raise_error** (bool, *optional*) – Raise KeyError if `self.attrs` does not contain some of `keys`.

Raises
KeyError – Not all `keys` found in *attrs*.

broadcast_numeric_attrs(*ignore_keys=None*, *overwrite=False*)

Attach numeric values in *attrs* onto *data*.

Iterate through values in *attrs* and attach `float` and `int` values to `data`.

This method modifies object in place.

Parameters

- **ignore_keys** (`str | Iterable[str]`, *optional*) – Do not broadcast selected keys. Defaults to None.
- **overwrite** (bool, *optional*) – If True, overwrite existing values in *data*. By default False.

copy(***kwargs*)

Return a copy of this VectorDatasetType class.

Parameters
****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
`VectorDatasetType` – Copy of class

classmethod create_empty(*keys*, *attrs=None*, ***attrs_kwargs*)

Create instance with variables defined by *keys* and size 0.

If instance requires additional variables to be defined, these keys will automatically be attached to returned instance.

Parameters

- **keys** (`Iterable[str]`) – Keys to include in empty VectorDataset instance.
- **attrs** (`dict[str, Any] | None`, *optional*) – Attributes to attach instance.
- ****attrs_kwargs** (Any) – Define attributes as keyword arguments.

Returns
`VectorDatasetType` – Empty VectorDataset instance.

data

Vector data with labels as keys and `numpy.ndarray` as values

property dataframe

Shorthand property to access *to_dataframe()* with `copy=False`.

Returns
`pandas.DataFrame` – Equivalent to the output from *to_dataframe()*

ensure_vars(*vars*, *raise_error=True*)

Ensure variables exist in column of *data* or *attrs*.

Parameters

- **vars** (`str | Iterable[str]`) – A single string variable name or a sequence of string variable names.

- **raise_error** (bool, *optional*) – Raise KeyError if data does not contain variables. Defaults to True.

Returns
bool – True if all variables exist. False otherwise.

Raises
KeyError – Raises when dataset does not contain variable in vars

filter(*mask*, *copy=True*, ***kwargs*)

Filter *data* according to a boolean array mask.

Entries corresponding to mask == True are kept.

Parameters

- **mask** (npt.NDArray[np.bool_]) – Boolean array with compatible shape.
- **copy** (bool, *optional*) – Copy data on filter. Defaults to True. See numpy best practices for insight into whether copy is appropriate.
- ****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Containing filtered data

Raises
TypeError – If mask is not a boolean array.

classmethod from_dict(*obj*, *copy=True*, ***obj_kwargs*)

Create instance from dict representation containing data and attrs.

Parameters

- **obj** (dict[str, Any]) – Dict representation of VectorDataset (e.g. *to_dict()*)
- **copy** (bool, *optional*) – Passed to *VectorDataset* constructor. Defaults to True.
- ****obj_kwargs** (Any) – Additional properties passed as keyword arguments.

Returns
VectorDatasetType – VectorDataset instance.

See also:

to_dict()

generate_splits(*n_splits*, *copy=True*)

Split instance into n_split sub-vectors.

Parameters

- **n_splits** (int) – Number of splits.
- **copy** (bool, *optional*) – Passed into *filter()*. Defaults to True. Recommend to keep as True based on numpy best practices.

Returns
Generator[VectorDatasetType, None, None] – Generator of split vectors.

See also:

numpy.array_split()

get(*key*, *default_value=None*)

Get values from *data* with `default_value` if `key` not in *data*.

Parameters

- **key** (`str`) – Key to get from *data*
- **default_value** (`Any`, *optional*) – Return `default_value` if *key* not in *data*, by default `None`

Returns

`Any` – Values at `data[key]` or `default_value`

get_data_or_attr(*key*, *default=<object object>*)

Get value from *data* or *attrs*.

This method first checks if `key` is in *data* and returns the value if so. If `key` is not in *data*, then this method checks if `key` is in *attrs* and returns the value if so. If `key` is not in *data* or *attrs*, then the `default` value is returned if provided. Otherwise a `KeyError` is raised.

Parameters

- **key** (`str`) – Key to get from *data* or *attrs*
- **default** (`Any`, *optional*) – Default value to return if `key` is not in *data* or *attrs*.

Returns

`Any` – Value at `data[key]` or `attrs[key]`

Raises

KeyError – If `key` is not in *data* or *attrs* and `default` is not provided.

Examples

```
>>> vector = VectorDataset({"a": [1, 2, 3]}, attrs={"b": 4})
>>> vector.get_data_or_attr("a")
array([1, 2, 3])
```

```
>>> vector.get_data_or_attr("b")
4
```

```
>>> vector.get_data_or_attr("c")
Traceback (most recent call last):
...
KeyError: "Key 'c' not found in data or attrs."
```

```
>>> vector.get_data_or_attr("c", default=5)
5
```

property hash

Generate a unique hash for this class instance.

Returns

`str` – Unique hash for flight instance (sha1)

select(*keys*, *copy=True*)

Return new class instance only containing specified keys.

Parameters

- **keys** (`Iterable[str]`) – An iterable of keys to filter by.
- **copy** (`bool`, *optional*) – Copy data on selection. Defaults to True.

Returns

VectorDataset – VectorDataset containing only data associated to `keys`. Note that this method always returns a *VectorDataset*, even if the calling class is a proper subclass of *VectorDataset*.

setdefault(*key*, *default=None*)

Shortcut to `VectorDataDict.setdefault()`.

Parameters

- **key** (`str`) – Key in *data* dict.
- **default** (`npt.ArrayLike`, *optional*) – Values to use as default, if key is not defined

Returns

`numpy.ndarray` – Values at `key`

property shape

Shape of each array in *data*.

Returns

`tuple[int]` – Shape of each array in *data*.

property size

Length of each array in *data*.

Returns

`int` – Length of each array in *data*.

sort(*by*)

Sort data by key(s).

This method always creates a copy of the data by calling `pandas.DataFrame.sort_values()`.

Parameters

by (`str | list[str]`) – Key or list of keys to sort by.

Returns

`VectorDatasetType` – Instance with sorted data.

classmethod sum(*vectors*, *infer_attrs=True*, *fill_value=None*)

Sum a list of *VectorDataset* instances.

Parameters

- **vectors** (`Sequence[VectorDataset]`) – List of *VectorDataset* instances to concatenate.
- **infer_attrs** (`bool`, *optional*) – If True, infer attributes from the first element in the sequence.
- **fill_value** (`float`, *optional*) – Fill value to use when concatenating arrays. By default None, which raises an error if incompatible keys are found.

Returns

VectorDataset – Sum of all instances in `vectors`.

Raises

KeyError – If incompatible *data* keys are found among `vectors`.

Examples

```
>>> from pycontrails import VectorDataset
>>> v1 = VectorDataset({"a": [1, 2, 3], "b": [4, 5, 6]})
>>> v2 = VectorDataset({"a": [7, 8, 9], "b": [10, 11, 12]})
>>> v3 = VectorDataset({"a": [13, 14, 15], "b": [16, 17, 18]})
>>> v = VectorDataset.sum([v1, v2, v3])
>>> v.dataframe
    a   b
0   1   4
1   2   5
2   3   6
3   7  10
4   8  11
5   9  12
6  13  16
7  14  17
8  15  18
```

to_dataframe(*copy=True*)

Create `pd.DataFrame` in which each key-value pair in *data* is a column.

DataFrame does **not** copy data by default. Use the `copy` parameter to copy data values on creation.

Parameters

copy (`bool`, *optional*) – Copy data on DataFrame creation.

Returns

`pandas.DataFrame` – DataFrame holding key-values as columns.

to_dict()

Create dictionary with *data* and *attrs*.

If geo-spatial coordinates (e.g. `"latitude"`, `"longitude"`, `"altitude"`) are present, round to a reasonable precision. If a `"time"` variable is present, round to unix seconds. When the instance is a *GeoVectorDataset*, disregard any `"altitude"` or `"level"` coordinate and only include `"altitude_ft"` in the output.

Returns

`dict[str, Any]` – Dictionary with *data* and *attrs*.

See also:

from_dict()

Examples

```
>>> import pprint
>>> from pycontrails import Flight
>>> fl = Flight(
...     longitude=[-100, -110],
...     latitude=[40, 50],
...     level=[200, 200],
...     time=[np.datetime64("2020-01-01T09"), np.datetime64("2020-01-01T09:30
↪")],
...     aircraft_type="B737",
... )
>>> fl = fl.resample_and_fill("5min")
>>> pprint.pprint(fl.to_dict())
{'aircraft_type': 'B737',
 'altitude_ft': [38661.0, 38661.0, 38661.0, 38661.0, 38661.0, 38661.0, 38661.0],
 'crs': 'EPSG:4326',
 'latitude': [40.0, 41.724, 43.428, 45.111, 46.769, 48.399, 50.0],
 'longitude': [-100.0,
               -101.441,
               -102.959,
               -104.563,
               -106.267,
               -108.076,
               -110.0],
 'time': [1577869200,
          1577869500,
          1577869800,
          1577870100,
          1577870400,
          1577870700,
          1577871000]}
```

update(*other=None*, ***kwargs*)

Update values in *data* dict without warning if overwriting.

Parameters

- **other** (`dict[str, npt.ArrayLike] | None`, *optional*) – Fields to update as dict
- ****kwargs** (`npt.ArrayLike`) – Fields to update as kwargs

pycontrails.GeoVectorDataset

class pycontrails.**GeoVectorDataset**(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, ***attrs_kwargs*)

Bases: *VectorDataset*

Base class to hold 1D geospatial arrays of consistent size.

GeoVectorDataset is required to have geospatial coordinate keys defined in *required_keys*.

Expect latitude-longitude CRS in WGS 84. Expect altitude in $[m]$. Expect level in $[hPa]$.

Each spatial variable is expected to have "float32" or "float64" `dtype`. The time variable is expected to have "datetime64[ns]" `dtype`.

Use the attribute `attr["crs"]` to specify coordinate reference system using PROJ or EPSG syntax.

Parameters

- **data** (`dict[str, npt.ArrayLike] | pd.DataFrame | VectorDataDict | VectorDataset | None`, *optional*) – Data dictionary or `pandas.DataFrame` . Must include keys/columns `time`, `latitude`, `longitude`, `altitude` or `level`. Keyword arguments for `time`, `latitude`, `longitude`, `altitude` or `level` override `data` inputs. Expects `altitude` in meters and `time` as a DatetimeLike (or array that can processed with `pd.to_datetime()`). Additional waypoint-specific data can be included as additional keys/columns.
- **longitude** (`npt.ArrayLike`, *optional*) – Longitude data. Defaults to None.
- **latitude** (`npt.ArrayLike`, *optional*) – Latitude data. Defaults to None.
- **altitude** (`npt.ArrayLike`, *optional*) – Altitude data, $[m]$. Defaults to None.
- **altitude_ft** (`npt.ArrayLike`, *optional*) – Altitude data, $[ft]$. Defaults to None.
- **level** (`npt.ArrayLike`, *optional*) – Level data, $[hPa]$. Defaults to None.
- **time** (`npt.ArrayLike`, *optional*) – Time data. Expects an array of DatetimeLike values, or array that can processed with `pd.to_datetime()`. Defaults to None.
- **attrs** (`dict[Hashable, Any] | AttrDict`, *optional*) – Additional properties as a dictionary. Defaults to {}.
- **copy** (`bool`, *optional*) – Copy data on class creation. Defaults to True.
- ****attrs_kwargs** (`Any`) – Additional properties passed as keyword arguments.

Raises

`KeyError` – Raises if `data` input does not contain at least `time`, `latitude`, `longitude`, (`altitude` or `level`).

__init__(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, ***attrs_kwargs*)

Methods

T_isa()	Calculate the ICAO standard atmosphere temperature at each point.
__init__([data, longitude, latitude, ...])	
broadcast_attrs(keys[, overwrite, raise_error])	Attach values from `keys` in `attrs` onto `data`.
broadcast_numeric_attrs([ignore_keys, overwrite])	Attach numeric values in `attrs` onto `data`.
coords_intersect_met(met)	Return boolean mask of data inside the bounding box defined by `met`.
copy(**kwargs)	Return a copy of this VectorDatasetType class.
create_empty([keys, attrs])	Create instance with variables defined by *keys* and size 0.
downselect_met(met, *[, longitude_buffer, ...])	Downselect `met` to encompass a spatiotemporal region of the data.
ensure_vars(vars[, raise_error])	Ensure variables exist in column of `data` or `attrs`.
filter(mask[, copy])	Filter `data` according to a boolean array `mask`.
from_dict(obj[, copy])	Create instance from dict representation containing data and attrs.
generate_splits(n_splits[, copy])	Split instance into `n_split` sub-vectors.
get(key[, default_value])	Get values from `data` with `default_value` if `key` not in `data`.
get_data_or_attr(key[, default])	Get value from `data` or `attrs`.
intersect_met(mda, *[, longitude, latitude, ...])	Intersect waypoints with MetDataArray.
select(keys[, copy])	Return new class instance only containing specified keys.
setdefault(key[, default])	Shortcut to `VectorDataDict.setdefault()`.
sort(by)	Sort data by key(s).
sum(vectors[, infer_attrs, fill_value])	Sum a list of *VectorDataset* instances.
to_dataframe([copy])	Create `pd.DataFrame` in which each key-value pair in `data` is a column.
to_dict()	Create dictionary with `data` and `attrs`.
to_geojson_points()	Return dataset as GeoJSON FeatureCollection of Points.
to_lon_lat_grid(agg, *[, spatial_bbox, ...])	Convert vectors to a longitude-latitude grid.
to_pseudo_mercator([copy])	Convert data from `attrs["crs"]` to Pseudo Mercator (EPSG:3857).
transform_crs(crs[, copy])	Transform trajectory data from one coordinate reference system (CRS) to another.
update([other])	Update values in `data` dict without warning if overwriting.

Attributes

air_pressure	Get `air_pressure` values for points.
altitude	Get altitude.
altitude_ft	Get altitude in feet.
attrs	Generic dataset attributes
constants	Return a dictionary of constant attributes and data values.
coords	Get geospatial coordinates for compatibility with MetDataArray.
data	Vector data with labels as keys and numpy.ndarray as values
dataframe	Shorthand property to access `to_dataframe()` with `copy=False`.
hash	Generate a unique hash for this class instance.
level	Get pressure `level` values for points.
required_keys	Required keys for creating GeoVectorDataset
shape	Shape of each array in `data`.
size	Length of each array in `data`.
vertical_keys	At least one of these vertical-coordinate keys must also be included

T_isa()

Calculate the ICAO standard atmosphere temperature at each point.

Returns
npt.NDArray[np.float64] – ISA temperature, [K]

See also:

pycontrails.physics.units.m_to_T_isa()

property air_pressure

Get `air_pressure` values for points.

Returns
npt.NDArray[np.float64] – Point air pressure values, [Pa]

property altitude

Get altitude.

Automatically calculates altitude using `units.pl_to_m()` using `level` key.

Note that if `altitude` key exists in `data`, the data at the `altitude` key will be returned. This allows an override of the default calculation of altitude from pressure level.

Returns
npt.NDArray[np.float64] – Altitude, [m]

property altitude_ft

Get altitude in feet.

Returns
npt.NDArray[np.float64] – Altitude, [ft]

`property constants`

Return a dictionary of constant attributes and data values.

Includes `attrs` and values from columns in `data` with a unique value.

Returns
`dict[str, Any]` – Properties and their constant values

`property coords`

Get geospatial coordinates for compatibility with MetDataArray.

Returns
`pandas.DataFrame` – `pd.DataFrame` with columns *longitude*, *latitude*, *level*, and *time*.

`coords_intersect_met`(*met*)

Return boolean mask of data inside the bounding box defined by `met`.

Parameters
met (`MetDataset | MetDataArray`) – MetDataset or MetDataArray to compare.

Returns
`npt.NDArray[np.bool_]` – True if point is inside the bounding box defined by `met`.

`classmethod create_empty`(*keys=None*, *attrs=None*, ***attrs_kwargs*)

Create instance with variables defined by *keys* and size 0.

If instance requires additional variables to be defined, these keys will automatically be attached to returned instance.

Parameters

- **keys** (`Iterable[str]`) – Keys to include in empty VectorDataset instance.
- **attrs** (`dict[str, Any] | None`, *optional*) – Attributes to attach instance.
- ****attrs_kwargs** (`Any`) – Define attributes as keyword arguments.

Returns
`VectorDatasetType` – Empty VectorDataset instance.

`downselect_met`(*met*, ***, *longitude_buffer=(0.0, 0.0)*, *latitude_buffer=(0.0, 0.0)*, *level_buffer=(0.0, 0.0)*, *time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *copy=True*)

Downselect `met` to encompass a spatiotemporal region of the data.

Parameters

- **met** (`MetDataset | MetDataArray`) – MetDataset or MetDataArray to downselect.
- **longitude_buffer** (`tuple[float, float]`, *optional*) – Extend longitude domain past by `longitude_buffer[0]` on the low side and `longitude_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` degrees.
- **latitude_buffer** (`tuple[float, float]`, *optional*) – Extend latitude domain past by `latitude_buffer[0]` on the low side and `latitude_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` degrees.
- **level_buffer** (`tuple[float, float]`, *optional*) – Extend level domain past by `level_buffer[0]` on the low side and `level_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` [hPa].
- **time_buffer** (`tuple[np.timedelta64, np.timedelta64]`, *optional*) – Extend time domain past by `time_buffer[0]` on the low side and `time_buffer[1]` on the high side.

Units must be the same as class coordinates. Defaults to `(np.timedelta64(0, "h"), np.timedelta64(0, "h"))`.

- **copy** (`bool`) – If returned object is a copy or view of the original. True by default.

Returns

`MetDataset | MetDataArray` – Copy of downselected MetDataset or MetDataArray.

intersect_met(*mda*, *, *longitude=None*, *latitude=None*, *level=None*, *time=None*, *use_indices=False*, ***interp_kwargs*)

Intersect waypoints with MetDataArray.

Parameters

- **mda** (*MetDataArray*) – MetDataArray containing a meteorological variable at spatio-temporal coordinates.
- **longitude** (`npt.NDArray[np.float64]`, *optional*) – Override existing coordinates for met interpolation
- **latitude** (`npt.NDArray[np.float64]`, *optional*) – Override existing coordinates for met interpolation
- **level** (`npt.NDArray[np.float64]`, *optional*) – Override existing coordinates for met interpolation
- **time** (`npt.NDArray[np.datetime64]`, *optional*) – Override existing coordinates for met interpolation
- **use_indices** (`bool`, *optional*) – Experimental.
- ****interp_kwargs** (`Any`) – Additional keyword arguments to pass to `MetDataArray.intersect_met()`. Examples include `method`, `bounds_error`, and `fill_value`. If an error such as

```
ValueError: One of the requested xi is out of bounds in dimension 2
```

occurs, try calling this function with `bounds_error=False`. In addition, setting `fill_value=0.0` will replace NaN values with 0.0.

Returns

`npt.NDArray[np.float64]` – Interpolated values

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
>>> import numpy as np
>>> from pycontrails.datalib.ecmwf import ERA5
>>> from pycontrails import Flight
```

```
>>> # Get met data
>>> times = (datetime(2022, 3, 1, 0),  datetime(2022, 3, 1, 3))
>>> variables = ["air_temperature", "specific_humidity"]
>>> levels = [300, 250, 200]
>>> era5 = ERA5(time=times, variables=variables, pressure_levels=levels)
>>> met = era5.open_metdataset()
```

```
>>> # Example flight
>>> df = pd.DataFrame()
>>> df['longitude'] = np.linspace(0, 50, 10)
>>> df['latitude'] = np.linspace(0, 10, 10)
>>> df['altitude'] = 11000
>>> df['time'] = pd.date_range("2022-03-01T00", "2022-03-01T02", periods=10)
>>> fl = Flight(df)
```

```
>>> # Intersect
>>> fl.intersect_met(met['air_temperature'], method='nearest')
array([231.62969892, 230.72604651, 232.24318771, 231.88338483,
       231.06429438, 231.59073409, 231.65125393, 231.93064004,
       232.03344087, 231.65954432])
```

```
>>> fl.intersect_met(met['air_temperature'], method='linear')
array([225.77794552, 225.13908414, 226.231218  , 226.31831528,
       225.56102321, 225.81192149, 226.03192642, 226.22056121,
       226.03770174, 225.63226188])
```

```
>>> # Interpolate and attach to `Flight` instance
>>> for key in met:
...     fl[key] = fl.intersect_met(met[key])
```

```
>>> # Show the final three columns of the dataframe
>>> fl.dataframe.iloc[:, -3:].head()
                 time  air_temperature  specific_humidity
0 2022-03-01 00:00:00       225.777946           0.000132
1 2022-03-01 00:13:20       225.139084           0.000132
2 2022-03-01 00:26:40       226.231218           0.000107
3 2022-03-01 00:40:00       226.318315           0.000171
4 2022-03-01 00:53:20       225.561022           0.000109
```

property level

Get pressure `level` values for points.

Automatically calculates pressure level using `units.m_to_pl()` using `altitude` key.

Note that if `level` key exists in `data`, the data at the `level` key will be returned. This allows an override of the default calculation of pressure level from altitude.

Returns

`npt.NDArray[np.float64]` – Point pressure level values, $[hPa]$

required_keys = ('longitude', 'latitude', 'time')

Required keys for creating GeoVectorDataset

to_geojson_points()

Return dataset as GeoJSON FeatureCollection of Points.

Each Feature has a properties attribute that includes `time` and other data besides `latitude`, `longitude`, and `altitude` in `data`.

Returns

`dict[str, Any]` – Python representation of GeoJSON FeatureCollection

to_lon_lat_grid(*agg*, ***, *spatial_bbox=(-180.0, -90.0, 180.0, 90.0)*, *spatial_grid_res=0.5*)

Convert vectors to a longitude-latitude grid.

See also:

`vector_to_lon_lat_grid`

to_pseudo_mercator(*copy=True*)

Convert data from `attrs["crs"]` to Pseudo Mercator (EPSG:3857).

Parameters
copy (`bool`, *optional*) – Copy data on transformation. Defaults to True.

Returns
`GeoVectorDatasetType`

transform_crs(*crs*, *copy=True*)

Transform trajectory data from one coordinate reference system (CRS) to another.

Parameters

- **crs** (`str`) – Target CRS. Passed into to `pyproj.Transformer`. The source CRS is inferred from the `attrs["crs"]` attribute.
- **copy** (`bool`, *optional*) – Copy data on transformation. Defaults to True.

Returns
`GeoVectorDatasetType` – Converted dataset with new coordinate reference system. `attrs["crs"]` reflects new crs.

vertical_keys = ('altitude', 'level', 'altitude_ft')

At least one of these vertical-coordinate keys must also be included

11.1.3 Flight & Aircraft

Flight([data, longitude, latitude, ...])	A single flight trajectory.
Fleet([data, longitude, latitude, altitude, ...])	Data structure for holding a sequence of *Flight* instances.
FlightPhase(value[, names, module, ...])	Flight phase enumeration.
Fuel(fuel_name, q_fuel, hydrogen_content, ...)	Base class for the physical parameters of the fuel.
JetA([fuel_name, q_fuel, hydrogen_content, ...])	Jet A-1 Fuel.
SAFBlend(pct_blend)	Jet A-1 / Sustainable Aviation Fuel Blend.
HydrogenFuel([fuel_name, q_fuel, ...])	Hydrogen Fuel.

pycontrails.Flight

class pycontrails.**Flight**(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, *fuel=None*, *drop_duplicated_times=False*, ***attrs_kwargs*)

Bases: *GeoVectorDataset*

A single flight trajectory.

Expect latitude-longitude coordinates in WGS 84. Expect altitude in $[m]$. Expect pressure level (*level*) in $[hPa]$.

Use the attribute `attrs["crs"]` to specify coordinate reference system using PROJ or EPSG syntax.

Parameters

- **data** (dict[str, np.ndarray] | pd.DataFrame | VectorDataDict | VectorDataset | None) – Flight trajectory waypoints as data dictionary or pandas.DataFrame. Must include columns time, latitude, longitude, altitude or level. Keyword arguments for time, latitude, longitude, altitude or level will override data inputs. Expects altitude in meters and time as a DatetimeLike (or array that can processed with pd.to_datetime()). Additional waypoint-specific data can be included as additional keys/columns.
- **longitude** (npt.ArrayLike, *optional*) – Flight trajectory waypoint longitude. Defaults to None.
- **latitude** (npt.ArrayLike, *optional*) – Flight trajectory waypoint latitude. Defaults to None.
- **altitude** (npt.ArrayLike, *optional*) – Flight trajectory waypoint altitude, $[m]$. Defaults to None.
- **altitude_ft** (npt.ArrayLike, *optional*) – Flight trajectory waypoint altitude, $[ft]$.
- **level** (npt.ArrayLike, *optional*) – Flight trajectory waypoint pressure level, $[hPa]$. Defaults to None.
- **time** (npt.ArrayLike, *optional*) – Flight trajectory waypoint time. Defaults to None.
- **attrs** (dict[str, Any], *optional*) – Additional flight properties as a dictionary. While different models may utilize Flight attributes differently, pycontrails applies the following conventions:
  - flight_id: An internal flight identifier. Used internally for *Fleet* interoperability.
  - aircraft_type: Aircraft type ICAO, e.g. "A320".
  - wingspan: Aircraft wingspan, $[m]$.
  - n_engine: Number of aircraft engines.
  - engine_uid: Aircraft engine unique identifier. Used for emissions calculations with the ICAO Aircraft Emissions Databank (EDB).
  - max_mach_number: Maximum Mach number at cruise altitude. Used by some aircraft performance models to clip true airspeed.

  Numeric quantities that are constant over the entire flight trajectory should be included as attributes.
- **copy** (bool, *optional*) – Copy data on Flight creation. Defaults to True.
- **fuel** (*Fuel*, *optional*) – Fuel used in flight trajectory. Defaults to *JetA*.
- **drop_duplicated_times** (bool, *optional*) – Drop duplicate times in flight trajectory. Defaults to False.
- ****attrs_kwargs** (Any) – Additional flight properties passed as keyword arguments.

Raises

KeyError – Raises if data input does not contain at least time, latitude, longitude, (altitude or level).

Notes

The Traffic library has many helpful flight processing utilities.

See `traffic.core.Flight` for more information.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from pycontrails import Flight
```

```
>>> # Create `Flight` from a DataFrame.
>>> df = pd.DataFrame({
...     "longitude": np.linspace(20, 30, 500),
...     "latitude": np.linspace(40, 10, 500),
...     "altitude": 10500,
...     "time": pd.date_range('2021-01-01T10', '2021-01-01T15', periods=500),
... })
>>> fl = Flight(data=df, flight_id=123)  # specify a flight_id by keyword
>>> fl
Flight [4 keys x 500 length, 2 attributes]
Keys: longitude, latitude, altitude, time
Attributes:
time                [2021-01-01 10:00:00, 2021-01-01 15:00:00]
longitude           [20.0, 30.0]
latitude            [10.0, 40.0]
altitude            [10500.0, 10500.0]
flight_id           123
crs                 EPSG:4326
```

```
>>> # Create `Flight` from keywords
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl
Flight [4 keys x 200 length, 1 attributes]
    Keys: longitude, latitude, time, altitude
    Attributes:
    time                [2021-01-01 12:00:00, 2021-01-01 14:00:00]
    longitude           [20.0, 30.0]
    latitude            [30.0, 40.0]
    altitude            [11000.0, 11000.0]
    crs                 EPSG:4326
```

```
>>> # Access the underlying data as DataFrame
>>> fl.dataframe.head()
   longitude   latitude                          time  altitude
0  20.000000  40.000000 2021-01-01 12:00:00.000000000   11000.0
```

(continues on next page)

(continued from previous page)

```
1  20.050251  39.949749 2021-01-01 12:00:36.180904522   11000.0
2  20.100503  39.899497 2021-01-01 12:01:12.361809045   11000.0
3  20.150754  39.849246 2021-01-01 12:01:48.542713567   11000.0
4  20.201005  39.798995 2021-01-01 12:02:24.723618090   11000.0
```

__init__(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, *fuel=None*, *drop_duplicated_times=False*, ***attrs_kwargs*)

Methods

T_isa()	Calculate the ICAO standard atmosphere temperature at each point.
__init__([data, longitude, latitude, ...])	
broadcast_attrs(keys[, overwrite, raise_error])	Attach values from `keys` in `attrs` onto `data`.
broadcast_numeric_attrs([ignore_keys, overwrite])	Attach numeric values in `attrs` onto `data`.
clean_and_resample([freq, fill_method, ...])	Resample and (possibly) filter a flight trajectory.
coords_intersect_met(met)	Return boolean mask of data inside the bounding box defined by `met`.
copy(**kwargs)	Return a copy of this VectorDatasetType class.
create_empty([keys, attrs])	Create instance with variables defined by *keys* and size 0.
distance_to_coords(distance)	Convert distance along flight path to geodesic coordinates.
downselect_met(met, *[, longitude_buffer, ...])	Downselect `met` to encompass a spatiotemporal region of the data.
ensure_vars(vars[, raise_error])	Ensure variables exist in column of `data` or `attrs`.
filter(mask[, copy])	Filter `data` according to a boolean array `mask`.
filter_altitude([kernel_size, cruise_threshold])	Filter noisy altitude on a single flight.
filter_by_first()	Keep first row of group of waypoints with identical coordinates.
from_dict(obj[, copy])	Create instance from dict representation containing data and attrs.
generate_splits(n_splits[, copy])	Split instance into `n_split` sub-vectors.
get(key[, default_value])	Get values from `data` with `default_value` if `key` not in `data`.
get_data_or_attr(key[, default])	Get value from `data` or `attrs`.
intersect_met(mda, *[, longitude, latitude, ...])	Intersect waypoints with MetDataArray.
length_met(key[, threshold])	Calculate total horizontal distance where column `key` exceeds `threshold`.
plot(**kwargs)	Plot flight trajectory longitude-latitude values.
proportion_met(key[, threshold])	Calculate proportion of flight with certain meteorological constraint.
resample_and_fill([freq, fill_method, ...])	Resample and fill flight trajectory with geodesics and linear interpolation.
segment_angle()	Calculate sine and cosine for the angle between each segment and the longitudinal axis.
segment_azimuth()	Calculate (forward) azimuth at each waypoint.
segment_duration([dtype])	Compute time elapsed between waypoints in seconds.

continues on next page

Table 1 – continued from previous page

segment_groundspeed([smooth, window_length, ...])	Return groundspeed across segments.
segment_haversine()	Compute Haversine (great circle) distance between flight waypoints.
segment_length()	Compute spherical distance between flight waypoints.
segment_mach_number(true_airspeed, ...)	Calculate the mach number of each segment.
segment_phase([threshold_rocd, ...])	Identify the phase of flight (climb, cruise, descent) for each segment.
segment_rocd()	Calculate the rate of climb and descent (ROCD).
segment_true_airspeed([u_wind, v_wind, ...])	Calculate the true airspeed $[m/s]$ from the ground speed and horizontal winds.
select(keys[, copy])	Return new class instance only containing specified keys.
setdefault(key[, default])	Shortcut to `VectorDataDict.setdefault()`.
sort(by)	Sort data by key(s).
sum(vectors[, infer_attrs, fill_value])	Sum a list of *VectorDataset* instances.
to_dataframe([copy])	Create `pd.DataFrame` in which each key-value pair in `data` is a column.
to_dict()	Create dictionary with `data` and `attrs`.
to_geojson_linestring()	Return trajectory as geojson FeatureCollection containing single LineString.
to_geojson_multilinestring(key[, ...])	Return trajectory as GeoJSON FeatureCollection of MultiLineStrings.
to_geojson_points()	Return dataset as GeoJSON FeatureCollection of Points.
to_lon_lat_grid(agg, *[, spatial_bbox, ...])	Convert vectors to a longitude-latitude grid.
to_pseudo_mercator([copy])	Convert data from `attrs["crs"]` to Pseudo Mercator (EPSG:3857).
to_traffic()	Convert Flight instance to `traffic.core.Flight` instance.
transform_crs(crs[, copy])	Transform trajectory data from one coordinate reference system (CRS) to another.
update([other])	Update values in `data` dict without warning if overwriting.

Attributes

fuel	Fuel used in flight trajectory
air_pressure	Get air_pressure values for points.
altitude	Get altitude.
altitude_ft	Get altitude in feet.
attrs	Generic dataset attributes
constants	Return a dictionary of constant attributes and data values.
coords	Get geospatial coordinates for compatibility with MetDataArray.
data	Vector data with labels as keys and numpy.ndarray as values
dataframe	Shorthand property to access to_dataframe() with copy=False.
duration	Determine flight duration.
hash	Generate a unique hash for this class instance.
length	Return flight length based on WGS84 geodesic.
level	Get pressure level values for points.
max_distance_gap	Return maximum distance gap between waypoints along flight trajectory.
max_time_gap	Return maximum time gap between waypoints along flight trajectory.
required_keys	Required keys for creating GeoVectorDataset
shape	Shape of each array in data.
size	Length of each array in data.
time_end	Last waypoint time.
time_start	First waypoint time.
vertical_keys	At least one of these vertical-coordinate keys must also be included

clean_and_resample(*freq='1min'*, *fill_method='geodesic'*, *geodesic_threshold=100000.0*, *nominal_rocd=12.7*, *kernel_size=17*, *cruise_threshold=120.0*, *force_filter=False*, *drop=True*, *keep_original_index=False*, *climb_descend_at_end=False*)

Resample and (possibly) filter a flight trajectory.

Waypoints are resampled according to the frequency `freq`. If the original flight data has a short sampling period, *filter_altitude* will also be called to clean the data. Large gaps in trajectories may be interpolated as step climbs through *_altitude_interpolation*.

Parameters

- **freq** (str, *optional*) – Resampling frequency, by default "1min"
- **fill_method** ({"geodesic", "linear"}, *optional*) – Choose between "geodesic" and "linear", by default "geodesic". In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.
- **geodesic_threshold** (float, *optional*) – Threshold for geodesic interpolation, [m]. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (float, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to constants.nominal_rocd.

- **kernel_size** (int, *optional*) – Passed directly to scipy.signal.medfilt(), by default 11. Passed also to scipy.signal.medfilt()
- **cruise_theshold** (float, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter
- **force_filter** (bool, *optional*) – If set to true, meth:*filter_altitude* will always be called. otherwise, it will only be called if the flight has a median sample period under 10 seconds
- **drop** (bool, *optional*) – Drop any columns that are not resampled and filled. Defaults to True, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with nan or None values, depending on the data type.
- **keep_original_index** (bool, *optional*) – Keep the original index of the *Flight* in addition to the new resampled index. Defaults to False. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (bool) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns
Flight – Filled Flight

copy(***kwargs*)

Return a copy of this VectorDatasetType class.

Parameters
****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Copy of class

distance_to_coords(*distance*)

Convert distance along flight path to geodesic coordinates.

Will return a tuple containing *(lat, lon, index)*, where index indicates which flight segment contains the returned coordinate.

Parameters
distance (ArrayOrFloat) – Distance along flight path, [m]

Returns
(ArrayOrFloat, ArrayOrFloat, int | npt.NDArray[int]) – latitude, longitude, and segment index cooresponding to distance.

property duration

Determine flight duration.

Returns
pd.Timedelta – Difference between terminal and initial time

filter(*mask*, *copy=True*, ***kwargs*)

Filter data according to a boolean array mask.

Entries corresponding to mask == True are kept.

Parameters

- **mask** (npt.NDArray[np.bool_]) – Boolean array with compatible shape.
- **copy** (bool, *optional*) – Copy data on filter. Defaults to True. See numpy best practices for insight into whether copy is appropriate.

- ****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Containing filtered data

Raises
TypeError – If mask is not a boolean array.

filter_altitude(*kernel_size=17*, *cruise_threshold=120.0*)

Filter noisy altitude on a single flight.

Currently runs altitude through a median filter using scipy.signal.medfilt() with kernel_size, then a Savitzky-Golay filter to filter noise. The median filter is only applied during cruise segments that are longer than cruise_threshold.

Parameters

- **kernel_size** (int, *optional*) – Passed directly to scipy.signal.medfilt(), by default 11. Passed also to scipy.signal.medfilt()
- **cruise_theshold** (float, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter

Returns
Flight – Filtered Flight

Notes

Algorithm is derived from traffic.core.flight.Flight.filter().

The traffic algorithm also computes thresholds on sliding windows and replaces unacceptable values with NaNs.

Errors may raised if the kernel_size is too large.

See also:

traffic.core.flight.Flight.filter(), scipy.signal.medfilt()

filter_by_first()

Keep first row of group of waypoints with identical coordinates.

Chaining this method with *resample_and_fill* often gives a cleaner trajectory when using noisy flight waypoints.

Returns
Flight – Filtered Flight instance

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
```

```
>>> df = pd.DataFrame()
>>> df['longitude'] = [0, 0, 50]
>>> df['latitude'] = 0
>>> df['altitude'] = 0
>>> df['time'] = [datetime(2020, 1, 1, h) for h in range(3)]
```

```
>>> fl = Flight(df)
```

```
>>> fl.filter_by_first().dataframe
   longitude  latitude  altitude                 time
0        0.0       0.0       0.0 2020-01-01 00:00:00
1       50.0       0.0       0.0 2020-01-01 02:00:00
```

fuel

Fuel used in flight trajectory

property length

Return flight length based on WGS84 geodesic.

Returns

`float` – Total flight length, [m]

Raises

`NotImplementedError` – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.length
1436924.67...
```

length_met(*key*, *threshold=1.0*)

Calculate total horizontal distance where column `key` exceeds `threshold`.

Parameters

- **key** (`str`) – Column key in `data`
- **threshold** (`float`) – Consider trajectory waypoints whose associated `key` value exceeds `threshold`, by default 1.0

Returns

`float` – Total distance, [m]

Raises

- `KeyError` – `data` does not contain column `key`
- `NotImplementedError` – Raised when `attrs["crs"]` is not EPSG:4326

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
>>> import numpy as np
>>> from pycontrails.datalib.ecmwf import ERA5
>>> from pycontrails import Flight
```

```
>>> # Get met data
>>> times = (datetime(2022, 3, 1, 0),  datetime(2022, 3, 1, 3))
>>> variables = ["air_temperature", "specific_humidity"]
>>> levels = [300, 250, 200]
>>> era5 = ERA5(time=times, variables=variables, pressure_levels=levels)
>>> met = era5.open_metdataset()
```

```
>>> # Build flight
>>> df = pd.DataFrame()
>>> df['time'] = pd.date_range('2022-03-01T00', '2022-03-01T03', periods=11)
>>> df['longitude'] = np.linspace(-20, 20, 11)
>>> df['latitude'] = np.linspace(-20, 20, 11)
>>> df['altitude'] = np.linspace(9500, 10000, 11)
>>> fl = Flight(df).resample_and_fill('10s')
```

```
>>> # Intersect and attach
>>> fl["air_temperature"] = fl.intersect_met(met['air_temperature'])
>>> fl["air_temperature"]
array([235.94657007, 235.95766965, 235.96873412, ..., 234.59917962,
       234.60387402, 234.60845312])
```

```
>>> # Length (in meters) of waypoints whose temperature exceeds 236K
>>> fl.length_met("air_temperature", threshold=236)
4132178.159...
```

```
>>> # Proportion (with respect to distance) of waypoints whose temperature␣
↪exceeds 236K
>>> fl.proportion_met("air_temperature", threshold=236)
0.663552...
```

property max_distance_gap

Return maximum distance gap between waypoints along flight trajectory.

Distance is calculated based on WGS84 geodesic.

Returns

`float` – Maximum distance between waypoints, $[m]$

Raises

`NotImplementedError` – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.max_distance_gap
7391.27...
```

property max_time_gap

Return maximum time gap between waypoints along flight trajectory.

Returns
pd.Timedelta – Gap size

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.max_time_gap
Timedelta('0 days 00:00:36.180...')
```

plot(***kwargs*)

Plot flight trajectory longitude-latitude values.

Parameters
****kwargs** (Any) – Additional plot properties to passed to *pd.DataFrame.plot*

Returns
matplotlib.axes.Axes – Plot

proportion_met(*key*, *threshold=1.0*)

Calculate proportion of flight with certain meteorological constraint.

Parameters

- **key** (str) – Column key in data
- **threshold** (float) – Consider trajectory waypoints whose associated key value exceeds threshold, Defaults to 1.0

Returns
float – Ratio

resample_and_fill(*freq='1min'*, *fill_method='geodesic'*, *geodesic_threshold=100000.0*, *nominal_rocd=12.7*, *drop=True*, *keep_original_index=False*, *climb_descend_at_end=False*)

Resample and fill flight trajectory with geodesics and linear interpolation.

Waypoints are resampled according to the frequency `freq`. Values for `data` columns `longitude`, `latitude`, and `altitude` are interpolated.

Resampled waypoints will include all multiples of `freq` between the flight start and end time. For example, when resampling to a frequency of 1 minute, a flight that starts at 2020/1/1 00:00:59 and ends at 2020/1/1 00:01:01 will return a single waypoint at 2020/1/1 00:01:00, whereas a flight that starts at 2020/1/1 00:01:01 and ends at 2020/1/1 00:01:59 will return an empty flight.

Parameters

- **freq** (`str`, *optional*) – Resampling frequency, by default "1min"
- **fill_method** (`{"geodesic", "linear"}`, *optional*) – Choose between `"geodesic"` and `"linear"`, by default `"geodesic"`. In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.
- **geodesic_threshold** (`float`, *optional*) – Threshold for geodesic interpolation, $[m]$. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (`float | None`, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to `constants.nominal_rocd`.
- **drop** (`bool`, *optional*) – Drop any columns that are not resampled and filled. Defaults to `True`, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with `nan` or `None` values, depending on the data type.
- **keep_original_index** (`bool`, *optional*) – Keep the original index of the *`Flight`* in addition to the new resampled index. Defaults to `False`. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (`bool`) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns

`Flight` – Filled Flight

Raises

`ValueError` – Unknown `fill_method`

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
```

```
>>> df = pd.DataFrame()
>>> df['longitude'] = [0, 0, 50]
>>> df['latitude'] = 0
>>> df['altitude'] = 0
>>> df['time'] = [datetime(2020, 1, 1, h) for h in range(3)]
```

```
>>> fl = Flight(df)
>>> fl.dataframe
   longitude  latitude  altitude                time
           0       0.0       0.0       0.0 2020-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
           1        0.0       0.0       0.0 2020-01-01 01:00:00
           2       50.0       0.0       0.0 2020-01-01 02:00:00
```

```
>>> fl.resample_and_fill('10min').dataframe  # resample with 10 minute frequency
    longitude  latitude  altitude                time
0    0.000000       0.0       0.0 2020-01-01 00:00:00
1    0.000000       0.0       0.0 2020-01-01 00:10:00
2    0.000000       0.0       0.0 2020-01-01 00:20:00
3    0.000000       0.0       0.0 2020-01-01 00:30:00
4    0.000000       0.0       0.0 2020-01-01 00:40:00
5    0.000000       0.0       0.0 2020-01-01 00:50:00
6    0.000000       0.0       0.0 2020-01-01 01:00:00
7    8.333333       0.0       0.0 2020-01-01 01:10:00
8   16.666667       0.0       0.0 2020-01-01 01:20:00
9   25.000000       0.0       0.0 2020-01-01 01:30:00
10  33.333333       0.0       0.0 2020-01-01 01:40:00
11  41.666667       0.0       0.0 2020-01-01 01:50:00
12  50.000000       0.0       0.0 2020-01-01 02:00:00
```

segment_angle()

Calculate sine and cosine for the angle between each segment and the longitudinal axis.

This is different from the usual navigational angle between two points known as *bearing*.

Bearing in 3D spherical coordinates is referred to as *azimuth*.

```
        (lon_2, lat_2)  X
                       /|
                      / |
                     /  |
                    /   |
                   /    |
                  /     |
                 /      |
(lon_1, lat_1)  X -------> longitude (x-axis)
```

Returns

npt.NDArray[np.float64], npt.NDArray[np.float64] – Returns sin(a), cos(a), where a is the angle between the segment and the longitudinal axis. The final values are of both arrays are np.nan.

See also:

geo.segment_angle(), units.heading_to_longitudinal_angle(), *segment_azimuth()*, geo.forward_azimuth()

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> sin, cos = fl.segment_angle()
>>> sin
array([0.70716063, 0.70737598, 0.44819424, 0.31820671,        nan])
```

```
>>> cos
array([0.70705293, 0.70683748, 0.8939362 , 0.94802136,        nan])
```

segment_azimuth()

Calculate (forward) azimuth at each waypoint.

Method calls *pyproj.Geod.inv*, which is slow. See *geo.forward_azimuth* for an outline of a faster implementation.

Changed in version 0.33.7: The dtype of the output now matches the dtype of `self["longitude"]`.

Returns

`npt.NDArray[np.float64]` – Array of azimuths.

See also:

segment_angle(), `geo.forward_azimuth()`

segment_duration(*dtype=<class 'numpy.float32'>*)

Compute time elapsed between waypoints in seconds.

`np.nan` appended so the length of the output is the same as number of waypoints.

Parameters

dtype (`np.dtype`) – Numpy dtype for time difference. Defaults to `np.float64`

Returns

`npt.NDArray[np.float64]` – Time difference between waypoints, $[s]$. Returns an array with dtype specified by``dtype``

segment_groundspeed(*smooth=False*, *window_length=7*, *polyorder=1*)

Return groundspeed across segments.

Calculate by dividing the horizontal segment length by the difference in waypoint times.

Parameters

- **smooth** (`bool`, *optional*) – Smooth airspeed with Savitzky-Golay filter. Defaults to False.
- **window_length** (`int`, *optional*) – Passed directly to `scipy.signal.savgol_filter()`, by default 7.
- **polyorder** (`int`, *optional*) – Passed directly to `scipy.signal.savgol_filter()`, by default 1.

Returns

`npt.NDArray[np.float64]` – Groundspeed of the segment, $[ms^{-1}]$

segment_haversine()

Compute Haversine (great circle) distance between flight waypoints.

Helper function used in *resample_and_fill()*. *np.nan* appended so the length of the output is the same as number of waypoints.

To account for vertical displacements when computing segment lengths, use *segment_length()*.

Returns
npt.NDArray[np.float64] – Array of great circle distances in $[m]$ between waypoints

Raises
NotImplementedError – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> fl.segment_haversine()
array([157255.03346286, 157231.08336815, 248456.48781503, 351047.44358851,
                   nan])
```

See also:

segment_haversine(), *segment_length()*

segment_length()

Compute spherical distance between flight waypoints.

Helper function used in *length()* and *length_met()*. *np.nan* appended so the length of the output is the same as number of waypoints.

Returns
npt.NDArray[np.float64] – Array of distances in $[m]$ between waypoints

Raises
NotImplementedError – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> fl.segment_length()
array([157255.03346286, 157231.08336815, 248456.48781503, 351047.44358851,
                   nan])
```

See also:

segment_length()

segment_mach_number(*true_airspeed*, *air_temperature*)

Calculate the mach number of each segment.

Parameters

- **true_airspeed** (npt.NDArray[np.float64]) – True airspeed of the segment, [$m\ s^{-1}$]. See *segment_true_airspeed()*.
- **air_temperature** (npt.NDArray[np.float64]) – Average air temperature of each segment, [K]

Returns

npt.NDArray[np.float64] – Mach number of each segment

segment_phase(*threshold_rocd=250.0*, *min_cruise_altitude_ft=20000.0*)

Identify the phase of flight (climb, cruise, descent) for each segment.

Parameters

- **threshold_rocd** (float, *optional*) – ROCD threshold to identify climb and descent, [$ftmin^{-1}$]. Currently set to 250 ft/min.
- **min_cruise_altitude_ft** (float, *optional*) – Minimum altitude for cruise, [ft] This is specific for each aircraft type, and can be approximated as 50% of the altitude ceiling. Defaults to 20000 ft.

Returns

npt.NDArray[np.uint8] – Array of values enumerating the flight phase. See flight.FlightPhase for enumeration.

See also:

FlightPhase, *segment_phase()*, *segment_rocd()*

segment_rocd()

Calculate the rate of climb and descent (ROCD).

Returns

npt.NDArray[np.float64] – Rate of climb and descent over segment, [$ftmin^{-1}$]

See also:

segment_rocd()

segment_true_airspeed(*u_wind=0.0*, *v_wind=0.0*, *smooth=True*, *window_length=7*, *polyorder=1*)

Calculate the true airspeed [m/s] from the ground speed and horizontal winds.

The calculated ground speed will first be smoothed with a Savitzky-Golay filter if enabled.

Parameters

- **u_wind** (npt.NDArray[np.float64] | float) – U wind speed, [$m\ s^{-1}$]. Defaults to 0 for all waypoints.
- **v_wind** (npt.NDArray[np.float64] | float) – V wind speed, [$m\ s^{-1}$]. Defaults to 0 for all waypoints.
- **smooth** (bool, *optional*) – Smooth airspeed with Savitzky-Golay filter. Defaults to True.
- **window_length** (int, *optional*) – Passed directly to scipy.signal.savgol_filter(), by default 7.

- **polyorder** (int, *optional*) – Passed directly to scipy.signal.savgol_filter(), by default 1.

Returns
npt.NDArray[np.float64] – True wind speed of each segment, [$m\ s^{-1}$]

sort(*by*)

Sort data by key(s).

This method always creates a copy of the data by calling pandas.DataFrame.sort_values().

Parameters
by (str | list[str]) – Key or list of keys to sort by.

Returns
VectorDatasetType – Instance with sorted data.

property time_end

Last waypoint time.

Returns
pandas.Timestamp – Last waypoint time

property time_start

First waypoint time.

Returns
pandas.Timestamp – First waypoint time

to_geojson_linestring()

Return trajectory as geojson FeatureCollection containing single LineString.

Returns
dict[str, Any] – Python representation of geojson FeatureCollection

to_geojson_multilinestring(*key*, *split_antimeridian=True*)

Return trajectory as GeoJSON FeatureCollection of MultiLineStrings.

Flight data is grouped according to values of key. Each group gives rise to a Feature containing a MultiLineString geometry. LineStrings can be split over the antimeridian.

Parameters

- **key** (str) – Name of data column to group by
- **split_antimeridian** (bool, *optional*) – Split linestrings that cross the antimeridian. Defaults to True

Returns
dict[str, Any] – Python representation of GeoJSON FeatureCollection of MultiLinestring Features

Raises
KeyError – data does not contain column key

to_traffic()

Convert Flight instance to traffic.core.Flight instance.

See https://traffic-viz.github.io/traffic.core.flight.html#traffic.core.Flight

Returns
traffic.core.Flight – *traffic.core.Flight* instance

Raises
ModuleNotFoundError – *traffic* package not installed

pycontrails.Fleet

class pycontrails.**Fleet**(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, *fuel=None*, *fl_attrs=None*, ***attrs_kwargs*)

Bases: *Flight*

Data structure for holding a sequence of *Flight* instances.

Flight waypoints are merged into a single *Flight*-like object.

__init__(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, *fuel=None*, *fl_attrs=None*, ***attrs_kwargs*)

Methods

T_isa()	Calculate the ICAO standard atmosphere temperature at each point.
__init__([data, longitude, latitude, ...])	
broadcast_attrs(keys[, overwrite, raise_error])	Attach values from `keys` in `attrs` onto `data`.
broadcast_numeric_attrs([ignore_keys, overwrite])	Attach numeric values in `attrs` onto `data`.
clean_and_resample([freq, fill_method, ...])	Resample and (possibly) filter a flight trajectory.
coords_intersect_met(met)	Return boolean mask of data inside the bounding box defined by `met`.
copy(**kwargs)	Return a copy of this VectorDatasetType class.
create_empty([keys, attrs])	Create instance with variables defined by *keys* and size 0.
distance_to_coords(distance)	Convert distance along flight path to geodesic coordinates.
downselect_met(met, *[, longitude_buffer, ...])	Downselect `met` to encompass a spatiotemporal region of the data.
ensure_vars(vars[, raise_error])	Ensure variables exist in column of `data` or `attrs`.
filter(mask[, copy])	Filter `data` according to a boolean array `mask`.
filter_altitude([kernel_size, cruise_threshold])	Filter noisy altitude on a single flight.
filter_by_first()	Keep first row of group of waypoints with identical coordinates.
from_dict(obj[, copy])	Create instance from dict representation containing data and attrs.
from_seq(seq[, broadcast_numeric, copy, attrs])	Instantiate a *Fleet* instance from an iterable of *Flight*.
generate_splits(n_splits[, copy])	Split instance into `n_split` sub-vectors.
get(key[, default_value])	Get values from `data` with `default_value` if `key` not in `data`.
get_data_or_attr(key[, default])	Get value from `data` or `attrs`.
intersect_met(mda, *[, longitude, latitude, ...])	Intersect waypoints with MetDataArray.
length_met(key[, threshold])	Calculate total horizontal distance where column `key` exceeds `threshold`.

continues on next page

Table 2 – continued from previous page

plot(**kwargs)	Plot flight trajectory longitude-latitude values.
proportion_met(key[, threshold])	Calculate proportion of flight with certain meteorological constraint.
resample_and_fill(*args, **kwargs)	Resample and fill flight trajectory with geodesics and linear interpolation.
segment_angle()	Calculate sine and cosine for the angle between each segment and the longitudinal axis.
segment_azimuth()	Calculate (forward) azimuth at each waypoint.
segment_duration([dtype])	Compute time elapsed between waypoints in seconds.
segment_groundspeed(*args, **kwargs)	Return groundspeed across segments.
segment_haversine()	Compute Haversine (great circle) distance between flight waypoints.
segment_length()	Compute spherical distance between flight waypoints.
segment_mach_number(true_airspeed, ...)	Calculate the mach number of each segment.
segment_phase([threshold_rocd, ...])	Identify the phase of flight (climb, cruise, descent) for each segment.
segment_rocd()	Calculate the rate of climb and descent (ROCD).
segment_true_airspeed([u_wind, v_wind, ...])	Calculate the true airspeed $[m/s]$ from the ground speed and horizontal winds.
select(keys[, copy])	Return new class instance only containing specified keys.
setdefault(key[, default])	Shortcut to `VectorDataDict.setdefault()`.
sort(by)	Sort data by key(s).
sum(vectors[, infer_attrs, fill_value])	Sum a list of *VectorDataset* instances.
to_dataframe([copy])	Create `pd.DataFrame` in which each key-value pair in `data` is a column.
to_dict()	Create dictionary with `data` and `attrs`.
to_flight_list([copy])	De-concatenate merged waypoints into a list of Flight instances.
to_geojson_linestring()	Return trajectory as geojson FeatureCollection containing single LineString.
to_geojson_multilinestring(key[, ...])	Return trajectory as GeoJSON FeatureCollection of MultiLineStrings.
to_geojson_points()	Return dataset as GeoJSON FeatureCollection of Points.
to_lon_lat_grid(agg, *[, spatial_bbox, ...])	Convert vectors to a longitude-latitude grid.
to_pseudo_mercator([copy])	Convert data from `attrs["crs"]` to Pseudo Mercator (EPSG:3857).
to_traffic()	Convert Flight instance to `traffic.core.Flight` instance.
transform_crs(crs[, copy])	Transform trajectory data from one coordinate reference system (CRS) to another.
update([other])	Update values in `data` dict without warning if overwriting.

Attributes

fl_attrs	
final_waypoints	
air_pressure	Get air_pressure values for points.
altitude	Get altitude.
altitude_ft	Get altitude in feet.
attrs	Generic dataset attributes
constants	Return a dictionary of constant attributes and data values.
coords	Get geospatial coordinates for compatibility with MetDataArray.
data	Vector data with labels as keys and numpy.ndarray as values
dataframe	Shorthand property to access to_dataframe() with copy=False.
duration	Determine flight duration.
fuel	Fuel used in flight trajectory
hash	Generate a unique hash for this class instance.
length	Return flight length based on WGS84 geodesic.
level	Get pressure level values for points.
max_distance_gap	Return maximum distance gap between waypoints along flight trajectory.
max_time_gap	Return maximum time gap between waypoints along flight trajectory.
n_flights	Return number of distinct flights.
required_keys	Required keys for creating GeoVectorDataset
shape	Shape of each array in data.
size	Length of each array in data.
time_end	Last waypoint time.
time_start	First waypoint time.
vertical_keys	At least one of these vertical-coordinate keys must also be included

clean_and_resample(*freq='1min'*, *fill_method='geodesic'*, *geodesic_threshold=100000.0*, *nominal_rocd=0.0*, *kernel_size=17*, *cruise_threshold=120*, *force_filter=False*, *drop=True*, *keep_original_index=False*, *climb_descend_at_end=False*)

Resample and (possibly) filter a flight trajectory.

Waypoints are resampled according to the frequency `freq`. If the original flight data has a short sampling period, *filter_altitude* will also be called to clean the data. Large gaps in trajectories may be interpolated as step climbs through *_altitude_interpolation*.

Parameters

- **freq** (str, *optional*) – Resampling frequency, by default "1min"
- **fill_method** ({"geodesic", "linear"}, *optional*) – Choose between "geodesic" and "linear", by default "geodesic". In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.

- **geodesic_threshold** (float, *optional*) – Threshold for geodesic interpolation, [m]. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (float, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to constants.nominal_rocd.
- **kernel_size** (int, *optional*) – Passed directly to scipy.signal.medfilt(), by default 11. Passed also to scipy.signal.medfilt()
- **cruise_theshold** (float, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter
- **force_filter** (bool, *optional*) – If set to true, meth:*filter_altitude* will always be called. otherwise, it will only be called if the flight has a median sample period under 10 seconds
- **drop** (bool, *optional*) – Drop any columns that are not resampled and filled. Defaults to True, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with nan or None values, depending on the data type.
- **keep_original_index** (bool, *optional*) – Keep the original index of the *Flight* in addition to the new resampled index. Defaults to False. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (bool) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns
Flight – Filled Flight

copy(***kwargs*)

Return a copy of this VectorDatasetType class.

Parameters
****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Copy of class

filter(*mask*, *copy=True*, ***kwargs*)

Filter data according to a boolean array mask.

Entries corresponding to mask == True are kept.

Parameters

- **mask** (npt.NDArray[np.bool_]) – Boolean array with compatible shape.
- **copy** (bool, *optional*) – Copy data on filter. Defaults to True. See numpy best practices for insight into whether copy is appropriate.
- ****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Containing filtered data

Raises
TypeError – If mask is not a boolean array.

final_waypoints

fl_attrs

classmethod from_seq(*seq*, *broadcast_numeric=True*, *copy=True*, *attrs=None*)

Instantiate a *Fleet* instance from an iterable of *Flight*.

Changed in version 0.49.3: Empty flights are now filtered out before concatenation.

Parameters

- **seq** (Iterable[Flight]) – An iterable of *Flight* instances.
- **broadcast_numeric** (bool, *optional*) – If True, broadcast numeric attributes to data variables.
- **copy** (bool, *optional*) – If True, make copy of each flight instance in seq.
- **attrs** (dict[str, Any] | None, *optional*) – Global attribute to attach to instance.

Returns

Fleet – A *Fleet* instance made from concatenating the *Flight* instances in seq. The fuel type is taken from the first *Flight* in seq.

property max_distance_gap

Return maximum distance gap between waypoints along flight trajectory.

Distance is calculated based on WGS84 geodesic.

Returns

float – Maximum distance between waypoints, $[m]$

Raises

NotImplementedError – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.max_distance_gap
7391.27...
```

property n_flights

Return number of distinct flights.

Returns

int – Number of flights

resample_and_fill(**args*, ***kwargs*)

Resample and fill flight trajectory with geodesics and linear interpolation.

Waypoints are resampled according to the frequency freq. Values for data columns longitude, latitude, and altitude are interpolated.

Resampled waypoints will include all multiples of freq between the flight start and end time. For example, when resampling to a frequency of 1 minute, a flight that starts at 2020/1/1 00:00:59 and ends at 2020/1/1

00:01:01 will return a single waypoint at 2020/1/1 00:01:00, whereas a flight that starts at 2020/1/1 00:01:01 and ends at 2020/1/1 00:01:59 will return an empty flight.

Parameters

- **freq** (`str`, *optional*) – Resampling frequency, by default "1min"
- **fill_method** (`{"geodesic", "linear"}`, *optional*) – Choose between `"geodesic"` and `"linear"`, by default `"geodesic"`. In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.
- **geodesic_threshold** (`float`, *optional*) – Threshold for geodesic interpolation, $[m]$. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (`float | None`, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to `constants.nominal_rocd`.
- **drop** (`bool`, *optional*) – Drop any columns that are not resampled and filled. Defaults to `True`, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with `nan` or `None` values, depending on the data type.
- **keep_original_index** (`bool`, *optional*) – Keep the original index of the *`Flight`* in addition to the new resampled index. Defaults to `False`. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (`bool`) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns

`Flight` – Filled Flight

Raises

`ValueError` – Unknown `fill_method`

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
```

```
>>> df = pd.DataFrame()
>>> df['longitude'] = [0, 0, 50]
>>> df['latitude'] = 0
>>> df['altitude'] = 0
>>> df['time'] = [datetime(2020, 1, 1, h) for h in range(3)]
```

```
>>> fl = Flight(df)
>>> fl.dataframe
   longitude  latitude  altitude                time
           0       0.0       0.0       0.0 2020-01-01 00:00:00
           1       0.0       0.0       0.0 2020-01-01 01:00:00
           2      50.0       0.0       0.0 2020-01-01 02:00:00
```

```
>>> fl.resample_and_fill('10min').dataframe  # resample with 10 minute frequency
    longitude  latitude  altitude                time
```

(continues on next page)

(continued from previous page)

```
0    0.000000       0.0       0.0 2020-01-01 00:00:00
1    0.000000       0.0       0.0 2020-01-01 00:10:00
2    0.000000       0.0       0.0 2020-01-01 00:20:00
3    0.000000       0.0       0.0 2020-01-01 00:30:00
4    0.000000       0.0       0.0 2020-01-01 00:40:00
5    0.000000       0.0       0.0 2020-01-01 00:50:00
6    0.000000       0.0       0.0 2020-01-01 01:00:00
7    8.333333       0.0       0.0 2020-01-01 01:10:00
8   16.666667       0.0       0.0 2020-01-01 01:20:00
9   25.000000       0.0       0.0 2020-01-01 01:30:00
10  33.333333       0.0       0.0 2020-01-01 01:40:00
11  41.666667       0.0       0.0 2020-01-01 01:50:00
12  50.000000       0.0       0.0 2020-01-01 02:00:00
```

segment_angle()

Calculate sine and cosine for the angle between each segment and the longitudinal axis.

This is different from the usual navigational angle between two points known as *bearing*.

Bearing in 3D spherical coordinates is referred to as *azimuth*.

```
        (lon_2, lat_2)  X
                       /|
                      / |
                     /  |
                    /   |
                   /    |
                  /     |
                 /      |
(lon_1, lat_1)  X -------> longitude (x-axis)
```

Returns

npt.NDArray[np.float64], npt.NDArray[np.float64] – Returns sin(a), cos(a), where a is the angle between the segment and the longitudinal axis. The final values are of both arrays are np.nan.

See also:

geo.segment_angle(), units.heading_to_longitudinal_angle(), *segment_azimuth()*, geo.forward_azimuth()

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> sin, cos = fl.segment_angle()
```

(continues on next page)

(continued from previous page)

```
>>> sin
array([0.70716063, 0.70737598, 0.44819424, 0.31820671,        nan])
```

```
>>> cos
array([0.70705293, 0.70683748, 0.8939362 , 0.94802136,        nan])
```

segment_azimuth()

Calculate (forward) azimuth at each waypoint.

Method calls *pyproj.Geod.inv*, which is slow. See *geo.forward_azimuth* for an outline of a faster implementation.

Changed in version 0.33.7: The dtype of the output now matches the dtype of `self["longitude"]`.

Returns
`npt.NDArray[np.float64]` – Array of azimuths.

See also:

segment_angle(), `geo.forward_azimuth()`

segment_groundspeed(**args*, ***kwargs*)

Return groundspeed across segments.

Calculate by dividing the horizontal segment length by the difference in waypoint times.

Parameters

- **smooth** (`bool`, *optional*) – Smooth airspeed with Savitzky-Golay filter. Defaults to False.
- **window_length** (`int`, *optional*) – Passed directly to `scipy.signal.savgol_filter()`, by default 7.
- **polyorder** (`int`, *optional*) – Passed directly to `scipy.signal.savgol_filter()`, by default 1.

Returns
`npt.NDArray[np.float64]` – Groundspeed of the segment, $[ms^{-1}]$

segment_length()

Compute spherical distance between flight waypoints.

Helper function used in `length()` and `length_met()`. *np.nan* appended so the length of the output is the same as number of waypoints.

Returns
`npt.NDArray[np.float64]` – Array of distances in $[m]$ between waypoints

Raises
`NotImplementedError` – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> fl.segment_length()
array([157255.03346286, 157231.08336815, 248456.48781503, 351047.44358851,
                   nan])
```

See also:

segment_length()

segment_true_airspeed(*u_wind=0.0*, *v_wind=0.0*, *smooth=True*, *window_length=7*, *polyorder=1*)

Calculate the true airspeed $[m/s]$ from the ground speed and horizontal winds.

Because Flight.segment_true_airspeed uses a smoothing pattern, waypoints in `data` are not independent. Moreover, we expect the final waypoint of each flight to have a nan value associated to any segment property. Consequently, we need to define a custom method here to deal with these issues when applying this method on a fleet of flights.

See docstring for *Flight.segment_true_airspeed()*.

Raises

RuntimeError – Unexpected key *__u_wind* or *__v_wind* found in `data`.

sort(*by*)

Sort data by key(s).

This method always creates a copy of the data by calling `pandas.DataFrame.sort_values()`.

Parameters

by (`str | list[str]`) – Key or list of keys to sort by.

Returns

`VectorDataset|Type` – Instance with sorted data.

to_flight_list(*copy=True*)

De-concatenate merged waypoints into a list of Flight instances.

Any global `attrs` are lost.

Parameters

copy (`bool`, *optional*) – If True, make copy of each flight instance in *seq*.

Returns

`list[Flight]` – List of Flights in the same order as was passed into the *Fleet* instance.

pycontrails.FlightPhase

class pycontrails.**FlightPhase**(*value*, *names=None*, *, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: IntEnum

Flight phase enumeration.

Use `segment_phase()` or *Flight.segment_phase()* to determine flight phase.

__init__(**args*, ***kwds*)

Methods

__init__(*args, **kwds)	
as_integer_ratio()	Return integer ratio.
bit_count()	Number of ones in the binary representation of the absolute value of self.
bit_length()	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes([byteorder, signed])	Return the integer represented by the given array of bytes.
to_bytes([length, byteorder, signed])	Return an array of bytes representing an integer.

Attributes

CLIMB	Waypoints at which the flight is in a climb phase
CRUISE	Waypoints at which the flight is in a cruise phase
DESCENT	Waypoints at which the flight is in a descent phase
LEVEL_FLIGHT	Waypoints at which the flight is not in a climb, cruise, or descent phase.
NAN	Waypoints at which the ROCD is not defined.
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

CLIMB = 1

Waypoints at which the flight is in a climb phase

CRUISE = 2

Waypoints at which the flight is in a cruise phase

DESCENT = 3

Waypoints at which the flight is in a descent phase

LEVEL_FLIGHT = 4

Waypoints at which the flight is not in a climb, cruise, or descent phase. In practice, this category is used for waypoints at which the ROCD resembles that of a cruise phase, but the altitude is below the minimum cruise altitude.

NAN = 5

Waypoints at which the ROCD is not defined.

pycontrails.Fuel

class pycontrails.**Fuel**(*fuel_name*, *q_fuel*, *hydrogen_content*, *ei_co2*, *ei_h2o*, *ei_so2*, *ei_sulphates*, *ei_oc*)

Bases: `object`

Base class for the physical parameters of the fuel.

__init__(*fuel_name*, *q_fuel*, *hydrogen_content*, *ei_co2*, *ei_h2o*, *ei_so2*, *ei_sulphates*, *ei_oc*)

Methods

__init__(fuel_name, q_fuel, ...)	

Attributes

fuel_name	Fuel Name
q_fuel	Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$
hydrogen_content	Percentage of hydrogen mass content in the fuel
ei_co2	CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$
ei_h2o	Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$
ei_so2	Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$
ei_sulphates	Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$
ei_oc	Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

ei_co2

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

ei_so2

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$

ei_sulphates

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$

fuel_name

Fuel Name

hydrogen_content

Percentage of hydrogen mass content in the fuel

q_fuel

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

pycontrails.JetA

class pycontrails.**JetA**(*fuel_name='Jet A-1'*, *q_fuel=43130000.0*, *hydrogen_content=13.8*, *ei_co2=3.159*, *ei_h2o=1.23*, *ei_so2=0.0012*, *ei_sulphates=2.4489795918367345e-05*, *ei_oc=1.9999999999999998e-05*)

Bases: *Fuel*

Jet A-1 Fuel.

References

- [Celikel and Jelinek, 2001]
- [Lee *et al.*, 2021]
- [Stettler *et al.*, 2011]
- [Wilkerson *et al.*, 2010]

__init__(*fuel_name='Jet A-1'*, *q_fuel=43130000.0*, *hydrogen_content=13.8*, *ei_co2=3.159*, *ei_h2o=1.23*, *ei_so2=0.0012*, *ei_sulphates=2.4489795918367345e-05*, *ei_oc=1.9999999999999998e-05*)

Methods

__init__([fuel_name, q_fuel, ...])

Attributes

ei_co2	CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$
ei_h2o	Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$
ei_oc	Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$ - High uncertainty - Wilkerson et al. (2010): EI_OC = 15 mg/kg-fuel - Stettler et al. (2011): EI_OC = 20 [1, 40] mg/kg-fuel.
ei_so2	Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$ - The EI SO2 is proportional to the fuel sulphur content - Celikel (2001): EI_SO2 = 0.84 g/kg-fuel for 450 ppm fuel - Lee et al. (2021): EI_SO2 = 1.2 g/kg-fuel for 600 ppm fuel.
ei_sulphates	Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$ - The SOx-S is partitioned into 98% SO2-S gas and 2% S(VI)-S particle - References: Wilkerson et al. (2010) & Stettler et al. (2011).
fuel_name	Fuel Name
hydrogen_content	Percentage of hydrogen mass content in the fuel
q_fuel	Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

ei_co2 = 3.159

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o = 1.23

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc = 1.9999999999999998e-05

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$ - High uncertainty - Wilkerson et al. (2010): EI_OC = 15 mg/kg-fuel - Stettler et al. (2011): EI_OC = 20 [1, 40] mg/kg-fuel

ei_so2 = 0.0012

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$ - The EI SO2 is proportional to the fuel sulphur content - Celikel (2001): EI_SO2 = 0.84 g/kg-fuel for 450 ppm fuel - Lee et al. (2021): EI_SO2 = 1.2 g/kg-fuel for 600 ppm fuel

ei_sulphates = 2.4489795918367345e-05

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$ - The SOx-S is partitioned into 98% SO2-S gas and 2% S(VI)-S particle - References: Wilkerson et al. (2010) & Stettler et al. (2011)

fuel_name = 'Jet A-1'

Fuel Name

hydrogen_content = 13.8

Percentage of hydrogen mass content in the fuel

q_fuel = 43130000.0

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

pycontrails.SAFBlend

class pycontrails.**SAFBlend**(*pct_blend*)

Bases: *Fuel*

Jet A-1 / Sustainable Aviation Fuel Blend.

SAF only changes the CO2 lifecycle emissions, not the CO2 emissions emitted at the aircraft exhaust. We assume that the EI OC stays the same as Jet A-1 fuel due to lack of data.

Parameters

pct_blend (`float`) – Sustainable aviation fuel percentage blend ratio by volume, %. Expected to be in the interval `[0, 100]`.

References

- [Teoh *et al.*, 2022]
- [Schripp *et al.*, 2022]

__init__(*pct_blend*)

Methods

__init__(pct_blend)	

Attributes

fuel_name	Fuel Name
q_fuel	Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$
hydrogen_content	Percentage of hydrogen mass content in the fuel
ei_co2	CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$
ei_h2o	Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$
ei_so2	Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$
ei_sulphates	Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$
ei_oc	Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

pycontrails.HydrogenFuel

class pycontrails.**HydrogenFuel**(*fuel_name='Hydrogen'*, *q_fuel=122800000.0*, *hydrogen_content=nan*, *ei_co2=0.0*, *ei_h2o=9.21*, *ei_so2=0.0*, *ei_sulphates=0.0*, *ei_oc=0.0*)

Bases: *Fuel*

Hydrogen Fuel.

References

- [Khan *et al.*, 2022]

__init__(*fuel_name='Hydrogen'*, *q_fuel=122800000.0*, *hydrogen_content=nan*, *ei_co2=0.0*, *ei_h2o=9.21*, *ei_so2=0.0*, *ei_sulphates=0.0*, *ei_oc=0.0*)

Methods

__init__([fuel_name, q_fuel, ...])	

Attributes

ei_co2	CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$
ei_h2o	Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$
ei_oc	Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$
ei_so2	Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$
ei_sulphates	Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$
fuel_name	Fuel Name
hydrogen_content	Percentage of hydrogen mass content in the fuel
q_fuel	Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

ei_co2 = 0.0

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o = 9.21

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc = 0.0

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

ei_so2 = 0.0

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$

ei_sulphates = 0.0

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$

fuel_name = 'Hydrogen'

Fuel Name

hydrogen_content = nan

Percentage of hydrogen mass content in the fuel

q_fuel = 122800000.0

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

11.2 Datalib

11.2.1 ECMWF

datalib.ecmwf.ERA5(time, variables[, ...])	Class to support ERA5 data access, download, and organization.
datalib.ecmwf.era5_model_level	Model-level ERA5 data access.
datalib.ecmwf.HRES(time, variables[, ...])	Class to support HRES data access, download, and organization.
datalib.ecmwf.hres_model_level	Model-level HRES data access from the ECMWF operational archive.
datalib.ecmwf.hres.get_forecast_filename(...)	Create forecast filename from ECMWF dissemination products.
datalib.ecmwf.model_levels	Utilities for working with ECMWF model-level data.
datalib.ecmwf.IFS(time, variables[, ...])	ECMWF Integrated Forecasting System (IFS) data source.
datalib.ecmwf.variables	ECMWF Parameter Support.

pycontrails.datalib.ecmwf.ERA5

class pycontrails.datalib.ecmwf.**ERA5**(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *timestep_freq=None*, *product_type='reanalysis'*, *grid=None*, *cachestore=<object object>*, *url=None*, *key=None*)

Bases: `ECMWFAPI`

Class to support ERA5 data access, download, and organization.

Requires account with Copernicus Data Portal and local credentials.

API credentials can be stored in a `~/.cdsapirc` file or as `CDSAPI_URL` and `CDSAPI_KEY` environment variables.

export CDSAPI_URL=... export CDSAPI_KEY=...

Credentials can also be provided directly `url` and `key` keyword args.

See cdsapi documentation for more information.

Parameters

- **time** (`datalib.TimeInput | None`) – The time range for data retrieval, either a single datetime or (start, end) datetime range. Input must be datetime-like or tuple of datetime-like (*datetime*, `pd.Timestamp`, `np.datetime64`) specifying the (start, end) of the date range,

inclusive. Datafiles will be downloaded from CDS for each day to reduce requests. If None, `paths` must be defined and all time coordinates will be loaded from files.

- **variables** (`datalib.VariableInput`) – Variable name (i.e. "t", "air_temperature", ["air_temperature, relative_humidity"])
- **pressure_levels** (`datalib.PressureLevelInput`, *optional*) – Pressure levels for data, in hPa (mbar) Set to -1 for to download surface level parameters. Defaults to -1.
- **paths** (`str | list[str] | pathlib.Path | list[pathlib.Path] | None`, *optional*) – Path to CDS NetCDF files to load manually. Can include glob patterns to load specific files. Defaults to None, which looks for files in the `cachestore` or CDS.
- **timestep_freq** (`str`, *optional*) – Manually set the timestep interval within the bounds defined by `time`. Supports any string that can be passed to *pd.date_range(freq=...)*. By default, this is set to "1h" for reanalysis products and "3h" for ensemble products.
- **product_type** (`str`, *optional*) – Product type, one of "reanalysis", "ensemble_mean", "ensemble_members", "ensemble_spread"
- **grid** (`float`, *optional*) – Specify latitude/longitude grid spacing in data. By default, this is set to 0.25 for reanalysis products and 0.5 for ensemble products.
- **cachestore** (`cache.CacheStore | None`, *optional*) – Cache data store for staging ECMWF ERA5 files. Defaults to `cache.DiskCacheStore`. If None, cache is turned off.
- **url** (`str`) – Override cdsapi url
- **key** (`str`) – Override cdsapi key

Notes

ERA5 parameter list: https://confluence.ecmwf.int/pages/viewpage.action?pageId=82870405#ERA5:-datadocumentation-Parameterlistings

All radiative quantities are accumulated. See https://www.ecmwf.int/sites/default/files/elibrary/2015/18490-radiation-quantities-ecmwf-model-and-mars.pdf for more information.

Local `paths` are loaded using `xarray.open_mfdataset()`. Pass `xr_kwargs` inputs to *`open_metdataset()`* to customize file loading.

Examples

```
>>> from datetime import datetime
>>> from pycontrails.datalib.ecmwf import ERA5
>>> from pycontrails import GCPCacheStore
```

```
>>> # Store data files from CDS to local disk (default behavior)
>>> era5 = ERA5(
...     "2020-06-01 12:00:00",
...     variables=["air_temperature", "relative_humidity"],
...     pressure_levels=[350, 300]
... )
```

```
>>> # cache files to google cloud storage
>>> gcp_cache = GCPCacheStore(
```

(continues on next page)

(continued from previous page)

```
...     bucket="contrails-301217-unit-test",
...     cache_dir="ecmwf",
... )
>>> era5 = ERA5(
...     "2020-06-01 12:00:00",
...     variables=["air_temperature", "relative_humidity"],
...     pressure_levels=[350, 300],
...     cachestore=gcp_cache
... )
```

__init__(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *timestep_freq=None*, *product_type='reanalysis'*, *grid=None*, *cachestore=<object object>*, *url=None*, *key=None*)

Methods

__init__(time, variables[, pressure_levels, ...])	
cache_dataset(dataset)	Cache data from data source.
create_cachepath(t)	Return cachepath to local ERA5 data file based on datetime.
download(**xr_kwargs)	Confirm all data files are downloaded and available locally in the `cachestore`.
download_dataset(times)	Download data from data source for input times.
is_datafile_cached(t, **xr_kwargs)	Check datafile defined by datetime for variables and pressure levels in class.
list_timesteps_cached(**xr_kwargs)	Get a list of data files available locally in the `cachestore`.
list_timesteps_not_cached(**xr_kwargs)	Get a list of data files not available locally in the `cachestore`.
open_dataset(disk_paths, **xr_kwargs)	Open multi-file dataset in xarray.
open_metdataset([dataset, xr_kwargs])	Open MetDataset from data source.
set_metadata(ds)	Set met source metadata on `ds.attrs`.

Attributes

product_type	Product type, one of "reanalysis", "ensemble_mean", "ensemble_members", "ensemble_spread"
cds	Handle to `cdsapi.Client`
url	User provided `cdsapi.Client` url
key	User provided `cdsapi.Client` url
dataset	Select dataset for download based on `pressure_levels`.
grid	Lat / Lon grid spacing
hash	Generate a unique hash for this datasource.
is_single_level	Return True if the datasource is single level data.
paths	Path to local source files to load.
pressure_level_variables	ECMWF pressure level parameters.
pressure_levels	List of pressure levels.
single_level_variables	ECMWF surface level parameters.
supported_pressure_levels	Get pressure levels available from ERA5 pressure level dataset.
supported_variables	Parameters available from data source.
timesteps	List of individual timesteps from data source derived from `time` Use `parse_time()` to handle `TimeInput`.
variable_ecmwfids	Return a list of variable ecmwf_ids.
variable_shortnames	Return a list of variable short names.
variable_standardnames	Return a list of variable standard names.
variables	Variables requested from data source Use `parse_variables()` to handle `VariableInput`.
cachestore	Cache store for intermediates while processing data source If None, cache is turned off.

cds

Handle to `cdsapi.Client`

create_cachepath(*t*)

Return cachepath to local ERA5 data file based on datetime.

This uniquely defines a cached data file ith class parameters.

Parameters
t (`datetime | pd.Timestamp`) – Datetime of datafile

Returns
`str` – Path to local ERA5 data file

property dataset

Select dataset for download based on `pressure_levels`.

One of “reanalysis-era5-pressure-levels” or “reanalysis-era5-single-levels”

Returns
`str` – ERA5 dataset name in CDS

download_dataset(*times*)

Download data from data source for input times.

Parameters
times (list[datetime]) – List of datetimes to download a store in cache

property hash

Generate a unique hash for this datasource.

Returns
str – Unique hash for met instance (sha1)

key

User provided cdsapi.Client url

open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (xr.Dataset | None, *optional*) – Input xr.Dataset loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (dict[str, Any] | None, *optional*) – Dictionary of keyword arguments passed into xarray.open_mfdataset() when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if dataset is input.
- ****kwargs** (Any) – Keyword arguments passed through directly into MetDataset constructor.

Returns
MetDataset – Meteorology dataset

See also:

xarray.open_mfdataset()

property pressure_level_variables

ECMWF pressure level parameters.

Returns
list[MetVariable] | None – List of MetVariable available in datasource

product_type

Product type, one of "reanalysis", "ensemble_mean", "ensemble_members", "ensemble_spread"

set_metadata(*ds*)

Set met source metadata on ds.attrs.

This is called within the *open_metdataset()* method to set metadata on the returned MetDataset instance.

Parameters
ds (xr.Dataset | MetDataset) – Dataset to set metadata on. Mutated in place.

property single_level_variables

ECMWF surface level parameters.

Returns
list[MetVariable] | None – List of MetVariable available in datasource

property supported_pressure_levels

Get pressure levels available from ERA5 pressure level dataset.

Returns
`list[int]` – List of integer pressure level values

url

User provided `cdsapi.Client` url

pycontrails.datalib.ecmwf.era5_model_level

Model-level ERA5 data access.

This module supports

- Retrieving model-level ERA5 data by submitting MARS requests through the Copernicus CDS.
- Processing retrieved GRIB files to produce netCDF files on target pressure levels.
- Local caching of processed netCDF files.
- Opening processed and cached files as a *pycontrails.MetDataset* object.

Consider using `pycontrails.datalib.ecmwf.ARCOERA5` to access model-level data from the nominal ERA5 reanalysis between 1959 and 2022. `pycontrails.datalib.ecmwf.ARCOERA5` accesses data through Google's Analysis-Ready, Cloud Optimized ERA5 dataset and has lower latency than this module, which retrieves data from the Copernicus Climate Data Store. This module must be used to retrieve model-level data from ERA5 ensemble members or for more recent dates.

This module requires the following additional dependency:

- metview (binaries and python bindings)

Classes

ERA5ModelLevel(time, variables[, ...])	Class to support model-level ERA5 data access, download, and organization.

class pycontrails.datalib.ecmwf.era5_model_level.**ERA5ModelLevel**(*time*, *variables*, *pressure_levels=None*, *timestep_freq=None*, *product_type='reanalysis'*, *grid=None*, *levels=None*, *ensemble_members=None*, *cachestore=<object object>*, *n_jobs=1*, *cache_grib=False*, *url=None*, *key=None*)

Bases: `ECMWFAPI`

Class to support model-level ERA5 data access, download, and organization.

The interface is similar to *pycontrails.datalib.ecmwf.ERA5*, which downloads pressure-level with much lower vertical resolution.

Requires account with Copernicus Data Portal and local credentials.

API credentials can be stored in a `~/.cdsapirc` file or as `CDSAPI_URL` and `CDSAPI_KEY` environment variables.

export CDSAPI_URL=... export CDSAPI_KEY=...

Credentials can also be provided directly `url` and `key` keyword args.

See cdsapi documentation for more information.

Parameters

- **time** (`datalib.TimeInput | None`) – The time range for data retrieval, either a single datetime or (start, end) datetime range. Input must be datetime-like or tuple of datetime-like (`datetime.datetime`, `pandas.Timestamp`, `numpy.datetime64`) specifying the (start, end) of the date range, inclusive. GRIB files will be downloaded from CDS in chunks no larger than 1 month for the nominal reanalysis and no larger than 1 day for ensemble members. This ensures that exactly one request is submitted per file on tape accessed. If None, `paths` must be defined and all time coordinates will be loaded from files.
- **variables** (`datalib.VariableInput`) – Variable name (i.e. "t", "air_temperature", ["air_temperature, specific_humidity"])
- **pressure_levels** (`datalib.PressureLevelInput`, *optional*) – Pressure levels for data, in hPa (mbar). To download surface-level parameters, use *pycontrails.datalib.ecmwf.ERA5*. Defaults to pressure levels that match model levels at a nominal surface pressure.
- **timestep_freq** (`str`, *optional*) – Manually set the timestep interval within the bounds defined by `time`. Supports any string that can be passed to `pd.date_range(freq=...)`. By default, this is set to "1h" for reanalysis products and "3h" for ensemble products.
- **product_type** (`str`, *optional*) – Product type, one of "reanalysis" and "ensemble_members". Unlike *pycontrails.datalib.ecmwf.ERA5*, this class does not support direct access to the ensemble mean and spread, which are not available on model levels.
- **grid** (`float`, *optional*) – Specify latitude/longitude grid spacing in data. By default, this is set to 0.25 for reanalysis products and 0.5 for ensemble products.
- **levels** (`list[int]`, *optional*) – Specify ECMWF model levels to include in MARS requests. By default, this is set to include all model levels.
- **ensemble_members** (`list[int]`, *optional*) – Specify ensemble members to include. Valid only when the product type is "ensemble_members". By default, includes every available ensemble member.
- **cachestore** (`cache.CacheStore | None`, *optional*) – Cache data store for staging processed netCDF files. Defaults to *pycontrails.core.cache.DiskCacheStore*. If None, cache is turned off.
- **cache_grib** (`bool`, *optional*) – If True, cache downloaded GRIB files rather than storing them in a temporary file. By default, False.
- **url** (`str`) – Override cdsapi url
- **key** (`str`) – Override cdsapi key

`create_cachepath(t)`

Return cachepath to local ERA5 data file based on datetime.

This uniquely defines a cached data file with class parameters.

Parameters

t (`datetime | pd.Timestamp`) – Datetime of datafile

Returns

`str` – Path to local ERA5 data file

property dataset

Select dataset for downloading model-level data.

Always returns "reanalysis-era5-complete".

Returns
`str` – Model-level ERA5 dataset name in CDS

download_dataset(*times*)

Download data from data source for input times.

Parameters
times (`list[datetime]`) – List of datetimes to download a store in cache

grid

Lat / Lon grid spacing

mars_request(*times*)

Generate MARS request for specific list of times.

Parameters
times (`list[datetime]`) – Times included in MARS request.

Returns
`dict[str, str]` – MARS request for submission to Copernicus CDS.

open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (`xr.Dataset | None`, *optional*) – Input `xr.Dataset` loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (`dict[str, Any] | None`, *optional*) – Dictionary of keyword arguments passed into `xarray.open_mfdataset()` when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if `dataset` is input.
- ****kwargs** (`Any`) – Keyword arguments passed through directly into `MetDataset` constructor.

Returns
`MetDataset` – Meteorology dataset

See also:

`xarray.open_mfdataset()`

paths

Path to local source files to load. Set to the paths of files cached in `cachestore` if no `paths` input is provided on init.

property pressure_level_variables

ECMWF pressure level parameters available on model levels.

Returns
`list[MetVariable]` – List of MetVariable available in datasource

pressure_levels

List of pressure levels. Set to [-1] for data without level coordinate. Use `parse_pressure_levels()` to handle `PressureLevelInput`.

set_metadata(*ds*)

Set met source metadata on `ds.attrs`.

This is called within the *open_metdataset()* method to set metadata on the returned `MetDataset` instance.

Parameters

ds (`xr.Dataset | MetDataset`) – Dataset to set metadata on. Mutated in place.

property single_level_variables

ECMWF single-level parameters available on model levels.

Returns

`list[MetVariable]` – Always returns an empty list. To access single-level variables, used *pycontrails.datalib.ecmwf.ERA5*.

timesteps

List of individual timesteps from data source derived from `time` Use `parse_time()` to handle `TimeInput`.

variables

Variables requested from data source Use `parse_variables()` to handle `VariableInput`.

pycontrails.datalib.ecmwf.HRES

class pycontrails.datalib.ecmwf.**HRES**(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *cachepath=None*, *grid=0.25*, *stream='oper'*, *field_type='fc'*, *forecast_time=None*, *cachestore=<object object>*, *url=None*, *key=None*, *email=None*)

Bases: `ECMWFAPI`

Class to support HRES data access, download, and organization.

Requires account with ECMWF and API key.

API credentials set in local `~/.ecmwfapirc` file:

```
{
    "url": "https://api.ecmwf.int/v1",
    "email": "<email>",
    "key": "<key>"
}
```

Credentials can also be provided directly `url` `key`, and `email` keyword args.

See ecmwf-api-client documentation for more information.

Parameters

- **time** (`datalib.TimeInput | None`) – The time range for data retrieval, either a single datetime or (start, end) datetime range. Input must be a datetime-like or tuple of datetime-like (datetime, `pandas.Timestamp`, `numpy.datetime64`) specifying the (start, end) of the date range, inclusive. If `forecast_time` is unspecified, the forecast time will be assumed to be the nearest synoptic hour: 00, 06, 12, 18. All subsequent times will be downloaded for relative to *forecast_time*. If None, `paths` must be defined and all time coordinates will be loaded from files.

- **variables** (`datalib.VariableInput`) – Variable name (i.e. "air_temperature", ["air_temperature, relative_humidity"]) See *pressure_level_variables* for the list of available variables.
- **pressure_levels** (`datalib.PressureLevelInput`, *optional*) – Pressure levels for data, in hPa (mbar) Set to -1 for to download surface level parameters. Defaults to -1.
- **paths** (`str | list[str] | pathlib.Path | list[pathlib.Path] | None`, *optional*) – Path to CDS NetCDF files to load manually. Can include glob patterns to load specific files. Defaults to None, which looks for files in the `cachestore` or CDS.
- **grid** (`float`, *optional*) – Specify latitude/longitude grid spacing in data. Defaults to 0.25.
- **stream** (`str`, *optional*) – "oper" = atmospheric model/HRES, "enfo" = ensemble forecast. Defaults to "oper" (HRES),
- **field_type** (`str`, *optional*) – Field type can be e.g. forecast (fc), perturbed forecast (pf), control forecast (cf), analysis (an). Defaults to "fc".
- **forecast_time** (`DatetimeLike`, *optional*) – Specify forecast run by runtime. Defaults to None.
- **cachestore** (`cache.CacheStore | None`, *optional*) – Cache data store for staging data files. Defaults to `cache.DiskCacheStore`. If None, cache is turned off.
- **url** (`str`) – Override ecmwf-api-client url
- **key** (`str`) – Override ecmwf-api-client key
- **email** (`str`) – Override ecmwf-api-client email

Notes

MARS key word definitions

- class: in most cases this will be operational data, or "od"
- stream: "enfo" = ensemble forecast, "oper" = atmospheric model/HRES
- expver: experimental version, production data is 1 or 2
- date: there are numerous acceptible date formats
- time: forecast base time, always in synoptic time (0,6,12,18 UTC)
- type: forecast (oper), perturbed or control forecast (enfo only), or analysis
- levtype: options include surface, pressure levels, or model levels
- levelist: list of levels in format specified by **levtype** levelist
- param: list of variables in catalog number, long name or short name
- step: hourly time steps from base forecast time
- number: for ensemble forecasts, ensemble numbers
- format: specify netcdf instead of default grib, DEPRECATED format
- grid: specify model return grid spacing

Local `paths` are loaded using `xarray.open_mfdataset()`. Pass `xr_kwargs` inputs to *open_metdataset()* to customize file loading.

Examples

```
>>> from datetime import datetime
>>> from pycontrails import GCPCacheStore
>>> from pycontrails.datalib.ecmwf import HRES
```

```
>>> # Store data files to local disk (default behavior)
>>> times = (datetime(2021, 5, 1, 2), datetime(2021, 5, 1, 3))
>>> hres = HRES(times, variables="air_temperature", pressure_levels=[300, 250])
```

```
>>> # Cache files to google cloud storage
>>> gcp_cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="ecmwf",
... )
>>> hres = HRES(
...     times,
...     variables="air_temperature",
...     pressure_levels=[300, 250],
...     cachestore=gcp_cache
... )
```

__init__(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *cachepath=None*, *grid=0.25*, *stream='oper'*, *field_type='fc'*, *forecast_time=None*, *cachestore=<object object>*, *url=None*, *key=None*, *email=None*)

Methods

__init__(time, variables[, pressure_levels, ...])	
cache_dataset(dataset)	Cache data from data source.
create_cachepath(t)	Return cachepath to local data file based on datetime.
create_synoptic_time_ranges(timesteps)	Create synoptic time bounds encompassing date range.
download(**xr_kwargs)	Confirm all data files are downloaded and available locally in the `cachestore`.
download_dataset(times)	Download data from data source for input times.
generate_mars_request([forecast_time, ...])	Generate MARS request in MARS request syntax.
is_datafile_cached(t, **xr_kwargs)	Check datafile defined by datetime for variables and pressure levels in class.
list_from_mars()	List metadata on query from MARS.
list_timesteps_cached(**xr_kwargs)	Get a list of data files available locally in the `cachestore`.
list_timesteps_not_cached(**xr_kwargs)	Get a list of data files not available locally in the `cachestore`.
open_dataset(disk_paths, **xr_kwargs)	Open multi-file dataset in xarray.
open_metdataset([dataset, xr_kwargs])	Open MetDataset from data source.
set_metadata(ds)	Set met source metadata on `ds.attrs`.

Attributes

server	Handle to ECMWFService client
stream	stream type, "oper" = atmospheric model/HRES, "enfo" = ensemble forecast.
field_type	Field type, forecast ("fc"), perturbed forecast ("pf"), control forecast ("cf"), analysis ("an").
forecast_time	Forecast run time, either specified or assigned by the closest previous forecast run
url	
key	
email	
grid	Lat / Lon grid spacing
hash	Generate a unique hash for this datasource.
is_single_level	Return True if the datasource is single level data.
paths	Path to local source files to load.
pressure_level_variables	ECMWF pressure level parameters.
pressure_levels	List of pressure levels.
single_level_variables	ECMWF surface level parameters.
step_offset	Difference between *forecast_time* and first timestep.
steps	Forecast steps from *forecast_time* corresponding within input `time`.
supported_pressure_levels	Get pressure levels available from MARS.
supported_variables	Parameters available from data source.
timesteps	List of individual timesteps from data source derived from `time` Use `parse_time()` to handle `TimeInput`.
variable_ecmwfids	Return a list of variable ecmwf_ids.
variable_shortnames	Return a list of variable short names.
variable_standardnames	Return a list of variable standard names.
variables	Variables requested from data source Use `parse_variables()` to handle `VariableInput`.
cachestore	Cache store for intermediates while processing data source If None, cache is turned off.

create_cachepath(*t*)

Return cachepath to local data file based on datetime.

Parameters
t (`datetime`) – Datetime of datafile

Returns
`str` – Path to cached data file

classmethod create_synoptic_time_ranges(*timesteps*)

Create synoptic time bounds encompassing date range.

Extracts time bounds for synoptic time range ([00:00, 11:59], [12:00, 23:59]) for a list of input timesteps.

Parameters

timesteps (`list[pd.Timestamp]`) – List of timesteps formatted as `pd.Timestamps`. Often this it the output from *pd.date_range()*

Returns

`list[tuple[pd.Timestamp, pd.Timestamp]]` – List of tuple time bounds that can be used as inputs to `HRES(time=...)`

download_dataset(*times*)

Download data from data source for input times.

Parameters

times (`list[:class:`datetime]`) – List of datetimes to download and store in cache datastore

email

field_type

Field type, forecast ("fc"), perturbed forecast ("pf"), control forecast ("cf"), analysis ("an").

forecast_time

Forecast run time, either specified or assigned by the closest previous forecast run

generate_mars_request(*forecast_time=None*, *steps=None*, *request_type='retrieve'*, *request_format='mars'*)

Generate MARS request in MARS request syntax.

Parameters

- **forecast_time** (`datetime`, *optional*) – Base datetime for the forecast. Defaults to *forecast_time*.
- **steps** (`list[int]`, *optional*) – list of steps. Defaults to *steps*.
- **request_type** (`str`, *optional*) – "retrieve" for download request or "list" for metadata request. Defaults to "retrieve".
- **request_format** (`str`, *optional*) – "mars" for MARS string format, or "dict" for dict version. Defaults to "mars".

Returns

`str | dict[str, Any]` – Returns MARS query string if `request_format` is "mars". Returns dict query if `request_format` is "dict"

Notes

Brief overview of MARS request syntax

property hash

Generate a unique hash for this datasource.

Returns

`str` – Unique hash for met instance (sha1)

key

list_from_mars()

List metadata on query from MARS.

Returns

`str` – Metadata for MARS request. Note this is queued the same as data requests.

open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (`xr.Dataset | None`, *optional*) – Input `xr.Dataset` loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (`dict[str, Any] | None`, *optional*) – Dictionary of keyword arguments passed into `xarray.open_mfdataset()` when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if `dataset` is input.
- ****kwargs** (`Any`) – Keyword arguments passed through directly into `MetDataset` constructor.

Returns

MetDataset – Meteorology dataset

See also:

`xarray.open_mfdataset()`

property pressure_level_variables

ECMWF pressure level parameters.

Returns

`list[MetVariable] | None` – List of MetVariable available in datasource

server

Handle to ECMWFService client

set_metadata(*ds*)

Set met source metadata on `ds.attrs`.

This is called within the *open_metdataset()* method to set metadata on the returned `MetDataset` instance.

Parameters

ds (`xr.Dataset | MetDataset`) – Dataset to set metadata on. Mutated in place.

property single_level_variables

ECMWF surface level parameters.

Returns

`list[MetVariable] | None` – List of MetVariable available in datasource

property step_offset

Difference between *forecast_time* and first timestep.

Returns

`int` – Number of steps to offset in order to retrieve data starting from input time. Returns 0 if `timesteps` is empty when loading from `paths`.

property steps

Forecast steps from *forecast_time* corresponding within input `time`.

Returns

`list[int]` – List of forecast steps relative to *forecast_time*

stream

stream type, "oper" = atmospheric model/HRES, "enfo" = ensemble forecast.

property supported_pressure_levels

Get pressure levels available from MARS.

Returns

`list[int]` – List of integer pressure level values

url

pycontrails.datalib.ecmwf.hres_model_level

Model-level HRES data access from the ECMWF operational archive.

This module supports

- Retrieving model-level HRES data by submitting MARS requests through the ECMWF API.
- Processing retrieved GRIB files to produce netCDF files on target pressure levels.
- Local caching of processed netCDF files.
- Opening processed and cached files as a *pycontrails.MetDataset* object.

This module requires the following additional dependency:

- metview (binaries and python bindings)

Classes

HRESModelLevel(time, variables[, ...])	Class to support model-level HRES data access, download, and organization.

class pycontrails.datalib.ecmwf.hres_model_level.**HRESModelLevel**(*time*, *variables*, *pressure_levels=None*, *timestep_freq=None*, *grid=None*, *forecast_time=None*, *levels=None*, *ensemble_members=None*, *cachestore=<object object>*, *cache_grib=False*, *url=None*, *key=None*, *email=None*)

Bases: `ECMWFAPI`

Class to support model-level HRES data access, download, and organization.

The interface is similar to *pycontrails.datalib.ecmwf.HRES*, which downloads pressure-level data with much lower vertical resolution and single-level data. Note, however, that only a subset of the pressure-level data available through the operational archive is available as model-level data. As a consequence, this interface only supports access to nominal HRES forecasts (corresponding to `stream = "oper"` and `field_type = "fc"` in *pycontrails.datalib.ecmwf.HRES*) initialized at 00z and 12z.

Requires account with ECMWF and API key.

API credentials can be set in local `~/.ecmwfapirc` file:

```
{
    "url": "https://api.ecmwf.int/v1",
    "email": "<email>",
    "key": "<key>"
}
```

Credentials can also be provided directly in `url`, `key`, and `email` keyword args.

See ecmwf-api-client documentation for more information.

Parameters

- **time** (`datalib.TimeInput`) – The time range for data retrieval, either a single datetime or (start, end) datetime range. Input must be datetime-like or tuple of datetime-like (`datetime.datetime`, `pandas.Timestamp`, `numpy.datetime64`) specifying the (start, end) of the date range, inclusive. All times will be downloaded in a single GRIB file, which ensures that exactly one request is submitted per file on tape accessed. If `forecast_time` is unspecified, the forecast time will be assumed to be the nearest synoptic hour available in the operational archive (00 or 12). All subsequent times will be downloaded for relative to `forecast_time`.
- **variables** (`datalib.VariableInput`) – Variable name (i.e. "t", "air_temperature", ["air_temperature, specific_humidity"])
- **pressure_levels** (`datalib.PressureLevelInput`, *optional*) – Pressure levels for data, in hPa (mbar). To download surface-level parameters, use *pycontrails.datalib.ecmwf.HRES*. Defaults to pressure levels that match model levels at a nominal surface pressure.
- **timestep_freq** (`str`, *optional*) – Manually set the timestep interval within the bounds defined by `time`. Supports any string that can be passed to `pandas.date_range(freq=...)`. By default, this is set to the highest frequency that can supported the requested time range ("1h" out to 96 hours, "3h" out to 144 hours, and "6h" out to 240 hours)
- **grid** (`float`, *optional*) – Specify latitude/longitude grid spacing in data. By default, this is set to 0.1.
- **forecast_time** (`DatetimeLike`, *optional*) – Specify forecast by initialization time. By default, set to the most recent forecast that includes the requested time range.
- **levels** (`list[int]`, *optional*) – Specify ECMWF model levels to include in MARS requests. By default, this is set to include all model levels.
- **cachestore** (`CacheStore | None`, *optional*) – Cache data store for staging processed netCDF files. Defaults to *pycontrails.core.cache.DiskCacheStore*. If None, cache is turned off.
- **cache_grib** (`bool`, *optional*) – If True, cache downloaded GRIB files rather than storing them in a temporary file. By default, False.
- **url** (`str`) – Override ecmwf-api-client url
- **key** (`str`) – Override ecmwf-api-client key
- **email** (`str`) – Override ecmwf-api-client email

`create_cachepath(t)`

Return cachepath to local HRES data file based on datetime.

This uniquely defines a cached data file with class parameters.

Parameters

t (`datetime | pd.Timestamp`) – Datetime of datafile

Returns
str – Path to local HRES data file

`download_dataset`(*times*)

Download data from data source for input times.

Parameters
times (list[datetime]) – List of datetimes to download a store in cache

`get_forecast_steps`(*times*)

Convert list of times to list of forecast steps.

Parameters
times (list[datetime]) – Times to convert to forecast steps

Returns
list[int] – Forecast step at each time

`grid`

Lat / Lon grid spacing

`mars_request`(*times*)

Generate MARS request for specific list of times.

Parameters
times (list[datetime]) – Times included in MARS request.

Returns
str – MARS request for submission to ECMWF API.

`open_metdataset`(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (xr.Dataset | None, *optional*) – Input xr.Dataset loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (dict[str, Any] | None, *optional*) – Dictionary of keyword arguments passed into xarray.open_mfdataset() when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if dataset is input.
- ****kwargs** (Any) – Keyword arguments passed through directly into MetDataset constructor.

Returns
MetDataset – Meteorology dataset

See also:

xarray.open_mfdataset()

`paths`

Path to local source files to load. Set to the paths of files cached in cachestore if no paths input is provided on init.

`property pressure_level_variables`

ECMWF pressure level parameters available on model levels.

Returns
list[MetVariable] – List of MetVariable available in datasource

pressure_levels

List of pressure levels. Set to [-1] for data without level coordinate. Use parse_pressure_levels() to handle PressureLevelInput.

set_metadata(*ds*)

Set met source metadata on ds.attrs.

This is called within the *open_metdataset()* method to set metadata on the returned MetDataset instance.

Parameters
ds (xr.Dataset | MetDataset) – Dataset to set metadata on. Mutated in place.

property single_level_variables

ECMWF single-level parameters available on model levels.

Returns
list[MetVariable] – Always returns an empty list. To access single-level variables, use *pycontrails.datalib.ecmwf.HRES*.

property step_offset

Difference between forecast_time and first timestep.

Returns
int – Number of steps to offset in order to retrieve data starting from input time.

property steps

Forecast steps from forecast_time corresponding within input time.

Returns
list[int] – List of forecast steps relative to forecast_time

timesteps

List of individual timesteps from data source derived from time Use parse_time() to handle TimeInput.

variables

Variables requested from data source Use parse_variables() to handle VariableInput.

pycontrails.datalib.ecmwf.hres.get_forecast_filename

pycontrails.datalib.ecmwf.hres.**get_forecast_filename**(*forecast_time*, *timestep*, *cc='A1'*, *S=None*, *E='1'*)

Create forecast filename from ECMWF dissemination products.

See ECMWF Dissemination for more information:

```
The following dissemination filename convention is used for the
transmission of ECMWF dissemination products:

ccSMMDDHHIImmddhhiiE where:

    cc is Dissemination stream name
    S is dissemination data stream indicator
    MMDDHHII is month, day, hour and minute on which the products are based
```

(continues on next page)

(continued from previous page)

```
    mmddhhii is month, day, hour and minute on which the products are valid at
        ddhhii is set to "______" for Seasonal Forecasting System products
        ii is set to 01 for high resolution forecast time step zero, type=fc, step=0
    E is the Experiment Version Number' (as EXPVER in MARS, normally 1)
```

Parameters

- **forecast_time** (`datetime`) – Forecast time to stage
- **timestep** (`datetime`) – Time within forecast
- **cc** (`str`, *optional*) – Dissemination stream name. Defaults to "A1"
- **S** (`str`, *optional*) – Dissemination data stream indicator. If None, S is set to "S" for 6 or 18 hour forecast and "D" for 0 or 12 hour forecast.
- **E** (`str`, *optional*) – Experiment Version Number. Defaults to "1"

Returns

`str` – Filename to forecast file

Raises

`ValueError` – If `forecast_time` is not on a synoptic hour (00, 06, 12, 18) If `timestep` is before `forecast_time`

pycontrails.datalib.ecmwf.model_levels

Utilities for working with ECMWF model-level data.

This module requires the following additional dependency:

- lxml

Functions

pressure_levels_at_model_levels(alt_ft_min, ...)	Return the pressure levels at each model level assuming a constant surface pressure.

pycontrails.datalib.ecmwf.model_levels.**pressure_levels_at_model_levels**(*alt_ft_min*, *alt_ft_max*)

Return the pressure levels at each model level assuming a constant surface pressure.

The pressure levels are rounded to the nearest hPa.

Parameters

- **alt_ft_min** (`float`) – Minimum altitude, $[ft]$.
- **alt_ft_max** (`float`) – Maximum altitude, $[ft]$.

Returns

`list[int]` – List of pressure levels, $[hPa]$.

pycontrails.datalib.ecmwf.IFS

class pycontrails.datalib.ecmwf.**IFS**(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *grid=None*, *forecast_path=None*, *forecast_date=None*)

Bases: *MetDataSource*

ECMWF Integrated Forecasting System (IFS) data source.

Warning: This data source is not fully implemented.

Parameters

- **time** (`datalib.TimeInput | None`) – The time range for data retrieval, either a single datetime or (start, end) datetime range. Input must be a single datetime-like or tuple of datetime-like (datetime, `pandas.Timestamp`, `numpy.datetime64`) specifying the (start, end) of the date range, inclusive. If None, all time coordinates will be loaded.
- **variables** (`datalib.VariableInput`) – Variable name (i.e. "air_temperature", ["air_temperature, relative_humidity"]) See `pressure_level_variables` for the list of available variables.
- **pressure_levels** (`datalib.PressureLevelInput`, *optional*) – Pressure level bounds for data (min, max), in hPa (mbar) Set to -1 for to download surface level parameters. Defaults to -1.
- **paths** (`str | list[str] | pathlib.Path | list[pathlib.Path] | None`, *optional*) – UNSUPPORTED FOR IFS
- **forecast_path** (`str | pathlib.Path | None`, *optional*) – Path to local forecast files. Defaults to None
- **forecast_date** (`DatetimeLike`, *optional*) – Forecast date to load specific netcdf files. Defaults to None

Notes

This takes an average pressure of the model level to create pressure level dimensions.

__init__(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *grid=None*, *forecast_path=None*, *forecast_date=None*)

Methods

__init__(time, variables[, pressure_levels, ...])	
cache_dataset(dataset)	Cache data from data source.
create_cachepath(t)	Return cachepath to local data file based on datetime.
download(**xr_kwargs)	Confirm all data files are downloaded and available locally in the cachestore.
download_dataset(times)	Download data from data source for input times.
is_datafile_cached(t, **xr_kwargs)	Check datafile defined by datetime for variables and pressure levels in class.
list_timesteps_cached(**xr_kwargs)	Get a list of data files available locally in the cachestore.
list_timesteps_not_cached(**xr_kwargs)	Get a list of data files not available locally in the cachestore.
open_dataset(disk_paths, **xr_kwargs)	Open multi-file dataset in xarray.
open_metdataset([dataset, xr_kwargs])	Open MetDataset from data source.
set_metadata(ds)	Set met source metadata on ds.attrs.

Attributes

forecast_date	Forecast datetime of IFS forecast
forecast_path	Root path of IFS data
grid	Lat / Lon grid spacing
hash	Generate a unique hash for this datasource.
is_single_level	Return True if the datasource is single level data.
paths	Path to local source files to load.
pressure_level_variables	Parameters available from data source.
pressure_levels	List of pressure levels.
single_level_variables	Parameters available from data source.
supported_pressure_levels	IFS does not provide constant pressure levels and instead uses model levels.
supported_variables	IFS parameters available.
timesteps	List of individual timesteps from data source derived from time Use parse_time() to handle TimeInput.
variable_shortnames	Return a list of variable short names.
variable_standardnames	Return a list of variable standard names.
variables	Variables requested from data source Use parse_variables() to handle VariableInput.
cachestore	Cache store for intermediates while processing data source If None, cache is turned off.

cache_dataset(*dataset*)

Cache data from data source.

Parameters

dataset (xarray.Dataset) – Dataset loaded from remote API or local files. The dataset must have the same format as the original data source API or files.

create_cachepath(*t*)

Return cachepath to local data file based on datetime.

Parameters
t (`datetime`) – Datetime of datafile

Returns
`str` – Path to cached data file

download_dataset(*times*)

Download data from data source for input times.

Parameters
times (`list[datetime]`) – List of datetimes to download a store in cache

forecast_date

Forecast datetime of IFS forecast

forecast_path

Root path of IFS data

open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (`xr.Dataset | None`, *optional*) – Input `xr.Dataset` loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (`dict[str, Any] | None`, *optional*) – Dictionary of keyword arguments passed into `xarray.open_mfdataset()` when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if `dataset` is input.
- ****kwargs** (`Any`) – Keyword arguments passed through directly into `MetDataset` constructor.

Returns
MetDataset – Meteorology dataset

See also:

`xarray.open_mfdataset()`

set_metadata(*ds*)

Set met source metadata on `ds.attrs`.

This is called within the *open_metdataset()* method to set metadata on the returned `MetDataset` instance.

Parameters
ds (`xr.Dataset | MetDataset`) – Dataset to set metadata on. Mutated in place.

property supported_pressure_levels

IFS does not provide constant pressure levels and instead uses model levels.

Returns
`list[int]`

property supported_variables

IFS parameters available.

Returns

`list[MetVariable] | None` – List of MetVariable available in datasource

pycontrails.datalib.ecmwf.variables

ECMWF Parameter Support.

Sourced from the ECMWF Parameter DB:
https://apps.ecmwf.int/codes/grib/param-db

11.2.2 GFS

datalib.gfs.GFSForecast(time, variables[, ...])	GFS Forecast data access.
datalib.gfs.variables	GFS Parameter Support.

pycontrails.datalib.gfs.GFSForecast

class pycontrails.datalib.gfs.**GFSForecast**(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *grid=0.25*, *forecast_time=None*, *cachestore=<object object>*, *show_progress=False*)

Bases: *MetDataSource*

GFS Forecast data access.

Parameters

- **time** (*datalib.TimeInput*) – The time range for data retrieval, either a single datetime or (start, end) datetime range. Input must be a single datetime-like or tuple of datetime-like (datetime, `pandas.Timestamp`, `numpy.datetime64`) specifying the (start, end) of the date range, inclusive. All times will be downloaded for a single forecast model run nearest to the start time (see *forecast_time*) If None, `paths` must be defined and all time coordinates will be loaded from files.
- **variables** (*datalib.VariableInput*) – Variable name (i.e. "temperature", ["temperature, relative_humidity"]) See *pressure_level_variables* for the list of available variables.
- **pressure_levels** (*datalib.PressureLevelInput*, *optional*) – Pressure levels for data, in hPa (mbar) Set to [-1] for to download surface level parameters. Defaults to [-1].
- **paths** (`str | list[str] | pathlib.Path | list[pathlib.Path] | None`, *optional*) – Path to files to load manually. Can include glob patterns to load specific files. Defaults to None, which looks for files in the *cachestore* or GFS AWS bucket.
- **grid** (`float`, *optional*) – Specify latitude/longitude grid spacing in data. Defaults to 0.25.
- **forecast_time** (*DatetimeLike*, *optional*) – Specify forecast run by runtime. If None (default), the forecast time is set to the 6 hour floor of the first timestep.
- **cachestore** (`cache.CacheStore` | None, *optional*) – Cache data store for staging data files. Defaults to `cache.DiskCacheStore`. If None, cachestore is turned off.
- **show_progress** (`bool`, *optional*) – Show progress when downloading files from GFS AWS Bucket. Defaults to False

Examples

```
>>> from datetime import datetime
>>> from pycontrails.datalib.gfs import GFSForecast
```

```
>>> # Store data files to local disk (default behavior)
>>> times = ("2022-03-22 00:00:00", "2022-03-22 03:00:00")
>>> gfs = GFSForecast(times, variables="air_temperature", pressure_levels=[300,␣
↪250])
>>> gfs
GFSForecast
    Timesteps: ['2022-03-22 00', '2022-03-22 01', '2022-03-22 02', '2022-03-22 03']
    Variables: ['t']
    Pressure levels: [250, 300]
    Grid: 0.25
    Forecast time: 2022-03-22 00:00:00
```

```
>>> gfs = GFSForecast(times, variables="air_temperature", pressure_levels=[300,␣
↪250], grid=0.5)
>>> gfs
GFSForecast
    Timesteps: ['2022-03-22 00', '2022-03-22 03']
    Variables: ['t']
    Pressure levels: [250, 300]
    Grid: 0.5
    Forecast time: 2022-03-22 00:00:00
```

Notes

- NOAA GFS
- Documentation
- Parameter sets
- GFS Documentation

__init__(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *grid=0.25*, *forecast_time=None*, *cachestore=<object object>*, *show_progress=False*)

Methods

__init__(time, variables[, pressure_levels, ...])	
cache_dataset(dataset)	Cache data from data source.
create_cachepath(t)	Return cachepath to local data file based on datetime.
download(**xr_kwargs)	Confirm all data files are downloaded and available locally in the *cachestore*.
download_dataset(times)	Download data from data source for input times.
filename(t)	Construct grib filename to retrieve from GFS bucket.
is_datafile_cached(t, **xr_kwargs)	Check datafile defined by datetime for variables and pressure levels in class.
list_timesteps_cached(**xr_kwargs)	Get a list of data files available locally in the *cachestore*.
list_timesteps_not_cached(**xr_kwargs)	Get a list of data files not available locally in the *cachestore*.
open_dataset(disk_paths, **xr_kwargs)	Open multi-file dataset in xarray.
open_metdataset([dataset, xr_kwargs])	Open MetDataset from data source.
set_metadata(ds)	Set met source metadata on ds.attrs.

Attributes

client	S3 client for accessing GFS bucket
grid	Lat / Lon grid spacing.
cachestore	Cache store for intermediates while processing data source If None, cache is turned off.
show_progress	Show progress bar when downloading files from AWS
forecast_time	Base time of the previous GFS forecast based on input times
forecast_path	Construct forecast path in bucket for *forecast_time*.
hash	Generate a unique hash for this datasource.
is_single_level	Return True if the datasource is single level data.
paths	Path to local source files to load.
pressure_level_variables	GFS pressure level parameters.
pressure_levels	List of pressure levels.
single_level_variables	GFS surface level parameters.
supported_pressure_levels	Get pressure levels available.
supported_variables	Parameters available from data source.
timesteps	List of individual timesteps from data source derived from time Use parse_time() to handle TimeInput.
variable_shortnames	Return a list of variable short names.
variable_standardnames	Return a list of variable standard names.
variables	Variables requested from data source Use parse_variables() to handle VariableInput.

cache_dataset(*dataset*)

Cache data from data source.

Parameters
dataset (xarray.Dataset) – Dataset loaded from remote API or local files. The dataset must have the same format as the original data source API or files.

`cachestore`

Cache store for intermediates while processing data source If None, cache is turned off.

`client`

S3 client for accessing GFS bucket

`create_cachepath`(*t*)

Return cachepath to local data file based on datetime.

Parameters
t (datetime) – Datetime of datafile

Returns
str – Path to cached data file

`download_dataset`(*times*)

Download data from data source for input times.

Parameters
times (list[datetime]) – List of datetimes to download a store in cache

`filename`(*t*)

Construct grib filename to retrieve from GFS bucket.

String template:

gfs.tCCz.pgrb2.GGGG.fFFF

- CC is the model cycle runtime (i.e. 00, 06, 12, 18)
- GGGG is the grid spacing
- FFF is the forecast hour of product from 000 - 384

Parameters
t (datetime) – Timestep to download

Returns
str – Forecast filenames to retrieve from GFS bucket.

References

- https://www.nco.ncep.noaa.gov/pmb/products/gfs/

`property forecast_path`

Construct forecast path in bucket for *forecast_time*.

String template:

GFS_FORECAST_BUCKET/gfs.YYYYMMDD/HH/atmos/{filename}",

Returns
str – Bucket prefix for forecast files.

forecast_time

Base time of the previous GFS forecast based on input times

grid

Lat / Lon grid spacing. One of [0.25, 0.5, 1]

property hash

Generate a unique hash for this datasource.

Returns
`str` – Unique hash for met instance (sha1)

open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (`xr.Dataset | None`, *optional*) – Input `xr.Dataset` loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (`dict[str, Any] | None`, *optional*) – Dictionary of keyword arguments passed into `xarray.open_mfdataset()` when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if `dataset` is input.
- ****kwargs** (`Any`) – Keyword arguments passed through directly into `MetDataset` constructor.

Returns
MetDataset – Meteorology dataset

See also:

`xarray.open_mfdataset()`

property pressure_level_variables

GFS pressure level parameters.

Returns
`list[MetVariable] | None` – List of MetVariable available in datasource

set_metadata(*ds*)

Set met source metadata on `ds.attrs`.

This is called within the *open_metdataset()* method to set metadata on the returned `MetDataset` instance.

Parameters
ds (`xr.Dataset | MetDataset`) – Dataset to set metadata on. Mutated in place.

show_progress

Show progress bar when downloading files from AWS

property single_level_variables

GFS surface level parameters.

Returns
`list[MetVariable] | None` – List of MetVariable available in datasource

property supported_pressure_levels

Get pressure levels available.

Returns

`list[int]` – List of integer pressure level values

pycontrails.datalib.gfs.variables

GFS Parameter Support.

Parameter definitions:

- Reanalysis
- Surface
- Pressure Levels

11.2.3 ARCO ERA5

datalib.ecmwf.arco_era5	Support for ARCO ERA5.

pycontrails.datalib.ecmwf.arco_era5

Support for ARCO ERA5.

This module supports:

- Downloading ARCO ERA5 model level data for specific times and pressure level variables.
- Downloading ARCO ERA5 single level data for specific times and single level variables.
- Interpolating model level data to a target lat-lon grid and pressure levels.
- Local caching of the downloaded and interpolated data as netCDF files.
- Opening cached data as a *pycontrails.MetDataset* object.

This module requires the following additional dependencies:

- metview (binaries and python bindings)
- gcsfs
- zarr

Functions

open_arco_era5_model_level_data(t, ...)	Open ARCO ERA5 model level data for a specific time and variables.
open_arco_era5_single_level(t, variables)	Open ARCO ERA5 single level data for a specific date and variables.

Classes

ARCOERA5(time, variables[, pressure_levels, ...])	ARCO ERA5 data accessed remotely through Google Cloud Storage.

class pycontrails.datalib.ecmwf.arco_era5.**ARCOERA5**(*time*, *variables*, *pressure_levels=None*, *grid=0.25*, *cachestore=<object object>*, *n_jobs=1*, *cleanup_metview_tempfiles=True*)

Bases: `ECMWFAPI`

ARCO ERA5 data accessed remotely through Google Cloud Storage.

This is a high-level interface to access and cache ARCO ERA5 for a predefined set of times, variables, and pressure levels.

New in version 0.50.0.

Parameters

- **time** (`TimeInput`) – Time of the data to open.
- **variables** (`VariableInput`) – List of variables to open.
- **pressure_levels** (`PressureLevelInput`, *optional*) – Target pressure levels, $[hPa]$. For pressure level data, this should be a sorted (increasing or decreasing) list of integers. For single level data, this should be `-1`. By default, the pressure levels are set to the pressure levels at each model level between 20,000 and 50,000 ft assuming a constant surface pressure.
- **grid** (`float`, *optional*) – Target grid resolution, [deg]. Default is 0.25.
- **cachestore** (`CacheStore`, *optional*) – Cache store to use. By default, a new disk cache store is used. If None, no caching is done.
- **n_jobs** (`int`, *optional*) – EXPERIMENTAL: Number of parallel jobs to use for downloading data. By default, 1.
- **cleanup_metview_tempfiles** (`bool`, *optional*) – If True, cleanup all `TEMP_DIRECTORY/tmp*.grib` files. Implementation is brittle and may not work on all systems. By default, True.

References

[Carver and Merose, 2023]

See also:

open_arco_era5_model_level_data(), *open_arco_era5_single_level()*

create_cachepath(*t*)

Return cachepath to local data file based on datetime.

Parameters
t (datetime) – Datetime of datafile

Returns
str – Path to cached data file

download_dataset(*times*)

Download data from data source for input times.

Parameters
times (list[datetime]) – List of datetimes to download a store in cache

grid

Lat / Lon grid spacing

open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (xr.Dataset | None, *optional*) – Input xr.Dataset loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (dict[str, Any] | None, *optional*) – Dictionary of keyword arguments passed into xarray.open_mfdataset() when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if dataset is input.
- ****kwargs** (Any) – Keyword arguments passed through directly into MetDataset constructor.

Returns
MetDataset – Meteorology dataset

See also:

xarray.open_mfdataset()

paths

Path to local source files to load. Set to the paths of files cached in cachestore if no paths input is provided on init.

property pressure_level_variables

Variables available in the ARCO ERA5 model level data.

Returns
list[MetVariable] | None – List of MetVariable available in datasource

pressure_levels

List of pressure levels. Set to [-1] for data without level coordinate. Use `parse_pressure_levels()` to handle `PressureLevelInput`.

set_metadata(*ds*)

Set met source metadata on `ds.attrs`.

This is called within the *open_metdataset()* method to set metadata on the returned `MetDataset` instance.

Parameters
ds (`xr.Dataset | MetDataset`) – Dataset to set metadata on. Mutated in place.

property single_level_variables

Variables available in the ARCO ERA5 single level data.

Returns
`list[MetVariable] | None` – List of MetVariable available in datasource

timesteps

List of individual timesteps from data source derived from `time` Use `parse_time()` to handle `TimeInput`.

variables

Variables requested from data source Use `parse_variables()` to handle `VariableInput`.

pycontrails.datalib.ecmwf.arco_era5.**open_arco_era5_model_level_data**(*t*, *variables*, *pressure_levels*, *grid*)

Open ARCO ERA5 model level data for a specific time and variables.

This function downloads moisture, wind, and surface data from the ARCO ERA5 Zarr stores and interpolates the data to a target grid and pressure levels.

This function requires the metview package to be installed. It is not available as an optional pycontrails dependency, and instead must be installed manually.

Parameters

- **t** (`datetime.datetime`) – Time of the data to open.
- **variables** (`list[met_var.MetVariable]`) – List of variables to open. Unsupported variables are ignored.
- **pressure_levels** (`list[int]`) – Target pressure levels, $[hPa]$. For `metview` compatibility, this should be a sorted (increasing or decreasing) list of integers. Floating point values are treated as integers in `metview`.
- **grid** (`float`) – Target grid resolution, $[\text{deg}]$. A value of 0.25 is recommended.

Returns
`xarray.Dataset` – Dataset with the requested variables on the target grid and pressure levels. Data is reformatted for `MetDataset` conventions. Data **is not** cached.

References

- [Carver and Merose, 2023]
- ARCO ERA5 moisture workflow
- Model Level Walkthrough
- Surface Reanalysis Walkthrough

pycontrails.datalib.ecmwf.arco_era5.**open_arco_era5_single_level**(*t*, *variables*)

Open ARCO ERA5 single level data for a specific date and variables.

Parameters

- **t** (`datetime.date`) – Date of the data to open.
- **variables** (`list[met_var.MetVariable]`) – List of variables to open.

Returns

`xarray.Dataset` – Dataset with the requested variables. Data is reformatted for `MetDataset` conventions. Data **is not** cached.

Raises

`FileNotFoundError` – If the variable is not found at the requested date. This could indicate that the variable is not available in the ARCO ERA5 dataset, or that the time requested is outside the available range.

11.2.4 GOES

datalib.goes	Support for GOES access and analysis.

pycontrails.datalib.goes

Support for GOES access and analysis.

Resources

- GOES 16/18 on GCP notes
- GOES on AWS notes
- Scan Mode information and timing
- Current position of the MESO1 sector
- Current position of the MESO2 sector
- Historical Mesoscale regions
- Real time GOES data quality

Module Attributes

DEFAULT_CHANNELS	Default channels to use if none are specified.
GOES_SCAN_MODE_CHANGE	The time at which the GOES scan mode changed from mode 3 to mode 6.

Functions

extract_goes_visualization(da[, ...])	Extract artifacts for visualizing GOES data with the given color scheme.
gcs_goes_path(time, region[, channels, ...])	Return GCS paths to GOES data at the given time for the given region and channels.
to_ash(da[, convention])	Compute 3d RGB array for the ASH color scheme.
to_true_color(da[, gamma])	Compute 3d RGB array for the true color scheme.

Classes

GOES([region, channels, cachestore, goes_bucket])	Support for GOES-16 data handling.
GOESRegion(value[, names, module, qualname, ...])	GOES Region of interest.

pycontrails.datalib.goes.**DEFAULT_CHANNELS = ('C11', 'C14', 'C15')**

Default channels to use if none are specified. These are the channels required by the SEVIRI (MIT) ash color scheme.

class pycontrails.datalib.goes.**GOES**(*region=GOESRegion.F*, *channels=None*, *cachestore=<object object>*, *goes_bucket='gcp-public-data-goes-16'*)

Bases: `object`

Support for GOES-16 data handling.

Parameters

- **region** (GOESRegion | str = `{"F", "C", "M1", "M2"}`) – GOES Region of interest. Uses the following conventions.
  - F: Full Disk
  - C: CONUS
  - M1: Mesoscale 1
  - M2: Mesoscale 2
- **channels** (`str | set[str] | None`) – Set of channels or bands for CMIP data. The 16 possible channels are represented by the strings "C01" to "C16". For the SEVIRI ash color scheme, set `channels=("C11", "C14", "C15")`. For the true color scheme, set `channels=("C01", "C02", "C03")`. By default, the channels required by the SEVIRI ash color scheme are used. The channels must have a common horizontal resolution. The resolutions are:
  - C01: 1.0 km
  - C02: 0.5 km (treated as 1.0 km)

  - C03: 1.0 km
  - C04: 2.0 km
  - C05: 1.0 km
  - C06 - C16: 2.0 km
- **cachestore** (`cache.CacheStore | None`) – Cache store for GOES data. If None, data is downloaded directly into memory. By default, a `cache.DiskCacheStore` is used.
- **goes_bucket** (str = `"gcp-public-data-goes-16"`) – GCP bucket for GOES data. AWS access is not supported.

See also:

GOESRegion, *gcs_goes_path*

Examples

```
>>> goes = GOES(region="M1", channels=("C11", "C14"))
>>> da = goes.get("2021-04-03 02:10:00")
>>> da.shape
(2, 500, 500)
```

```
>>> da.dims
('band_id', 'y', 'x')
```

```
>>> da.band_id.values
array([11, 14], dtype=int32)
```

```
>>> # Print out a sample of the data
>>> da.sel(band_id=11).isel(x=slice(0, 50, 10), y=slice(0, 50, 10)).values
array([[266.8644 , 265.50812, 271.5592 , 271.45486, 272.75897],
       [250.53697, 273.28064, 273.80225, 270.77673, 274.8977 ],
       [272.8633 , 272.65466, 271.5592 , 274.01093, 273.12415],
       [274.16742, 274.11523, 276.5148 , 273.85443, 270.51593],
       [274.84555, 275.15854, 272.60248, 270.67242, 272.23734]],
      dtype=float32)
```

```
>>> # The data has been cached locally
>>> assert goes.cachestore.listdir()
```

```
>>> # Download GOES data directly into memory by setting cachestore=None
>>> goes = GOES(region="M2", channels=("C11", "C12", "C13"), cachestore=None)
>>> da = goes.get("2021-04-03 02:10:00")
```

```
>>> da.shape
(3, 500, 500)
```

```
>>> da.dims
('band_id', 'y', 'x')
```

```
>>> da.band_id.values
array([11, 12, 13], dtype=int32)
```

```
>>> da.attrs["long_name"]
'ABI L2+ Cloud and Moisture Imagery brightness temperature'
```

```
>>> da.sel(band_id=11).values
array([[251.31944, 249.59802, 249.65018, ..., 270.30725, 270.51593,
        269.83777],
       [250.53697, 249.0242 , 249.12854, ..., 270.15076, 270.30725,
        269.73346],
       [249.1807 , 249.33719, 251.99757, ..., 270.15076, 270.20294,
        268.7945 ],
       ...,
       [277.24512, 277.29727, 277.45377, ..., 274.42822, 274.11523,
        273.7501 ],
       [277.24512, 277.45377, 278.18408, ..., 274.6369 , 274.01093,
        274.06308],
       [276.8278 , 277.14078, 277.7146 , ..., 274.6369 , 273.9066 ,
        274.16742]], dtype=float32)
```

gcs_goes_path(*time*, *channels=None*)

Return GCS paths to GOES data at given time.

Presently only supported for GOES data whose scan time minute coincides with the minute of the time parameter.

Parameters

- **time** (datetime.datetime) – Time of GOES data.
- **channels** (set[str] | None) – Set of channels or bands for CMIP data. If None, the channels attribute is used.

Returns

list[str] – List of GCS paths to GOES data.

get(*time*)

Return GOES data at given time.

Parameters

time (datetime.datetime | str) – Time of GOES data. This should be a timezone-naive datetime object or an ISO 8601 formatted string.

Returns

xarray.DataArray – DataArray of GOES data with coordinates:

- band_id: Channel or band ID
- x: GOES x-coordinate
- y: GOES y-coordinate

class pycontrails.datalib.goes.**GOESRegion**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

GOES Region of interest.

Uses the following conventions.

- F: Full Disk
- C: CONUS
- M1: Mesoscale 1
- M2: Mesoscale 2

C = 2

F = 1

M1 = 3

M2 = 4

pycontrails.datalib.goes.**GOES_SCAN_MODE_CHANGE = datetime.datetime(2019, 4, 2, 16, 0)**

The time at which the GOES scan mode changed from mode 3 to mode 6. This is used to determine the scan time resolution. See GOES ABI scan information.

pycontrails.datalib.goes.**extract_goes_visualization**(*da*, *color_scheme='ash'*, *ash_convention='SEVIRI'*, *gamma=2.2*)

Extract artifacts for visualizing GOES data with the given color scheme.

Parameters

- **da** (xarray.DataArray) – DataArray of GOES data as returned by *GOES.get()*. Must have the channels required by *to_ash()*.
- **color_scheme** (str = {"ash", "true"}) – Color scheme to use for visualization.
- **ash_convention** (str = {"SEVIRI", "standard"}) – Passed into *to_ash()*. Only used if color_scheme="ash".
- **gamma** (float = 2.2) – Passed into *to_true_color()*. Only used if color_scheme="true".

Returns

- **rgb** (npt.NDArray[np.float32]) – 3D RGB array of shape (height, width, 3). Any nan values are replaced with 0.
- **src_crs** (ccrs.Geostationary) – The Geostationary projection built from the GOES metadata.
- **src_extent** (tuple[float, float, float, float]) – Extent of GOES data in the Geostationary projection

pycontrails.datalib.goes.**gcs_goes_path**(*time*, *region*, *channels=None*, *bucket='gcp-public-data-goes-16'*, *fs=None*)

Return GCS paths to GOES data at the given time for the given region and channels.

Presently only supported for GOES data whose scan time minute coincides with the minute of the time parameter.

Parameters

- **time** (datetime.datetime) – Time of GOES data. This should be a timezone-naive datetime object or an ISO 8601 formatted string.
- **region** (*GOESRegion*) – GOES Region of interest.

- **channels** (`str | Iterable[str]`) – Set of channels or bands for CMIP data. The 16 possible channels are represented by the strings "C01" to "C16". For the SEVIRI ash color scheme, set `channels=("C11", "C14", "C15")`. For the true color scheme, set `channels=("C01", "C02", "C03")`. By default, the channels required by the SEVIRI ash color scheme are used.

Returns

`list[str]` – List of GCS paths to GOES data.

Examples

```
>>> from pprint import pprint
>>> t = datetime.datetime(2023, 4, 3, 2, 10)
```

```
>>> paths = gcs_goes_path(t, GOESRegion.F, channels=("C11", "C12", "C13"))
>>> pprint(paths)
['gcp-public-data-goes-16/ABI-L2-CMIPF/2023/093/02/OR_ABI-L2-CMIPF-M6C11_G16_
↪s20230930210203_e20230930219511_c20230930219586.nc',
 'gcp-public-data-goes-16/ABI-L2-CMIPF/2023/093/02/OR_ABI-L2-CMIPF-M6C12_G16_
↪s20230930210203_e20230930219516_c20230930219596.nc',
 'gcp-public-data-goes-16/ABI-L2-CMIPF/2023/093/02/OR_ABI-L2-CMIPF-M6C13_G16_
↪s20230930210203_e20230930219523_c20230930219586.nc']
```

```
>>> paths = gcs_goes_path(t, GOESRegion.C, channels=("C11", "C12", "C13"))
>>> pprint(paths)
['gcp-public-data-goes-16/ABI-L2-CMIPC/2023/093/02/OR_ABI-L2-CMIPC-M6C11_G16_
↪s20230930211170_e20230930213543_c20230930214055.nc',
 'gcp-public-data-goes-16/ABI-L2-CMIPC/2023/093/02/OR_ABI-L2-CMIPC-M6C12_G16_
↪s20230930211170_e20230930213551_c20230930214045.nc',
 'gcp-public-data-goes-16/ABI-L2-CMIPC/2023/093/02/OR_ABI-L2-CMIPC-M6C13_G16_
↪s20230930211170_e20230930213557_c20230930214065.nc']
```

```
>>> t = datetime.datetime(2023, 4, 3, 2, 11)
>>> paths = gcs_goes_path(t, GOESRegion.M1, channels="C01")
>>> pprint(paths)
['gcp-public-data-goes-16/ABI-L2-CMIPM/2023/093/02/OR_ABI-L2-CMIPM1-M6C01_G16_
↪s20230930211249_e20230930211309_c20230930211386.nc']
```

`pycontrails.datalib.goes.`**`to_ash`**(*da*, *convention='SEVIRI'*)

Compute 3d RGB array for the ASH color scheme.

Parameters

- **da** (`xarray.DataArray`) – DataArray of GOES data with appropriate channels.
- **convention** (str = `{"SEVIRI", "standard"}`) – Convention for color space.
  - SEVIRI convention requires channels C11, C14, C15. Used in [Kulik, 2019].
  - Standard convention requires channels C11, C13, C14, C15

Returns

`npt.NDArray[np.float32]` – 3d RGB array with ASH color scheme according to convention.

References

- Ash RGB quick guide (the color space and color interpretations)
- [*SEVIRI RGB Cal Module - Part II*, n.d.]
- [Kulik, 2019]

Examples

```
>>> goes = GOES(region="M2", channels=("C11", "C14", "C15"))
>>> da = goes.get("2022-10-03 04:34:00")
>>> rgb = to_ash(da)
>>> rgb.shape
(500, 500, 3)
```

```
>>> rgb[0, 0, :]
array([0.0127004 , 0.22793579, 0.3930847 ], dtype=float32)
```

pycontrails.datalib.goes.**to_true_color**(*da*, *gamma=2.2*)

Compute 3d RGB array for the true color scheme.

Parameters

- **da** (xarray.DataArray) – DataArray of GOES data with channels C01, C02, C03.
- **gamma** (float = 2.2) – Gamma correction for the RGB channels.

Returns

npt.NDArray[np.float32] – 3d RGB array with true color scheme.

References

- Unidata's true color recipe

11.3 Models

11.3.1 Base Classes

Model([met, params])	Base class for physical models.
ModelParams([copy_source, ...])	Class for constructing model parameters.

pycontrails.Model

class pycontrails.**Model**(*met=None*, *params=None*, ***params_kwargs*)

Bases: ABC

Base class for physical models.

Implementing classes must implement the *eval()* method

__init__(*met=None*, *params=None*, ***params_kwargs*)

Methods

__init__([met, params])	
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
eval([source])	Abstract method to handle evaluation.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is `type_`.
set_source([source])	Attach original or copy of input `source` to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to `source`.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Data evaluated in model
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	Get long name descriptor, annotated on `xr.DataArray` outputs.
met_required	Require meteorology is not None on __init__()
name	class`Flight`.
met_variables	Required meteorology pressure level variables.
processed_met_variables	Set of required parameters if processing already complete on `met` input.
optional_met_variables	Optional meteorology variables

default_params

alias of *ModelParams*

downselect_met()

Downselect *met* domain to the max/min bounds of *source*.

Override this method if special handling is needed in met down-selection.

- *source* must be defined before calling *downselect_met()*.
- This method copies and re-assigns *met* using `met.copy()` to avoid side-effects.

Raises

- **ValueError** – Raised if *source* is not defined. Raised if *source* is not a *GeoVectorDataset*.
- **TypeError** – Raised if *met* is not a *MetDataset*.

abstract eval(*source=None*, ***params*)

Abstract method to handle evaluation.

Implementing classes should override call signature to overload `source` inputs and model outputs.

Parameters

- **source** (`ModelInput`, *optional*) – Dataset defining coordinates to evaluate model. Defined by implementing class, but must be a subset of ModelInput. If None, *met* is assumed to be evaluation points.
- ****params** (`Any`) – Overwrite model parameters before evaluation.

Returns

`ModelOutput` – Return type depends on implementing model

get_source_param(*key*, *default=<object object>*, ***, *set_attr=True*)

Get source data with default set by parameter key.

Retrieves data with the following hierarchy:

1. `source.data[key]`. Returns `np.ndarray | xr.DataArray`.
2. `source.attrs[key]`
3. `params[key]`
4. `default`

In case 3., the value of `params[key]` is attached to `source.attrs[key]`.

Parameters

- **key** (`str`) – Key to retrieve
- **default** (`Any`, *optional*) – Default value if key is not found.
- **set_attr** (`bool`, *optional*) – If True (default), set `source.attrs[key]` to `params[key]` if found. This allows for better post model evaluation tracking.

Returns

`Any` – Value(s) found for key in source data, source attrs, or model params

Raises

KeyError – Raises KeyError if key is not found in any location and `default` is not provided.

See also:

-

property hash

Generate a unique hash for model instance.

Returns

`str` – Unique hash for model instance (sha1)

property interp_kwargs

Shortcut to create interpolation arguments from *params*.

Returns

dict[str, Any] – Dictionary with keys

- "method"
- "bounds_error"
- "fill_value"
- "localize"
- "use_indices"
- "q_method"

as determined by *params*.

abstract property long_name

Get long name descriptor, annotated on xr.DataArray outputs.

met

Meteorology data

met_required = False

Require meteorology is not None on __init__()

met_variables

Required meteorology pressure level variables. Each element in the list is a MetVariable or a tuple[MetVariable]. If element is a tuple[MetVariable], the variable depends on the data source. Only one variable in the tuple is required.

abstract property name

class`Flight`.

Type
Get model name for use as a data key in xr.DataArray or

optional_met_variables

Optional meteorology variables

params

Instantiated model parameters, in dictionary form

processed_met_variables

Set of required parameters if processing already complete on met input.

require_met()

Ensure that *met* is a MetDataset.

Returns
MetDataset – Returns reference to *met*. This is helpful for type narrowing *met* when meteorology is required.

Raises
ValueError – Raises when *met* is None.

require_source_type(*type_*)

Ensure that *source* is `type_`.

Returns

`_Source` – Returns reference to *source*. This is helpful for type narrowing *source* to specific type(s).

Raises

ValueError – Raises when *source* is not `_type_`.

set_source(*source=None*)

Attach original or copy of input `source` to *source*.

Parameters

source (`MetDataset | GeoVectorDataset | Flight | Iterable[Flight] | None`) – Parameter `source` passed in *eval()*. If None, an empty MetDataset with coordinates like *met* is set to *source*.

See also:

-

meth:*eval*

set_source_met(*optional=False*, *variable=None*)

Ensure or interpolate each required *met_variables* on *source* .

For each variable in *met_variables*, check *source* for data variable with the same name.

For *GeoVectorDataset* sources, try to interpolate *met* if variable does not exist.

For *MetDataset* sources, try to get data from *met* if variable does not exist.

Parameters

- **optional** (`bool`, *optional*) – Include *optional_met_variables*
- **variable** (`MetVariable | Sequence[MetVariable] | None`, *optional*) – MetVariable to set, from *met_variables*. If None, set all variables in *met_variables* and *optional_met_variables* if `optional` is True.

Raises

- **ValueError** – Variable does not exist and *source* is a MetDataset.
- **KeyError** – Variable not found in *source* or *met*.

source

Data evaluated in model

transfer_met_source_attrs(*source=None*)

Transfer met source metadata from *met* to `source`.

update_params(*params=None*, ***params_kwargs*)

Update model parameters on *params*.

Parameters

- **params** (`dict[str, Any]`, *optional*) – Model parameters to update, as dictionary. Defaults to {}
- ****params_kwargs** (`Any`) – Override keys in `params` with keyword arguments.

pycontrails.ModelParams

class pycontrails.**ModelParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*)

Bases: `object`

Class for constructing model parameters.

Implementing classes must still use the `@dataclass` operator.

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

copy_source	Copy input `source` data on eval
downselect_met	Downselect input `MetDataset`` to region around `source`.
interpolation_bounds_error	If True, points lying outside interpolation will raise an error
interpolation_fill_value	Used for outside interpolation value if *interpolation_bounds_error* is False
interpolation_localize	Experimental.
interpolation_method	Interpolation method.
interpolation_q_method	Experimental.
interpolation_use_indices	Experimental.
met_latitude_buffer	Met latitude buffer for input to `Flight.downselect_met()`, in WGS84 coordinates.
met_level_buffer	Met level buffer for input to `Flight.downselect_met()`, in $[hPa]$.
met_longitude_buffer	Met longitude buffer for input to `Flight.downselect_met()`, in WGS84 coordinates.
met_time_buffer	Met time buffer for input to `Flight.downselect_met()` Only applies when *downselect_met* is True.
verify_met	Call `_verify_met()` on model instantiation.

as_dict()

Convert object to dictionary.

We use this method instead of *dataclasses.asdict* to use a shallow/unrecursive copy. This will return values as Any instead of dict.

Returns
`dict[str, Any]` – Dictionary version of self.

copy_source = True

Copy input `source` data on eval

downselect_met = True

Downselect input `MetDataset` to region around `source`.

interpolation_bounds_error = False

If True, points lying outside interpolation will raise an error

interpolation_fill_value = nan

Used for outside interpolation value if *interpolation_bounds_error* is False

interpolation_localize = False

Experimental. See *pycontrails.core.interpolation*.

interpolation_method = 'linear'

Interpolation method. Supported methods include "linear", "nearest", "slinear", "cubic", and "quintic". See `scipy.interpolate.RegularGridInterpolator` for the description of each method. Not all methods are supported by all met grids. For example, the "cubic" method requires at least 4 points per dimension.

interpolation_q_method = None

Experimental. Alternative interpolation method to account for specific humidity lapse rate bias. Must be one of `None`, `"cubic-spline"`, or `"log-q-log-p"`. If `None`, no special interpolation is used for specific humidity. The `"cubic-spline"` method applies a custom stretching of the met interpolation table to account for the specific humidity lapse rate bias. The `"log-q-log-p"` method interpolates in the log of specific humidity and pressure, then converts back to specific humidity. Only used by models calling to `interpolate_met()`.

interpolation_use_indices = False

Experimental. See *pycontrails.core.interpolation*.

met_latitude_buffer = (0.0, 0.0)

Met latitude buffer for input to `Flight.downselect_met()`, in WGS84 coordinates. Only applies when *downselect_met* is True.

met_level_buffer = (0.0, 0.0)

Met level buffer for input to `Flight.downselect_met()`, in $[hPa]$. Only applies when *downselect_met* is True.

met_longitude_buffer = (0.0, 0.0)

Met longitude buffer for input to `Flight.downselect_met()`, in WGS84 coordinates. Only applies when *downselect_met* is True.

met_time_buffer = (numpy.timedelta64(0,'h'), numpy.timedelta64(0,'h'))

Met time buffer for input to `Flight.downselect_met()` Only applies when *downselect_met* is True.

verify_met = True

Call `_verify_met()` on model instantiation.

11.3.2 SAC, ISSR & PCC

models.issr	Ice super-saturated regions (ISSR).
models.sac	Schmidt-Appleman criteria (SAC).
models.pcr	Persistent contrail regions (PCR = SAC & ISSR).
models.pcc	Probability of persistent contrail coverage (PCC).

pycontrails.models.issr

Ice super-saturated regions (ISSR).

Functions

issr(air_temperature[, specific_humidity, ...])	Calculate ice super-saturated regions.

Classes

ISSR([met, params])	Ice super-saturated regions over a `Flight` trajectory or `MetDataset` grid.
ISSRParams([copy_source, ...])	Default ISSR model parameters.

class pycontrails.models.issr.**ISSR**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Ice super-saturated regions over a `Flight` trajectory or `MetDataset` grid.

This model calculates points where the relative humidity over ice is greater than 1.

Parameters

met (*MetDataset*) – Dataset containing "air_temperature" and "specific_humidity" variables

Examples

```
>>> from datetime import datetime
>>> from pycontrails.datalib.ecmwf import ERA5
>>> from pycontrails.models.issr import ISSR
>>> from pycontrails.models.humidity_scaling import ConstantHumidityScaling
```

```
>>> # Get met data
>>> time = datetime(2022, 3, 1, 0), datetime(2022, 3, 1, 2)
>>> variables = ["air_temperature", "specific_humidity"]
>>> pressure_levels = [200, 250, 300]
>>> era5 = ERA5(time, variables, pressure_levels)
>>> met = era5.open_metdataset()
```

```
>>> # Instantiate and run model
>>> scaling = ConstantHumidityScaling(rhi_adj=0.98)
>>> model = ISSR(met, humidity_scaling=scaling)
>>> out1 = model.eval()
>>> issr1 = out1["issr"]
>>> issr1.proportion  # Get proportion of values with ice supersaturation
0.11414134603859523
```

```
>>> # Run with a lower threshold
>>> out2 = model.eval(rhi_threshold=0.95)
>>> issr2 = out2["issr"]
>>> issr2.proportion
0.146647
```

default_params

alias of *ISSRParams*

eval(*source=None*, ***params*)

Evaluate ice super-saturated regions along flight trajectory or on meteorology grid.

Changed in version 0.27.0: Humidity scaling now handled automatically. This is controlled by model parameter `humidity_scaling`.

Changed in version 0.48.0: If the `source` is a `MetDataset`, the returned object will also be a `MetDataset`. Previous the "issr" `MetDataArray` was returned.

Parameters

- **source** (`GeoVectorDataset | Flight | MetDataset | None`, *optional*) – Input GeoVectorDataset or Flight. If None, evaluates at the *met* grid points.
- ****params** (`Any`) – Overwrite model parameters before eval

Returns

`GeoVectorDataset | Flight | MetDataset` – Returns 1 in ISSR, 0 everywhere else. Returns *np.nan* if interpolating outside meteorology grid.

Raises

NotImplementedError – Raises if input `source` is not supported.

long_name = 'Ice super-saturated regions'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'))**

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

name = 'issr'

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

class pycontrails.models.issr.**ISSRParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *rhi_threshold=1.0*, *humidity_scaling=None*)

Bases: *ModelParams*

Default ISSR model parameters.

humidity_scaling = None

Humidity scaling

rhi_threshold = 1.0

RHI Threshold

pycontrails.models.issr.**issr**(*air_temperature*, *specific_humidity=None*, *air_pressure=None*, *rhi=None*, *rhi_threshold=1.0*)

Calculate ice super-saturated regions.

Regions where the atmospheric relative humidity over ice is greater than 1.

Parameters `air_temperature`, `specific_humidity`, `air_pressure`, and `rhi` must have compatible shapes when defined.

Either `specific_humidity` and `air_pressure` must both be provided, or `rhi` must be provided.

Parameters

- **air_temperature** (*ArrayLike*) – A sequence or array of temperature values, $[K]$.
- **specific_humidity** (`ArrayLike | None`) – A sequence or array of specific humidity values, $[kg_{H_2O}\ kg_{moistair}]$ None by default.
- **air_pressure** (`ArrayLike | None`) – A sequence or array of atmospheric pressure values, $[Pa]$. None by default.
- **rhi** (`ArrayLike | None`, *optional*) – A sequence of array of RHi values, if already known. If not provided, this function will compute RHi from *air_temperature*, *specific_humidity*, and *air_pressure*. None by default.
- **rhi_threshold** (`float`, *optional*) – Relative humidity over ice threshold for determining ISSR state

Returns

ArrayLike – ISSR state of each point indexed by the parameters.

pycontrails.models.sac

Schmidt-Appleman criteria (SAC).

Functions

T_critical_sac(T_LM, relative_humidity, G[, ...])	Estimate temperature threshold for persistent contrail formation.
T_sat_liquid(G)	Calculate temperature at which liquid saturation curve has slope G.
T_sat_liquid_high_accuracy(G[, maxiter])	Calculate temperature at which liquid saturation curve has slope G.
rh_critical_sac(air_temperature, T_sat_liquid, G)	Calculate critical relative humidity threshold of contrail formation.
sac(rh, rh_crit_sac)	Points at which the Schmidt-Appleman Criteria is satisfied.
slope_mixing_line(specific_humidity, ...)	Calculate the slope of the mixing line in a temperature-humidity diagram.

Classes

SAC([met, params])	Determine points where Schmidt-Appleman Criteria is satisfied.
SACParams([copy_source, ...])	Parameters for *SAC*.

class pycontrails.models.sac.**SAC**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Determine points where Schmidt-Appleman Criteria is satisfied.

Parameters

- **met** (*MetDataset*) – Dataset containing "air_temperature", "specific_humidity" variables.
- **params** (dict[str, Any], *optional*) – Override *SACParams* with dictionary.
- ****params_kwargs** – Override *SACParams* with keyword arguments.

default_params

alias of *SACParams*

eval(*source=None*, ***params*)

Evaluate the Schmidt-Appleman criteria along flight trajectory or on meteorology grid.

Changed in version 0.27.0: Humidity scaling now handled automatically. This is controlled by model parameter humidity_scaling.

Changed in version 0.48.0: If the source is a MetDataset, the returned object will also be a MetDataset. Previous the "sac" MetDataArray was returned.

Parameters

- **source** (GeoVectorDataset | Flight | MetDataset | None, *optional*) – Input GeoVectorDataset or Flight. If None, evaluates at the *met* grid points.

- ****params** (Any) – Overwrite model parameters before eval

Returns

GeoVectorDataset | Flight | MetDataset – Returns 1 where SAC is satisfied, 0 everywhere else. Returns np.nan if interpolating outside meteorology grid.

long_name = 'Schmidt-Appleman contrail formation criteria'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'))**

Required meteorology pressure level variables. Each element in the list is a MetVariable or a tuple[MetVariable]. If element is a tuple[MetVariable], the variable depends on the data source. Only one variable in the tuple is required.

name = 'sac'

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

class pycontrails.models.sac.**SACParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *engine_efficiency=0.3*, *fuel=<factory>*, *humidity_scaling=None*)

Bases: *ModelParams*

Parameters for *SAC*.

engine_efficiency = 0.3

Jet engine efficiency, $[0 - 1]$

fuel

Fuel type. Overridden by Fuel provided on input source attributes

humidity_scaling = None

Humidity scaling

pycontrails.models.sac.**T_critical_sac**(*T_LM*, *relative_humidity*, *G*, *maxiter=10*)

Estimate temperature threshold for persistent contrail formation.

This quantity is defined as T_LC in Schumann (see reference below). Equation (11) of this paper implicitly defines T_LC as the solution to the equation

```
T_LC = T_LM - (e_L(T_LM) - rh * e_L(T_LC)) / G
```

For relative humidity above 0.999, the corresponding entry from `T_LM` is returned (page 10, top of the right-hand column). Otherwise, the solution to the equation above is approximated via Newton's method.

Parameters

- **T_LM** (*ArrayLike*) – Output of *T_sat_liquid()* calculation.
- **relative_humidity** (*ArrayLike*) – Relative humidity values
- **G** (*ArrayLike*) – Slope of the mixing line in a temperature-humidity diagram.
- **maxiter** (`int`, *optional*) – Passed into `scipy.optimize.newton()`. By default, 10.

Returns

ArrayLike – Critical temperature threshold values.

References

- [Schumann, 1996]

pycontrails.models.sac.**T_sat_liquid**(*G*)

Calculate temperature at which liquid saturation curve has slope G.

Parameters

G (*ArrayLike*) – Slope of the mixing line in a temperature-humidity diagram.

Returns

ArrayLike – Maximum threshold temperature for 100% relative humidity with respect to liquid, $[K]$. This can also be interpreted as the temperature at which the liquid saturation curve has slope G.

References

- [Schumann, 1996]

See also:

T_sat_liquid_high_accuracy()

Notes

Defined (using notation T_LM) in [Schumann, 1996] in the first full paragraph on page 10 as

> for T = T_LC, the mixing line just touches [is tangent to] the saturation curve. See equation (10).

The formula used here is taken from equation (31).

pycontrails.models.sac.**T_sat_liquid_high_accuracy**(*G*, *maxiter=5*)

Calculate temperature at which liquid saturation curve has slope G.

The function *T_sat_liquid()* gives a first order approximation to equation (10) of the Schumann paper referenced below. This function uses Newton's method to compute the numeric solution to (10).

Parameters

- **G** (*ArrayLike*) – Slope of the mixing line

- **maxiter** (`int`, *optional*) – Passed into `scipy.optimize.newton()`. Because `T_sat_liquid` is already fairly accurate, few iterations are needed for Newton's method to converge. By default, 5.

Returns

ArrayLike – Maximum threshold temperature for 100% relative humidity with respect to liquid, $[K]$.

References

- [Schumann, 1996]

See also:

`T_sat_liquid_high()`

`pycontrails.models.sac.`**`rh_critical_sac`**(*air_temperature*, *T_sat_liquid*, *G*)

Calculate critical relative humidity threshold of contrail formation.

Parameters

- **air_temperature** (*ArrayLike*) – A sequence or array of temperature values, $[K]$
- **T_sat_liquid** (*ArrayLike*) – Maximum threshold temperature for 100% relative humidity with respect to liquid, $[K]$
- **G** (*ArrayLike*) – Slope of the mixing line in a temperature-humidity diagram.

Returns

ArrayLike – Critical relative humidity of contrail formation, $[[0 - 1]]$

References

- [Ponater, 2002]

`pycontrails.models.sac.`**`sac`**(*rh*, *rh_crit_sac*)

Points at which the Schmidt-Appleman Criteria is satisfied.

Parameters of type `ArrayLike` must have compatible shapes.

Parameters

- **rh** (*ArrayLike*) – Relative humidity values
- **rh_crit_sac** (*ArrayLike*) – Critical relative humidity threshold of contrail formation

Returns

ArrayLike – SAC state of each point indexed by the `ArrayLike` parameters. Returned array has floating `dtype` with values

- 0.0 signifying SAC fails
- 1.0 signifying SAC holds

NaN entries of parameters propagate into the returned array.

pycontrails.models.sac.**slope_mixing_line**(*specific_humidity*, *air_pressure*, *engine_efficiency*, *ei_h2o*, *q_fuel*)

Calculate the slope of the mixing line in a temperature-humidity diagram.

This quantity is often notated with `G` in the literature.

Parameters

- **specific_humidity** (*ArrayLike*) – A sequence or array of specific humidity values, $[kg_{H_2O}\ kg_{air}]$
- **air_pressure** (*ArrayLike*) – A sequence or array of atmospheric pressure values, $[Pa]$.
- **engine_efficiency** (`float | ArrayLike`) – Engine efficiency, $[0 - 1]$
- **ei_h2o** (`float`) – Emission index of water vapor, $[kg\ kg^{-1}]$
- **q_fuel** (`float`) – Specific combustion heat of fuel combustion, $[J\ kg^{-1}\ K^{-1}]$

Returns

ArrayLike – Slope of the mixing line in a temperature-humidity diagram, $[Pa\ K^{-1}]$

pycontrails.models.pcr

Persistent contrail regions (PCR = SAC & ISSR).

Equivalent to (SAC & ISSR)

Functions

pcr(air_temperature, specific_humidity, ...)	Calculate regions of persistent contrail formation.

Classes

PCR([met, params])	Determine points with likely persistent contrails (PCR).
PCRParams([copy_source, ...])	Persistent Contrail Regions (PCR) parameters.

class pycontrails.models.pcr.**PCR**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Determine points with likely persistent contrails (PCR).

Intersection of Ice Super Saturated Regions (ISSR) with regions in which the Schmidt-Appleman Criteria (SAC) is satisfied.

Parameters

- **met** (*MetDataset*) – Dataset containing "air_temperature", "specific_humidity" variables
- **params** (`dict[str, Any]`, *optional*) – Override PCR model parameters with dictionary. See `PCRGridParams` for model parameters.
- ****params_kwargs** – Override PCR model parameters with keyword arguments. See `PCRGridParams` for model parameters.

default_params

alias of *PCRParams*

eval(*source=None*, ***params*)

Evaluate potential contrails regions of the *met* grid.

Parameters

- **source** (GeoVectorDataset | Flight | MetDataset | None, *optional*) – Input GeoVectorDataset or Flight. If None, evaluates at the *met* grid points.
- ****params** (Any) – Overwrite model parameters.

Returns

GeoVectorDataset | Flight | MetDataset – Returns 1 in potential contrail regions, 0 everywhere else. Returns np.nan if interpolating outside meteorology grid.

long_name = 'Persistent contrail regions'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'))**

Required meteorology pressure level variables. Each element in the list is a MetVariable or a tuple[MetVariable]. If element is a tuple[MetVariable], the variable depends on the data source. Only one variable in the tuple is required.

name = 'pcr'

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

class pycontrails.models.pcr.**PCRParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *rhi_threshold=1.0*, *humidity_scaling=None*, *engine_efficiency=0.3*, *fuel=<factory>*)

Bases: *SACParams*, *ISSRParams*

Persistent Contrail Regions (PCR) parameters.

pycontrails.models.pcr.**pcr**(*air_temperature*, *specific_humidity*, *air_pressure*, *engine_efficiency*, *ei_h2o*, *q_fuel*)

Calculate regions of persistent contrail formation.

Ice Super Saturated Regions (ISSR) where the Schmidt-Appleman Criteria (SAC) is satisfied.

Parameters of type `ArrayLike` must have compatible shapes.

Parameters

- **air_temperature** (*ArrayLike*) – A sequence or array of temperature values, $[K]$
- **specific_humidity** (*ArrayLike*) – A sequence or array of specific humidity values, $[kg_{H_2O}\ kg_{air}^{-1}]$
- **air_pressure** (*ArrayLike*) – A sequence or array of atmospheric pressure values, $[Pa]$.
- **engine_efficiency** (`float | ArrayLike`) – Engine efficiency, $[0 - 1]$
- **ei_h2o** (`float`) – Emission index of water vapor, $[kg\ kg^{-1}]$
- **q_fuel** (`float`) – Specific combustion heat of fuel combustion, $[J\ kg^{-1}\ K^{-1}]$

Returns

- **pcr** (*ArrayLike*) – PCR state of each point indexed by the `ArrayLike` parameters.
- **sac** (*ArrayLike*) – SAC state
- **issr** (*ArrayLike*) – ISSR state

pycontrails.models.pcc

Probability of persistent contrail coverage (PCC).

Classes

PCC(met, surface[, params])	Potential Contrail Coverage Algorithm.
PCCParams([copy_source, ...])	PCC Model Parameters.

class pycontrails.models.pcc.**PCC**(*met*, *surface*, *params=None*, ***params_kwargs*)

Bases: *Model*

Potential Contrail Coverage Algorithm.

Determines the potential of ambient atmosphere to allow contrail formation at grid points.

Parameters

- **met** (*MetDataset*) – Dataset containing *met_variables* variables.
- **surface** (*MetDataset*) – Surface level dataset containing "air_pressure".
- **params** (`dict[str, Any]`, *optional*) – Override PCC model parameters with dictionary. See *PCCParams* for model parameters.
- ****params_kwargs** – Override PCC model parameters with keyword arguments. See *PCCParams* for model parameters.

Notes

Based on Ponater et al. (2002)

Slingo1980(*T*, *p*, *iwc*, *q*, *rh_crit_old*, *rh_crit_new*)

Apply Slingo scheme described in Wood and Field, 1999.

Relationships between Total Water, Condensed Water, and Cloud Fraction in Stratiform Clouds Examined Using Aircraft Data

Parameters

- **T** (`xarray:DataArray`) – Air Temperature, $[K]$
- **p** (`xarray:DataArray`) – Air Pressure, $[Pa]$
- **iwc** (`xarray:DataArray`) – Cloud ice water content, $[kg\ kg^{-1}]$
- **q** (`xarray:DataArray`) – Specific humidity
- **rh_crit_old** (`xarray:DataArray`) – Critical relative humidity, $[[0-1]]$
- **rh_crit_new** (`xarray:DataArray`) – Critical relative humidity, $[[0-1]]$

Returns

`xarray:DataArray` – Probability of cirrus formation, $[[0-1]]$

Smith1990(*T*, *p*, *iwc*, *q*, *rh_crit_old*, *rh_crit_new*)

Apply Smith Scheme described in Rap et al. (2009).

Parameterization of contrails in the UK Met OfficeClimate Model;

Parameters

- **T** (`xarray:DataArray`) – Air Temperature, $[K]$
- **p** (`xarray:DataArray`) – Air Pressure, $[Pa]$
- **iwc** (`xarray:DataArray`) – Cloud ice water content, $[kg\ kg^{-1}]$
- **q** (`xarray:DataArray`) – Specific humidity
- **rh_crit_old** (`xarray:DataArray`) – Critical relative humidity, $[[0-1]]$
- **rh_crit_new** (`xarray:DataArray`) – Critical relative humidity, $[[0-1]]$

Returns

`xarray:DataArray` – Probability of cirrus formation, $[[0-1]]$

Sundqvist1989(*T*, *p*, *iwc*, *q*, *rh_crit_old*, *rh_crit_new*)

Apply Sundqvist scheme described in Ponater et al. (2002).

Contrails in a comprehensive global climate model: Parameterization and radiative forcing results

Parameters

- **T** (`xarray:DataArray`) – Air Temperature, $[K]$
- **p** (`xarray:DataArray`) – Air Pressure, $[Pa]$
- **iwc** (`xarray:DataArray`) – Cloud ice water content, $[kg\ kg^{-1}]$
- **q** (`xarray:DataArray`) – Specific humidity
- **rh_crit_old** (`xarray:DataArray`) – Critical relative humidity, $[[0-1]]$
- **rh_crit_new** (`xarray:DataArray`) – Critical relative humidity, $[[0-1]]$

Returns
xarray:DataArray – Probability of cirrus formation, [[0 − 1]]

b_contr()

Calculate critical relative humidity threshold of contrail formation.

Returns
xarray.DataArray – Critical relative humidity of contrail formation, [[0 − 1]]

Notes

Instead of using a prescribed threshold relative humidity for rh_crit_old the threshold relative humidity now change with pressure.

This equation is described in Roeckner et al. 1996, Eq.57 THE ATMOSPHERIC GENERAL CIRCULATION MODEL ECHAM-4: MODEL DESCRIPTION AND SIMULATION OF PRESENT-DAY CLIMATE

default_params

alias of *PCCParams*

eval(*source=None*, ***params*)

Evaluate PCC model.

Currently only implemented to work on the met data input.

Parameters

- **source** (MetDataset | None, *optional*) –

  Input MetDataset.
  If None, evaluates at the met grid points.

- ****params** (Any) – Overwrite model parameters before eval

Returns
MetDataArray – PCC model output

long_name = 'Potential contrail coverage'

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'), MetVariable(short_name='ciwc', standard_name='specific_cloud_ice_water_content', long_name='Specific cloud ice water content', level_type='isobaricInhPa', ecmwf_id=247, grib1_id=None, grib2_id=(0, 1, 84), units='kg kg**-1', amip=None, description="This parameter is the mass of cloud ice particles per kilogram of the total mass of moist air. The 'total mass of moist air' is the sum of the dry air, water vapour, cloud liquid, cloud ice, rain and falling snow. This parameter represents the average value for a grid box."))**

Required meteorology pressure level variables. Each element in the list is a MetVariable or a tuple[MetVariable]. If element is a tuple[MetVariable], the variable depends on the data source. Only one variable in the tuple is required.

name = 'pcc'

surface

class pycontrails.models.pcc.**PCCParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h')*, *numpy.timedelta64(0, 'h'))*, *cloud_model='Smith1990'*, *rh_crit_factor=0.7*, *fuel=<factory>*, *engine_efficiency=0.35*, *humidity_scaling=None*)

Bases: *ModelParams*

PCC Model Parameters.

cloud_model = 'Smith1990'

Cloud model Options include "Smith1990", "Sundqvist1989", "Slingo1980"

engine_efficiency = 0.35

Engine efficiency

fuel

Fuel type

humidity_scaling = None

Humidity scaling

rh_crit_factor = 0.7

Critical RH Factor for the model to cirrus clouds

11.3.3 CoCiP

models.cocip.Cocip(met, rad[, params])	Contrail Cirrus Prediction Model (CoCiP).
models.cocip.CocipParams([copy_source, ...])	Model parameters required by the CoCiP models.
models.cocip.contrail_properties	Contrail Property Calculations.
models.cocip.radiative_forcing	Module for calculating radiative forcing of contrail cirrus.
models.cocip.wake_vortex	Wave-vortex downwash functions.
models.cocip.wind_shear	Wind shear functions.

pycontrails.models.cocip.Cocip

class pycontrails.models.cocip.**Cocip**(*met*, *rad*, *params=None*, ***params_kwargs*)

Bases: *Model*

Contrail Cirrus Prediction Model (CoCiP).

Published by Ulrich Schumann *et. al.* (DLR Institute of Atmospheric Physics) in [Schumann, 2012], [Schumann *et al.*, 2012].

Parameters

- **met** (*MetDataset*) – Pressure level dataset containing *met_variables* variables. See *Notes* for variable names by data source.
- **rad** (*MetDataset*) – Single level dataset containing top of atmosphere radiation fluxes. See *Notes* for variable names by data source.
- **params** (dict[str, Any], *optional*) – Override Cocip model parameters with dictionary. See CocipFlightParams for model parameters.
- ****params_kwargs** (Any) – Override Cocip model parameters with keyword arguments. See CocipFlightParams for model parameters.

Notes

Inputs

The required meteorology variables depend on the data source (e.g. ECMWF, GFS).

See *met_variables* and *rad_variables* for the list of required variables to the met and rad parameters, respectively. When an item in one of these arrays is a tuple, variable keys depend on data source.

The current list of required variables (labelled by "standard_name"):

Table 3: Variable keys for pressure level data

Parameter	ECMWF	GFS
Air Temperature	air_temperature	air_temperature
Specific Humidity	specific_humidity	specific_humidity
Eastward wind	eastward_wind	eastward_wind
Northward wind	northward_wind	northward_wind
Vertical velocity	lagrangian_tendency_of_air_pressur	lagrangian_tendency_of_air_pressure
Ice water content	specific_cloud_ice_water_content	ice_water_mixing_ratio

Table 4: Variable keys for single-level radiation data

Parameter	ECMWF	GFS
Top solar radiation	top_net_solar_radiation	toa_upward_shortwave_flux
Top thermal radiation	top_net_thermal_radiation	toa_upward_longwave_flux

Modifications

This implementation differs from original CoCiP (Fortran) implementation in a few places:

- This model uses aircraft performance and emissions models to calculate nvPM, fuel flow, and overall propulsion efficiency, if not already provided.
- As described in [Teoh *et al.*, 2022], this implementation sets the initial ice particle activation rate to be a function of the difference between the ambient temperature and the critical SAC threshold temperature. See *pycontrails.models.sac.T_critical_sac()*.
- Isobaric heat capacity calculation. The original model uses a constant value of 1004 $J\ kg^{-1}\ K^{-1}$, whereas this model calculates isobaric heat capacity as a function of specific humidity. See *pycontrails.physics.thermo.c_pm()*.

- Solar direct radiation. The original algorithm uses ECMWF radiation variable *tisr* (top incident solar radiation) as solar direct radiation value. This implementation calculates the theoretical solar direct radiation at any arbitrary point in the atmosphere. See *pycontrails.physics.geo.solar_direct_radiation()*.
- Segment angle. The segment angle calculations for flights and contrail segments have been updated to use more precise spherical geometry instead of a triangular approximation. As the triangle approaches zero, the two calculations agree. See *pycontrails.physics.geo.segment_angle()*.
- Integration. This implementation consistently uses left-Riemann sums in the time integration of contrail segments.
- Segment length ratio. Instead of taking a geometric mean between contrail segments before/after advection, a simple ratio is computed. See *contrail_properties.segment_length_ratio()*.
- Segment energy flux. This implementation does not average spatially contiguous contrail segments when calculating the mean energy flux for the segment of interest. See *contrail_properties.mean_energy_flux_per_m()*.

This implementation is regression tested against results from [Teoh *et al.*, 2022]. See *tests/benchmark/north-atlantic-study/validate.py*.

Outputs

NaN values may appear in model output. Specifically, `np.nan` values are used to indicate:

- Flight waypoint or contrail waypoint is not contained with the `met` domain.
- The variable was NOT computed during the model evaluation. For example, at flight waypoints not producing any persistent contrails, "radiative" variables (`rsr`, `olr`, `rf_sw`, `rf_lw`, `rf_net`) are not computed. Consequently, the corresponding values in the output of *eval()* are NaN. One exception to this rule is found on `ef` (energy forcing) *contrail_age* predictions. For these two "cumulative" variables, waypoints not producing any persistent contrails are assigned 0 values.

References

- [Schumann and Wendling, 1990]
- [Schumann, 2010]
- [Voigt *et al.*, 2010]
- [Schumann *et al.*, 2011]
- [Schumann, 2012]
- [Schumann *et al.*, 2012]
- [Schumann *et al.*, 2012]
- [Schumann *et al.*, 2011]
- [Schumann *et al.*, 2015]
- [Teoh *et al.*, 2020]
- [Schumann *et al.*, 2021]
- [Schumann *et al.*, 2021]
- [Teoh *et al.*, 2022]

See also:

`CocipFlightParams`, `wake_vortex`, `contrail_properties`, `radiative_forcing`, `humidity_scaling`, `Emissions`, `sac`, `tau_cirrus`

__init__(*met*, *rad*, *params=None*, ***params_kwargs*)

Methods

__init__(met, rad[, params])	
downselect_met()	Downselect `met` domain to the max/min bounds of `source`.
eval([source])	Run CoCiP simulation on flight.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that `met` is a MetDataset.
require_source_type(type_)	Ensure that `source` is `type_`.
set_source([source])	Attach original or copy of input `source` to `source`.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on `source` .
transfer_met_source_attrs([source])	Transfer met source metadata from `met` to `source`.
update_params([params])	Update model parameters on `params`.

Attributes

rad	Radiation data formatted as a `MetDataset` at a single pressure level [-1]
contrail	Contrail evolution output from model.
contrail_dataset	`xr.Dataset` representation of contrail evolution.
contrail_list	List of `GeoVectorDataset` contrail objects - one for each timestep
timesteps	Array of `numpy.datetime64` time steps for contrail evolution
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from `params`.
long_name	
met	Met data is not optional
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
optional_met_variables	Additional met variables used to support outputs
params	Instantiated model parameters, in dictionary form
processed_met_variables	Minimal set of met variables needed to run the model after pre-processing.
rad_variables	Required single-level top of atmosphere radiation variables.
source	Last Flight modeled in *eval()*

contrail

Contrail evolution output from model.

Set to None when no contrails are formed. Otherwise, this is a `pandas.DataFrame` describing the evolution of the contrail. Columns include:

- `waypoint`: The index of the waypoint in the original flight creating the contrail. This can be used to join the contrail DataFrame to the `source`.
- `formation_time`: Time of contrail formation. Agrees with the `time` column in `source`.
- `continuous`: Boolean indicating whether the contrail is continuous or not.
- `persistent`: Boolean indicating whether the contrail is persistent or not. A contrail segment is considered continuous if both the current and the next contrail waypoint at the same time step persist.
- `segment_length`: Length of the contrail segment, $[m]$.
- `sin_a`, `cos_a`: Sine and cosine of the segment angle.
- `width`, `depth`: Contrail width and depth, $[m]$.
- `sigma_yz`: The `yz` component of the covariance matrix, $[m^2]$. See *contrail_properties.plume_temporal_evolution()*.
- `q_sat`: Saturation specific humidity over ice, $[kg\ kg^{-1}]$.
- `n_ice_per_m`: Number of ice particles per distance, $[m^{-1}]$.
- `iwc`: Ice water content, $[kg_{ice} kg_{air}^{-1}]$.
- `tau_contrail`: Optical depth of the contrail. See *contrail_properties.contrail_optical_depth()*.
- `rf_sw`, `rf_lw`, `rf_net`: Shortwave, longwave, and net instantaneous radiative forcing, $[W\ m^{-2}]$ at the contrail waypoint.
- `ef`: Energy forcing, $[J]$ at the contrail waypoint. See *contrail_properties.energy_forcing()*.

contrail_dataset

`xr.Dataset` representation of contrail evolution.

contrail_list

List of `GeoVectorDataset` contrail objects - one for each timestep

default_params

alias of `CocipFlightParams`

eval(*source=None*, ***params*)

Run CoCiP simulation on flight.

Simulates the formation and evolution of contrails from a Flight using the contrail cirrus prediction model (CoCiP) from Schumann (2012) [Schumann, 2012].

Changed in version 0.25.11: Previously, any waypoint not surviving the wake vortex downwash phase of CoCiP was assigned a nan-value in the `ef` array within the model output. This is no longer the case. Instead, energy forcing is set to 0.0 for all waypoints which fail to produce persistent contrails. In particular, nan values in the `ef` array are only used to indicate an out-of-met-domain waypoint. The same convention is now used for output variables `contrail_age` and `cocip` as well.

Parameters

- **source** (`Flight | Sequence[Flight] | None`) – Input Flight(s) to model.
- ****params** (`Any`) – Overwrite model parameters before eval.

Returns

`Flight | list[Flight] | NoReturn` – Flight(s) with updated Contrail data. The model parameter "verbose_outputs" determines the variables on the return flight object.

References

- [Schumann, 2012]

long_name = 'Contrail Cirrus Prediction Model'

met_required = True

Require meteorology is not None on __init__()

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'), MetVariable(short_name='u', standard_name='eastward_wind', long_name='Eastward Wind', level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m s**-1', amip='ua', description='"Eastward" indicates a vector component which is positive when directed eastward (negative westward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3), units='m s**-1', amip='va', description='"Northward" indicates a vector component which is positive when directed northward (negative southward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='w', standard_name='lagrangian_tendency_of_air_pressure', long_name='Vertical Velocity (omega)', level_type='isobaricInhPa', ecmwf_id=135, grib1_id=39, grib2_id=(0, 2, 8), units='Pa s**-1', amip='wap', description='The Lagrangian tendency of air pressure, often called "omega", plays the role of the upward component of air velocity when air pressure is being used as the vertical coordinate. If the vertical air velocity is upwards, it is negative when expressed as a tendency of air pressure; downwards is positive. Air pressure is the force per unit area which would be exerted when the moving gas molecules of which the air is composed strike a theoretical surface of any orientation.'), (MetVariable(short_name='ciwc', standard_name='specific_cloud_ice_water_content', long_name='Specific cloud ice water content', level_type='isobaricInhPa', ecmwf_id=247, grib1_id=None, grib2_id=(0, 1, 84), units='kg kg**-1', amip=None, description="This parameter is the mass of cloud ice particles per kilogram of the total mass of moist air. The 'total mass of moist air' is the sum of the dry air, water vapour, cloud liquid, cloud ice, rain and falling snow. This parameter represents the average value for a grid box."), MetVariable(short_name='icmr', standard_name='ice_water_mixing_ratio', long_name='Cloud ice water mixing ratio', level_type='isobaricInhPa', ecmwf_id=260019, grib1_id=None, grib2_id=(0, 1, 23), units='kg kg**-1', amip=None, description='This parameter is the mass of cloud ice particles per kilogram of the total mass of dry air. ')))**

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

```
name = 'cocip'

optional_met_variables = ((MetVariable(short_name='z', standard_name='geopotential',
long_name='Geopotential', level_type='isobaricInhPa', ecmwf_id=129, grib1_id=6,
grib2_id=(0, 3, 4), units='m**2 s**-2', amip=None, description='Geopotential is the
sum of the specific gravitational potential energy relative to the geoid and the
specific centripetal potential energy.'), MetVariable(short_name='gh',
standard_name='geopotential_height', long_name='Geopotential Height',
level_type='isobaricInhPa', ecmwf_id=156, grib1_id=7, grib2_id=(0, 3, 5), units='m',
amip='zg', description='Geopotential is the sum of the specific gravitational
potential energy relative to the geoid and the specific centripetal potential
energy. Geopotential height is the geopotential divided by the standard acceleration
due to gravity. It is numerically similar to the altitude (or geometric height) and
not to the quantity with standard name height, which is relative to the surface.')),
(MetVariable(short_name='cc', standard_name='fraction_of_cloud_cover',
long_name='Cloud area fraction in atmosphere layer', level_type='isobaricInhPa',
ecmwf_id=248, grib1_id=None, grib2_id=(0, 6, 32), units='[0 - 1]', amip='cl',
description='This parameter is the proportion of a grid box covered by cloud (liquid
or ice) at a specific pressure level.'), MetVariable(short_name='tcc',
standard_name='total_cloud_cover_isobaric', long_name='Total cloud cover at isobaric
surface', level_type='isobaricInhPa', ecmwf_id=228164, grib1_id=None, grib2_id=(0,
6, 1), units='%', amip=None, description='This parameter is the percentage of a grid
box covered by cloud (liquid or ice) at a specific pressure level.')))
```

Additional met variables used to support outputs

Changed in version 0.48.0: Moved Geopotential from *met_variables* to *optional_met_variables*

processed_met_variables = (MetVariable(short_name='t',
standard_name='air_temperature', long_name='Air Temperature',
level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0),
units='K', amip='ta', description='Air temperature is the bulk temperature of the
air, not the surface (skin) temperature.'), MetVariable(short_name='q',
standard_name='specific_humidity', long_name='Specific Humidity',
level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg
kg**-1', amip='hus', description='Specific means per unit mass. Specific humidity is
the mass fraction of water vapor in (moist) air.'), MetVariable(short_name='u',
standard_name='eastward_wind', long_name='Eastward Wind',
level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m
s**-1', amip='ua', description='"Eastward" indicates a vector component which is
positive when directed eastward (negative westward). Wind is defined as a
two-dimensional (horizontal) air velocity vector, with no vertical component.'),
MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward
Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3),
units='m s**-1', amip='va', description='"Northward" indicates a vector component
which is positive when directed northward (negative southward). Wind is defined as a
two-dimensional (horizontal) air velocity vector, with no vertical component.'),
MetVariable(short_name='w', standard_name='lagrangian_tendency_of_air_pressure',
long_name='Vertical Velocity (omega)', level_type='isobaricInhPa', ecmwf_id=135,
grib1_id=39, grib2_id=(0, 2, 8), units='Pa s**-1', amip='wap', description='The
Lagrangian tendency of air pressure, often called "omega", plays the role of the
upward component of air velocity when air pressure is being used as the vertical
coordinate. If the vertical air velocity is upwards, it is negative when expressed
as a tendency of air pressure; downwards is positive. Air pressure is the force per
unit area which would be exerted when the moving gas molecules of which the air is
composed strike a theoretical surface of any orientation.'),
MetVariable(short_name='tau_cirrus', standard_name='tau_cirrus', long_name='Cirrus
optical depth', level_type=None, ecmwf_id=None, grib1_id=None, grib2_id=None,
units='dimensionless', amip=None, description=None))

Minimal set of met variables needed to run the model after pre-processing. The intention here is that `ciwc` is unnecessary after `tau_cirrus` has already been calculated.

rad

Radiation data formatted as a `MetDataset` at a single pressure level [-1]

rad_variables = ((MetVariable(short_name='tsr',
standard_name='top_net_solar_radiation', long_name='Top of atmosphere net solar
(shortwave) radiation', level_type='nominalTop', ecmwf_id=178, grib1_id=None,
grib2_id=(0, 4, 1), units='J m**-2', amip=None, description="This parameter is the
incoming solar radiation (also known as shortwave radiation) minus the outgoing
solar radiation at the top of the atmosphere. It is the amount of radiation passing
through a horizontal plane. The incoming solar radiation is the amount received from
the Sun. The outgoing solar radiation is the amount reflected and scattered by the
Earth's atmosphere and surfaceSee https://www.ecmwf.int/sites/default/files/
elibrary/2015/18490-radiation-quantities-ecmwf-model-and-mars.pdf"),
MetVariable(short_name='uswrf', standard_name='toa_upward_shortwave_flux',
long_name='Top of atmosphere upward shortwave radiation', level_type='nominalTop',
ecmwf_id=None, grib1_id=None, grib2_id=(0, 4, 193), units='W m**-2', amip=None,
description='This parameter is the outgoing shortwave (solar) radiation at the
nominal top of the atmosphere.')), (MetVariable(short_name='ttr',
standard_name='top_net_thermal_radiation', long_name='Top of atmosphere net thermal
(longwave) radiation', level_type='nominalTop', ecmwf_id=179, grib1_id=None,
grib2_id=(0, 5, 5), units='J m**-2', amip=None, description='The thermal (also known
as terrestrial or longwave) radiation emitted to space at the top of the atmosphere
is commonly known as the Outgoing Longwave Radiation (OLR). The top net thermal
radiation (this parameter) is equal to the negative of OLR.See
https://www.ecmwf.int/sites/default/files/elibrary/2015/
18490-radiation-quantities-ecmwf-model-and-mars.pdf'),
MetVariable(short_name='ulwrf', standard_name='toa_upward_longwave_flux',
long_name='Top of atmosphere upward longwave radiation', level_type='nominalTop',
ecmwf_id=None, grib1_id=None, grib2_id=(0, 5, 193), units='W m**-2', amip=None,
description='This parameter is the outgoing longwave (thermal) radiation at the
nominal top of the atmosphere.')))

Required single-level top of atmosphere radiation variables. Variable keys depend on data source (e.g. ECMWF, GFS).

timesteps

Array of `numpy.datetime64` time steps for contrail evolution

pycontrails.models.cocip.CocipParams

class pycontrails.models.cocip.**CocipParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(10.0, 10.0)*, *met_latitude_buffer=(10.0, 10.0)*, *met_level_buffer=(40.0, 40.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *process_emissions=True*, *dt_integration=numpy.timedelta64(30, 'm')*, *dz_m=200.0*, *effective_vertical_resolution=2000.0*, *smooth_true_airspeed=True*, *smooth_true_airspeed_window_length=7*, *smooth_true_airspeed_polyorder=1*, *humidity_scaling=None*, *compute_tau_cirrus_in_model_init='auto'*, *filter_sac=True*, *filter_initially_persistent=True*, *persistent_buffer=None*, *verbose_outputs=False*, *compute_atr20=False*, *global_rf_to_atr20_factor=0.0151*, *initial_wake_vortex_depth=0.5*, *sedimentation_impact_factor=0.5*, *default_nvpm_ei_n=1000000000000000.0*, *wind_shear_enhancement_exponent=0.5*, *nvpm_ei_n_enhancement_factor=1.0*, *min_ice_particle_number_nvpm_ei_n=10000000000000.0*, *max_depth=1500.0*, *unterstrasser_ice_survival_fraction=False*, *radiative_heating_effects=False*, *contrail_contrail_overlapping=False*, *dz_overlap_m=500.0*, *radius_threshold_um=<factory>*, *habits=<factory>*, *habit_distributions=<factory>*, *rf_sw_enhancement_factor=1.0*, *rf_lw_enhancement_factor=1.0*, *min_altitude_m=6000.0*, *max_altitude_m=13000.0*, *max_seg_length_m=40000.0*, *max_age=numpy.timedelta64(20, 'h')*, *min_tau=1e-06*, *max_tau=10000000000.0*, *min_n_ice_per_m3=1000.0*, *max_n_ice_per_m3=1e+20*)

Bases: *ModelParams*

Model parameters required by the CoCiP models.

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(10.0, 10.0)*, *met_latitude_buffer=(10.0, 10.0)*, *met_level_buffer=(40.0, 40.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *process_emissions=True*, *dt_integration=numpy.timedelta64(30, 'm')*, *dz_m=200.0*, *effective_vertical_resolution=2000.0*, *smooth_true_airspeed=True*, *smooth_true_airspeed_window_length=7*, *smooth_true_airspeed_polyorder=1*, *humidity_scaling=None*, *compute_tau_cirrus_in_model_init='auto'*, *filter_sac=True*, *filter_initially_persistent=True*, *persistent_buffer=None*, *verbose_outputs=False*, *compute_atr20=False*, *global_rf_to_atr20_factor=0.0151*, *initial_wake_vortex_depth=0.5*, *sedimentation_impact_factor=0.5*, *default_nvpm_ei_n=1000000000000000.0*, *wind_shear_enhancement_exponent=0.5*, *nvpm_ei_n_enhancement_factor=1.0*, *min_ice_particle_number_nvpm_ei_n=10000000000000.0*, *max_depth=1500.0*, *unterstrasser_ice_survival_fraction=False*, *radiative_heating_effects=False*, *contrail_contrail_overlapping=False*, *dz_overlap_m=500.0*, *radius_threshold_um=<factory>*, *habits=<factory>*, *habit_distributions=<factory>*, *rf_sw_enhancement_factor=1.0*, *rf_lw_enhancement_factor=1.0*, *min_altitude_m=6000.0*, *max_altitude_m=13000.0*, *max_seg_length_m=40000.0*, *max_age=numpy.timedelta64(20, 'h')*, *min_tau=1e-06*, *max_tau=10000000000.0*, *min_n_ice_per_m3=1000.0*, *max_n_ice_per_m3=1e+20*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

compute_atr20	Add additional metric of ATR20 and global yearly mean RF to model output.
compute_tau_cirrus_in_model_init	Compute "tau_cirrus" variable in pressure-level met data during model initialization.
contrail_contrail_overlapping	Experimental.
copy_source	Copy input source data on eval
default_nvpm_ei_n	Default nvpm_ei_n value if no data provided and emissions calculations fails.
downselect_met	Downselect input MetDataset` to region around source.
dt_integration	Apply Euler's method with a fixed step size of dt_integration.
dz_m	Difference in altitude between top and bottom layer for stratification calculations, $[m]$.
dz_overlap_m	Experimental.
effective_vertical_resolution	Vertical resolution (m) associated to met data.
filter_initially_persistent	Filter out waypoints if they don't satisfy the initial persistent criteria Passing in a non-default value is unusual, but is included to allow for false negative calibration and model uncertainty studies.

continues on next page

Table 5 – continued from previous page

filter_sac	Filter out waypoints if the don't satisfy the SAC criteria Note that the SAC algorithm will still be run to calculate `T_critical_sac` for use estimating initial ice particle number.
global_rf_to_atr20_factor	Constant factor used to convert global- and year-mean RF, $[Wm^{-2}]$, to ATR20, $[K]$, given by [Yin *et al.*, 2023].
humidity_scaling	Humidity scaling
initial_wake_vortex_depth	Initial wake vortex depth scaling factor.
`interpolation_bounds_error`	If True, points lying outside interpolation will raise an error
`interpolation_fill_value`	Used for outside interpolation value if `interpolation_bounds_error` is False
`interpolation_localize`	Experimental.
`interpolation_method`	Interpolation method.
`interpolation_q_method`	Experimental.
`interpolation_use_indices`	Experimental.
max_age	Max age of contrail evolution.
max_altitude_m	Maximum altitude domain in simulation, $[m]$ If set to `None`, this check is disabled.
max_depth	Upper bound for contrail plume depth, constraining it to realistic values.
max_n_ice_per_m3	Maximum contrail ice particle number per volume of air to prevent unrealistic values.
max_seg_length_m	Maximum contrail segment length in simulation to prevent unrealistic values, $[m]$.
max_tau	Maximum contrail optical depth to prevent unrealistic values.
met_latitude_buffer	Met latitude buffer [WGS84] for Cocip evolution.
met_level_buffer	Met level buffer $[hPa]$ for Cocip initialization and evolution.
met_longitude_buffer	Met longitude [WGS84] buffer for Cocip evolution.
`met_time_buffer`	Met time buffer for input to `Flight.downselect_met()` Only applies when `downselect_met` is True.
min_altitude_m	Minimum altitude domain in simulation, $[m]$ If set to `None`, this check is disabled.
min_ice_particle_number_nvpm_ei_n	Lower bound for `nvpm_ei_n` to account for ambient aerosol particles for newer engines, $[kg^{-1}]$
min_n_ice_per_m3	Minimum contrail ice particle number per volume of air.
min_tau	Minimum contrail optical depth.
nvpm_ei_n_enhancement_factor	Multiply flight black carbon number by enhancement factor.
persistent_buffer	Continue evolving contrail waypoints `persistent_buffer` beyond end of contrail life.
process_emissions	Determines whether `Cocip.process_emissions()` runs on model *Cocip.eval()* Set to `False` when input Flight includes emissions data.
radiative_heating_effects	Experimental.

continues on next page

Table 5 – continued from previous page

rf_lw_enhancement_factor	Scale longwave radiative forcing.
rf_sw_enhancement_factor	Scale shortwave radiative forcing.
sedimentation_impact_factor	Sedimentation impact factor.
smooth_true_airspeed	Smoothing parameters for true airspeed.
smooth_true_airspeed_polyorder	
smooth_true_airspeed_window_length	
unterstrasser_ice_survival_fraction	Experimental.
verbose_outputs	Add additional values to the flight and contrail that are not explicitly necessary for calculation.
verify_met	Call `_verify_met()` on model instantiation.
wind_shear_enhancement_exponent	Parameter denoted by n in eq.
radius_threshold_um	Radius threshold for regime bins, [μm] This is the row index label for `habit_distributions`.
habits	Particle habit (shape) types.
habit_distributions	Mix of ice particle habits in each radius regime.

compute_atr20 = False

Add additional metric of ATR20 and global yearly mean RF to model output. These are not standard CoCiP outputs but based on the derivation used in the first supplement to [Yin *et al.*, 2023]. ATR20 is defined as the average temperature response over a 20 year horizon.

compute_tau_cirrus_in_model_init = 'auto'

Compute `"tau_cirrus"` variable in pressure-level met data during model initialization. Must be one of `"auto"`, `True`, or `False`. If set to `"auto"`, `"tau_cirrus"` will be computed during model initialization iff the met data is dask-backed. Otherwise, it will be computed during model evaluation after the met data is downselected.

contrail_contrail_overlapping = False

Experimental. Radiative effects due to contrail-contrail overlapping Account for change in local contrail shortwave and longwave radiative forcing due to contrail-contrail overlapping.

New in version 0.45.

default_nvpm_ei_n = 1000000000000000.0

Default `nvpm_ei_n` value if no data provided and emissions calculations fails.

dt_integration = numpy.timedelta64(30,'m')

Apply Euler's method with a fixed step size of `dt_integration`. Advected waypoints are interpolated against met data once each `dt_integration`.

dz_m = 200.0

Difference in altitude between top and bottom layer for stratification calculations, [m]. Used to approximate derivative of "lagrangian_tendency_of_air_pressure" layer.

dz_overlap_m = 500.0

Experimental. Contrail-contrail overlapping altitude interval If *contrail_contrail_overlapping* is set to True, contrails will be grouped into altitude intervals, and all contrails within each altitude interval are treated as one contrail layer where they do not overlap.

effective_vertical_resolution = 2000.0

Vertical resolution (m) associated to met data. Constant below applies to ECMWF data.

filter_initially_persistent = True

Filter out waypoints if they don't satisfy the initial persistent criteria Passing in a non-default value is unusual, but is included to allow for false negative calibration and model uncertainty studies.

filter_sac = True

Filter out waypoints if the don't satisfy the SAC criteria Note that the SAC algorithm will still be run to calculate `T_critical_sac` for use estimating initial ice particle number. Passing in a non-default value is unusual, but is included to allow for false negative calibration and model uncertainty studies.

global_rf_to_atr20_factor = 0.0151

Constant factor used to convert global- and year-mean RF, $[Wm^{-2}]$, to ATR20, $[K]$, given by [Yin *et al.*, 2023].

habit_distributions

Mix of ice particle habits in each radius regime. Rows indexes are `radius_threshold_um` elements. Columns indexes are `habits` particle habit type. See Table 2 from [Schumann *et al.*, 2011].

habits

Particle habit (shape) types. This is the column index label for `habit_distributions`. See Table 2 in [Schumann *et al.*, 2011].

humidity_scaling = None

Humidity scaling

initial_wake_vortex_depth = 0.5

Initial wake vortex depth scaling factor. This factor scales max contrail downward displacement after the wake vortex phase to set the initial contrail depth. Denoted C_{D0} in eq (14) in [Schumann, 2012].

max_age = numpy.timedelta64(20,'h')

Max age of contrail evolution.

max_altitude_m = 13000.0

Maximum altitude domain in simulation, $[m]$ If set to `None`, this check is disabled.

max_depth = 1500.0

Upper bound for contrail plume depth, constraining it to realistic values. CoCiP only uses the ambient conditions at the mid-point of the Gaussian plume, and the edges could be in subsaturated conditions and sublimate. Important when *radiative_heating_effects* is enabled.

max_n_ice_per_m3 = 1e+20

Maximum contrail ice particle number per volume of air to prevent unrealistic values.

max_seg_length_m = 40000.0

Maximum contrail segment length in simulation to prevent unrealistic values, $[m]$.

max_tau = 10000000000.0

Maximum contrail optical depth to prevent unrealistic values.

met_latitude_buffer = (10.0, 10.0)

Met latitude buffer [WGS84] for Cocip evolution.

met_level_buffer = (40.0, 40.0)

Met level buffer $[hPa]$ for Cocip initialization and evolution.

met_longitude_buffer = (10.0, 10.0)

Met longitude [WGS84] buffer for Cocip evolution.

min_altitude_m = 6000.0

Minimum altitude domain in simulation, [m] If set to `None`, this check is disabled.

min_ice_particle_number_nvpm_ei_n = 10000000000000.0

Lower bound for `nvpm_ei_n` to account for ambient aerosol particles for newer engines, [kg^{-1}]

min_n_ice_per_m3 = 1000.0

Minimum contrail ice particle number per volume of air.

min_tau = 1e-06

Minimum contrail optical depth.

nvpm_ei_n_enhancement_factor = 1.0

Multiply flight black carbon number by enhancement factor. A value of 1.0 provides no scaling. Primarily used to support uncertainty estimation.

persistent_buffer = None

Continue evolving contrail waypoints `persistent_buffer` beyond end of contrail life. Passing in a non-default value is unusual, but is included to allow for false negative calibration and model uncertainty studies.

process_emissions = True

Determines whether `Cocip.process_emissions()` runs on model *`Cocip.eval()`* Set to `False` when input Flight includes emissions data.

radiative_heating_effects = False

Experimental. Radiative heating effects on contrail cirrus properties. Terrestrial and solar radiances warm the contrail ice particles and cause convective turbulence. This effect is expected to enhance vertical mixing and reduce the lifetime of contrail cirrus. This parameter is experimental, and the CoCiP implementation of this parameter may change.

New in version 0.28.9.

radius_threshold_um

Radius threshold for regime bins, [μm] This is the row index label for `habit_distributions`. See Table 2 in [Schumann *et al.*, 2011].

rf_lw_enhancement_factor = 1.0

Scale longwave radiative forcing. Primarily used to support uncertainty estimation.

rf_sw_enhancement_factor = 1.0

Scale shortwave radiative forcing. Primarily used to support uncertainty estimation.

sedimentation_impact_factor = 0.5

Sedimentation impact factor. Denoted by f_T in eq. (35) of [Schumann, 2012]. Schumann describes this as "an important adjustable parameter", and sets it to 0.1 in the original publication. In [Schumann and Graf, 2013], a value of 0.5 is referenced after comparing CoCiP predictions to observations.

smooth_true_airspeed = True

Smoothing parameters for true airspeed. Only used for Flight models. Passed directly to `scipy.signal.savgol_filter()`. See *`pycontrails.Flight.segment_true_airspeed()`* for details.

smooth_true_airspeed_polyorder = 1

smooth_true_airspeed_window_length = 7

unterstrasser_ice_survival_fraction = False

Experimental. Improved ice crystal number survival fraction in the wake vortex phase. Implement [Unterstrasser, 2016], who developed a parametric model that estimates the survival fraction of the contrail ice crystal number after the wake vortex phase based on the results from large eddy simulations. This replicates Fig. 4 of [Kärcher, 2018].

New in version 0.50.1.

verbose_outputs = False

Add additional values to the flight and contrail that are not explicitly necessary for calculation. See also `CocipGridParams.verbose_outputs_formation` and `CocipGridParams.verbose_outputs_evolution`.

wind_shear_enhancement_exponent = 0.5

Parameter denoted by n in eq. (39) of [Schumann, 2012].

pycontrails.models.cocip.contrail_properties

Contrail Property Calculations.

Functions

contrail_edges(lon, lat, sin_a, cos_a, width)	Calculate the longitude and latitude of the contrail edges to account for contrail spreading.
contrail_optical_depth(r_ice_vol, ...)	Calculate the contrail optical depth for each waypoint.
contrail_persistent(latitude, altitude, ...)	Determine surviving contrail segments after time integration step.
contrail_vertices(lon, lat, sin_a, cos_a, ...)	Calculate the longitude and latitude of the contrail vertices.
energy_forcing(rf_net_t1, rf_net_t2, ...)	Calculate the contrail energy forcing over time step.
horizontal_diffusivity(ds_dz, depth)	Calculate contrail horizontal diffusivity.
ice_particle_activation_rate(...)	Calculate the activation rate of black carbon particles to contrail ice crystals.
ice_particle_mass(r_ice_vol)	Calculate the contrail ice particle mass.
ice_particle_number(nvpm_ei_n, fuel_dist, ...)	Calculate the initial number of ice particles per distance after the wake vortex phase.
ice_particle_number_per_mass_of_air(...)	Calculate the number of contrail ice particles per mass of air.
ice_particle_number_per_volume_of_plume(...)	Calculate the number of contrail ice particles per volume of plume (`n_ice_per_vol`).
ice_particle_survival_fraction(iwc, iwc_1)	Estimate the fraction of contrail ice particle number that survive the wake vortex phase.
ice_particle_terminal_fall_speed(...)	Calculate the terminal fall speed of contrail ice particles.
ice_particle_volume_mean_radius(iwc, ...)	Calculate the ice particle volume mean radius.
initial_iwc(air_temperature, ...)	Estimate the initial contrail ice water content (iwc) before the wake vortex phase.
initial_persistent(iwc_1, rhi_1)	Determine if waypoints have persistent contrails.
iwc_adiabatic_heating(air_temperature, ...)	Calculate the change in ice water content due to adiabatic heating from the wake vortex phase.
iwc_post_wake_vortex(iwc, iwc_ad)	Calculate the ice water content after the wake vortex phase (`iwc_1`).

continues on next page

Table 6 – continued from previous page

light_wave_phase_delay(r_ice_vol)	Calculate the phase delay of the light wave passing through the contrail ice particle.
mean_energy_flux_per_m(rad_flux_per_m, dt)	Calculate the mean energy flux per length of contrail on segment following waypoint.
mean_radiative_flux_per_m(rf_net_t1, ...)	Calculate the mean radiative flux per length of contrail between two time steps.
new_contrail_dimensions(sigma_yy_t2, sigma_zz_t2)	Calculate the new contrail width and depth.
new_effective_area_from_sigma(sigma_yy, ...)	Calculate effective cross-sectional area of contrail plume (`area_eff`) from sigma parameters.
new_ice_particle_number(n_ice_per_m_t1, ...)	Calculate the number of ice particles per distance at the end of the time step.
new_ice_water_content(iwc_t1, q_t1, q_t2, ...)	Calculate the new contrail ice water content after the time integration step (`iwc_t2`).
particle_losses_aggregation(r_ice_vol, ...)	Calculate the rate of contrail ice particle losses due to sedimentation-induced aggregation.
particle_losses_turbulence(width, depth, ...)	Calculate the rate of contrail ice particle losses due to plume-internal turbulence.
plume_effective_cross_sectional_area(width, ...)	Calculate the effective cross-sectional area of the contrail plume (`area_eff`).
plume_effective_depth(width, area_eff)	Calculate the effective depth of the contrail plume (`depth_eff`).
plume_mass_per_distance(area_eff, rho_air)	Calculate the contrail plume mass per unit length.
plume_temporal_evolution(width_t1, depth_t1, ...)	Calculate the temporal evolution of the contrail plume parameters.
q_exhaust(air_temperature, air_pressure, ...)	Calculate the specific humidity released by water vapor from aircraft emissions.
scattering_extinction_efficiency(r_ice_vol)	Calculate the scattering extinction efficiency (`q_ext`) based on Mie-theory.
segment_length_ratio(seg_length_t1, ...)	Calculate the ratio of contrail segment length pre-advection to post-advection.
temperature_adiabatic_heating(...)	Calculate the ambient air temperature for each waypoint after the wake vortex phase.
vertical_diffusivity(air_pressure, ...)	Calculate contrail vertical diffusivity.

pycontrails.models.cocip.contrail_properties.**contrail_edges**(*lon*, *lat*, *sin_a*, *cos_a*, *width*)

Calculate the longitude and latitude of the contrail edges to account for contrail spreading.

(lon_edge_l, lat_edge_l) x——————

(Contrail midpoint: lon, lat) X==========

(lon_edge_r, lat_edge_r) x——————

Parameters

- **lon** (`npt.NDArray[np.float64]`) – longitude of contrail waypoint, degrees
- **lat** (`npt.NDArray[np.float64]`) – latitude of contrail waypoint, degrees
- **sin_a** (`npt.NDArray[np.float64]`) – sin(a), where a is the angle between the plume and the longitudinal axis
- **cos_a** (`npt.NDArray[np.float64]`) – cos(a), where a is the angle between the plume and the longitudinal axis
- **width** (`npt.NDArray[np.float64]`) – contrail width at each waypoint, $[m]$

Returns

tuple[npt.NDArray[np.float64], npt.NDArray[np.float64], npt.NDArray[np.float64], npt.NDArray[np.float64]] – (lon_edge_l, lat_edge_l, lon_edge_r, lat_edge_r), longitudes and latitudes at the left and right edges of the contrail, degrees

pycontrails.models.cocip.contrail_properties.**contrail_optical_depth**(*r_ice_vol*, *n_ice_per_m*, *width*)

Calculate the contrail optical depth for each waypoint.

Parameters

- **r_ice_vol** (npt.NDArray[np.float64]) – ice particle volume mean radius, $[m]$
- **n_ice_per_m** (npt.NDArray[np.float64]) – Number of contrail ice particles per distance, $[m^{-1}]$
- **width** (npt.NDArray[np.float64]) – Contrail width, $[m]$

Returns

npt.NDArray[np.float64] – Contrail optical depth

pycontrails.models.cocip.contrail_properties.**contrail_persistent**(*latitude*, *altitude*, *segment_length*, *age*, *tau_contrail*, *n_ice_per_m3*, *params*)

Determine surviving contrail segments after time integration step.

A contrail waypoint reaches its end of life if any of the following conditions hold:

1. Contrail age exceeds max_age
2. Contrail optical depth lies outside of interval [min_tau, max_tau]
3. Ice number density lies outside of interval [min_n_ice_per_m3, max_n_ice_per_m3]
4. Altitude lies outside of the interval [min_altitude_m, max_altitude_m]
5. Segment length exceeds max_seg_length_m
6. Latitude values are within 1 degree of the north or south pole

This function warns if all values in the tau_contrail array are nan.

Changed in version 0.25.10: Extreme values of latitude (ie, close to the north or south pole) now create an end of life condition. This check helps address issues related to divergence in the polar regions. (With large enough integration time delta, it is still possible for post-advection latitude values to lie outside of [-90, 90], but this is no longer possible with typical parameters and wind speeds.)

Parameters

- **latitude** (npt.NDArray[np.float64]) – Contrail latitude, $[\mathrm{deg}]$
- **altitude** (npt.NDArray[np.float64]) – Contrail altitude, $[m]$
- **segment_length** (npt.NDArray[np.float64]) – Contrail segment length, $[m]$
- **age** (npt.NDArray[np.timedelta64]) – Contrail age
- **tau_contrail** (npt.NDArray[np.float64]) – Contrail optical depth
- **n_ice_per_m3** (npt.NDArray[np.float64]) – Contrail ice particle number per volume of air, $[m^{-3}]$
- **params** (dict[str, Any]) – Dictionary of CocipParams parameters determining the conditions for end of contrail life.

Returns

npt.NDArray[np.bool_] – Boolean array indicating surviving contrails. Persisting contrails will be marked as True.

pycontrails.models.cocip.contrail_properties.**contrail_vertices**(*lon*, *lat*, *sin_a*, *cos_a*, *width*, *segment_length*)

Calculate the longitude and latitude of the contrail vertices.

This is equivalent to running *contrail_edges()* at each contrail waypoint and associating the next continuous waypoint with the previous. This method is helpful when you want to treat each contrail waypoint independently.

(lon_1, lat_1) x——————x (lon_4, lat_4)

(Contrail waypoint: lon, lat) X==========

(lon_2, lat_2) x——————x (lon_3, lat_3)

Parameters

- **lon** (npt.NDArray[np.float64]) – longitude of contrail waypoint, degrees
- **lat** (npt.NDArray[np.float64]) – latitude of contrail waypoint, degrees
- **sin_a** (npt.NDArray[np.float64]) – sin(a), where a is the angle between the plume and the longitudinal axis
- **cos_a** (npt.NDArray[np.float64]) – cos(a), where a is the angle between the plume and the longitudinal axis
- **width** (npt.NDArray[np.float64]) – contrail width at each waypoint, [m]
- **segment_length** (npt.NDArray[np.float64]) – contrail length at each waypoint, [m]

Returns

tuple[npt.NDArray[np.float64], npt.NDArray[np.float64], npt.NDArray[np.float64], npt.NDArray[np.float64]] – (lon_1, lat_1, lon_2, lat_2, lon_3, lat_3, lon_4, lat_4) degrees

pycontrails.models.cocip.contrail_properties.**energy_forcing**(*rf_net_t1*, *rf_net_t2*, *width_t1*, *width_t2*, *seg_length_t2*, *dt*)

Calculate the contrail energy forcing over time step.

The contrail energy forcing is calculated as the local contrail net radiative forcing (RF', change in energy flux per contrail area) multiplied by its width and integrated over its length and lifetime.

Parameters

- **rf_net_t1** (npt.NDArray[np.float64]) – local contrail net radiative forcing at the start of the time step, [Wm^{-2}]
- **rf_net_t2** (npt.NDArray[np.float64]) – local contrail net radiative forcing at the end of the time step, [Wm^{-2}]
- **width_t1** (npt.NDArray[np.float64]) – contrail width at the start of the time step, [m]
- **width_t2** (npt.NDArray[np.float64]) – contrail width at the end of the time step, [m]
- **seg_length_t2** (npt.NDArray[np.float64] | float) – Segment length of contrail waypoint at the end of the time step, [m]
- **dt** (npt.NDArray[np.timedelta64] | np.timedelta64) – integrate contrails with time steps of dt, [s]

Returns

npt.NDArray[np.float64] – Contrail energy forcing over time step dt, [J].

pycontrails.models.cocip.contrail_properties.**horizontal_diffusivity**(*ds_dz*, *depth*)

Calculate contrail horizontal diffusivity.

Parameters

- **ds_dz** (`npt.NDArray[np.float64]`) – Total wind shear (eastward and northward winds) with respect to altitude (`dz`), $[ms^{-1}/Pa]$
- **depth** (`npt.NDArray[np.float64]`) – Contrail depth at each waypoint, $[m]$

Returns

`npt.NDArray[np.float64]` – horizontal diffusivity, $[m^2 s^{-1}]$

References

- [Schumann, 2012]

Notes

Accounts for the turbulence-induced diffusive contrail spreading in the horizontal direction.

pycontrails.models.cocip.contrail_properties.**ice_particle_activation_rate**(*air_temperature*, *T_crit_sac*)

Calculate the activation rate of black carbon particles to contrail ice crystals.

The activation rate is calculated as a function of the difference between the ambient temperature and the Schmidt-Appleman threshold temperature `T_crit_sac`.

Parameters

- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature at each waypoint before wake_wortex, $[K]$
- **T_crit_sac** (`npt.NDArray[np.float64]`) – estimated Schmidt-Appleman temperature threshold for contrail formation, $[K]$

Returns

`npt.NDArray[np.float64]` – Proportion of black carbon particles that activates to contrail ice parties.

Notes

The equation is not published but based on the raw data from [Bräuer *et al.*, 2021].

References

- [Bräuer *et al.*, 2021]

pycontrails.models.cocip.contrail_properties.**ice_particle_mass**(*r_ice_vol*)

Calculate the contrail ice particle mass.

It is calculated by multiplying the mean ice particle volume with the density of ice

Parameters

r_ice_vol (`npt.NDArray[np.float64]`) – Ice particle volume mean radius, $[m]$

Returns
npt.NDArray[np.float64] – Mean contrail ice particle mass, [kg]

pycontrails.models.cocip.contrail_properties.**ice_particle_number**(*nvpm_ei_n*, *fuel_dist*, *f_surv*, *air_temperature*, *T_crit_sac*, *min_ice_particle_number_nvpm_ei_n*)

Calculate the initial number of ice particles per distance after the wake vortex phase.

The initial number of ice particle per distance is calculated from the black carbon number emissions index nvpm_ei_n and fuel burn per distance fuel_dist. Note that a lower bound for nvpm_ei_n is set at 1e13 kg^{-1} to account for the activation of ambient aerosol particles and organic volatile particles.

Parameters

- **nvpm_ei_n** (npt.NDArray[np.float64]) – black carbon number emissions index, [kg^{-1}]
- **fuel_dist** (npt.NDArray[np.float64]) – fuel consumption of the flight segment per distance traveled, [kgm^{-1}]
- **f_surv** (npt.NDArray[np.float64]) – Fraction of contrail ice particle number that survive the wake vortex phase.
- **air_temperature** (npt.NDArray[np.float64]) – ambient temperature for each waypoint, [K]
- **T_crit_sac** (npt.NDArray[np.float64]) – estimated Schmidt-Appleman temperature threshold for contrail formation, [K]
- **min_ice_particle_number_nvpm_ei_n** (float) – lower bound for nvpm_ei_n to account for ambient aerosol particles for newer engines [kg^{-1}]

Returns
npt.NDArray[np.float64] – initial number of ice particles per distance after the wake vortex phase, [m^{-1}]

pycontrails.models.cocip.contrail_properties.**ice_particle_number_per_mass_of_air**(*n_ice_per_vol*, *rho_air*)

Calculate the number of contrail ice particles per mass of air.

Parameters

- **n_ice_per_vol** (npt.NDArray[np.float64]) – number of ice particles per volume of contrail plume at time t, [m^{-3}]
- **rho_air** (npt.NDArray[np.float64]) – density of air for each waypoint, [kgm^{-3}]

Returns
npt.NDArray[np.float64] – number of ice particles per mass of air at time t, [kg^{-1}]

pycontrails.models.cocip.contrail_properties.**ice_particle_number_per_volume_of_plume**(*n_ice_per_m*, *area_eff*)

Calculate the number of contrail ice particles per volume of plume (n_ice_per_vol).

Parameters

- **n_ice_per_m** (npt.NDArray[np.float64]) – number of ice particles per distance at time t, [m^{-1}]
- **area_eff** (npt.NDArray[np.float64]) – effective cross-sectional area of the contrail plume, [m^2]

Returns

`npt.NDArray[np.float64]` – number of ice particles per volume of contrail plume at time t, $[m^{-3}]$

pycontrails.models.cocip.contrail_properties.**ice_particle_survival_fraction**(*iwc*, *iwc_1*)

Estimate the fraction of contrail ice particle number that survive the wake vortex phase.

CoCiP assumes that this fraction is proportional to the change in ice water content (`iwc_1 - iwc`) before and after the wake vortex phase.

Parameters

- **iwc** (`npt.NDArray[np.float64]`) – initial ice water content at each waypoint before the wake vortex phase, $[kg_{H_2O}/kg_{air}]$
- **iwc_1** (`npt.NDArray[np.float64]`) – ice water content after the wake vortex phase, $[kg_{H_2O}/kg_{air}]$

Returns

`npt.NDArray[np.float64]` – Fraction of contrail ice particle number that survive the wake vortex phase.

pycontrails.models.cocip.contrail_properties.**ice_particle_terminal_fall_speed**(*air_pressure*, *air_temperature*, *r_ice_vol*)

Calculate the terminal fall speed of contrail ice particles.

`v_t` is calculated based on a parametric model from [Spichtinger and Gierens, 2009], using inputs of pressure level, ambient temperature and the ice particle volume mean radius. See Table 2 for the model parameters.

Parameters

- **air_pressure** (`npt.NDArray[np.float64]`) – Pressure altitude at each waypoint, $[Pa]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **r_ice_vol** (`npt.NDArray[np.float64]`) – Ice particle volume mean radius, $[m]$

Returns

`npt.NDArray[np.float64]` – Terminal fall speed of contrail ice particles, $[ms^{-1}]$

References

- [Spichtinger and Gierens, 2009]

pycontrails.models.cocip.contrail_properties.**ice_particle_volume_mean_radius**(*iwc*, *n_ice_per_kg_air*)

Calculate the ice particle volume mean radius.

Parameters

- **iwc** (`npt.NDArray[np.float64]`) – contrail ice water content, i.e., contrail ice mass per kg of air, $[kg_{H_2O}/kg_{air}]$
- **n_ice_per_kg_air** (`npt.NDArray[np.float64]`) – number of ice particles per mass of air, $[kg^{-1}]$

Returns

`npt.NDArray[np.float64]` – ice particle volume mean radius, $[m]$

Notes

`r_ice_vol` is the mean radius of a sphere that has the same volume as the contrail ice particle.

`r_ice_vol` calculated by dividing the total volume of contrail ice particle per kg of air (`total_ice_volume`, $m**3/kg-air$) with the number of contrail ice particles per kg of air (`n_ice_per_kg_air`, $/kg-air$).

pycontrails.models.cocip.contrail_properties.**initial_iwc**(*air_temperature*, *specific_humidity*, *air_pressure*, *fuel_dist*, *width*, *depth*, *ei_h2o*)

Estimate the initial contrail ice water content (iwc) before the wake vortex phase.

Note that the ice water content is replaced by zero if it is negative (dry air), and this will end the contrail life-cycle in subsequent steps.

Parameters

- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature for each waypoint, $[K]$
- **specific_humidity** (`npt.NDArray[np.float64]`) – ambient specific humidity for each waypoint, $[kg_{H_2O}/kg_{air}]$
- **air_pressure** (`npt.NDArray[np.float64]`) – initial pressure altitude at each waypoint, before the wake vortex phase, $[Pa]$
- **fuel_dist** (`npt.NDArray[np.float64]`) – fuel consumption of the flight segment per distance traveled, $[kgm^{-1}]$
- **width** (`npt.NDArray[np.float64]`) – initial contrail width, $[m]$
- **depth** (`npt.NDArray[np.float64]`) – initial contrail depth, $[m]$
- **ei_h2o** (`float`) – water vapor emissions index of fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

Returns

`npt.NDArray[np.float64]` – Initial contrail ice water content (iwc) at the original waypoint before the wake vortex phase, $[kg_{H_2O}/kg_{air}]$. Returns zero if iwc is is negative (dry air).

pycontrails.models.cocip.contrail_properties.**initial_persistent**(*iwc_1*, *rhi_1*)

Determine if waypoints have persistent contrails.

Conditions for persistent initial_contrails:

1. ice water content at level 1: `1e-12 < iwc < 1e10`
2. rhi at level 1: `0 < rhi < 1e10`

Changed in version 0.25.1: Returned array now has floating dtype. This is consistent with other filtering steps in the CoCiP model (ie, `sac`).

Parameters

- **iwc_1** (`npt.NDArray[np.float64]`) – ice water content after the wake vortex phase, $[kg_{H_2O}/kg_{air}]$
- **rhi_1** (`npt.NDArray[np.float64]`) – relative humidity with respect to ice after the wake vortex phase

Returns

`npt.NDArray[np.float64]` – Mask of waypoints with persistent contrails. Waypoints with persistent contrails will have value 1.

Notes

The RHi at level 1 does not need to be above 100% if the iwc > 0 kg/kg. If iwc > 0 and RHi < 100%, the contrail lifetime will not end immediately as the ice particles will gradually evaporate with the rate depending on the background RHi.

pycontrails.models.cocip.contrail_properties.**iwc_adiabatic_heating**(*air_temperature*, *air_pressure*, *air_pressure_1*)

Calculate the change in ice water content due to adiabatic heating from the wake vortex phase.

Parameters

- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature for each waypoint, $[K]$
- **air_pressure** (`npt.NDArray[np.float64]`) – initial pressure altitude at each waypoint, before the wake vortex phase, $[Pa]$
- **air_pressure_1** (`npt.NDArray[np.float64]`) – pressure altitude at each waypoint, after the wake vortex phase, $[Pa]$

Returns

`npt.NDArray[np.float64]` – Change in ice water content due to adiabatic heating from the wake vortex phase, $[kg_{H_2O}/kg_{air}]$

pycontrails.models.cocip.contrail_properties.**iwc_post_wake_vortex**(*iwc*, *iwc_ad*)

Calculate the ice water content after the wake vortex phase (`iwc_1`).

`iwc_1` is calculated by subtracting the initial iwc before the wake vortex phase (`iwc`) by the change in iwc from adiabatic heating experienced during the wake vortex phase.

Note that the iwc is replaced by zero if it is negative (dry air), and this will end the contrail lifecycle in subsequent steps.

Parameters

- **iwc** (`npt.NDArray[np.float64]`) – initial ice water content at each waypoint before the wake vortex phase, $[kg_{H_2O}/kg_{air}]$
- **iwc_ad** (`npt.NDArray[np.float64]`) – change in iwc from adiabatic heating during the wake vortex phase, $[kg_{H_2O}/kg_{air}]$

Returns

`npt.NDArray[np.float64]` – ice water content after the wake vortex phase, `iwc_1`, $[kg_{H_2O}/kg_{air}]$

Notes

Level 1, see Figure 1 of [Schumann, 2012]

References

- [Schumann, 2012]

pycontrails.models.cocip.contrail_properties.**light_wave_phase_delay**(*r_ice_vol*)

Calculate the phase delay of the light wave passing through the contrail ice particle.

Parameters
r_ice_vol (`npt.NDArray[np.float64]`) – ice particle volume mean radius, [m]

Returns
`npt.NDArray[np.float64]` – phase delay of the light wave passing through the contrail ice particle

References

- https://en.wikipedia.org/wiki/Mie_scattering

pycontrails.models.cocip.contrail_properties.**mean_energy_flux_per_m**(*rad_flux_per_m*, *dt*)

Calculate the mean energy flux per length of contrail on segment following waypoint.

Parameters

- **rad_flux_per_m** (`npt.NDArray[np.float64]`) – Mean radiative flux between time steps for waypoint, [Wm^{-1}]. See *mean_radiative_flux_per_m()*.
- **dt** (`npt.NDArray[np.timedelta64]`) – timedelta of integration timestep for each waypoint.

Returns
`npt.NDArray[np.float64]` – Mean energy flux per length of contrail after waypoint, [Jm^{-1}]

Notes

Implementation differs from original fortran in two ways:

- Discontinuity is no longer set to 0 (this occurs directly in model `Cocip`)
- Instead of taking an average of the previous and following segments, energy flux is only calculated for the following segment.

See also:

mean_radiative_flux_per_m()

pycontrails.models.cocip.contrail_properties.**mean_radiative_flux_per_m**(*rf_net_t1*, *rf_net_t2*, *width_t1*, *width_t2*)

Calculate the mean radiative flux per length of contrail between two time steps.

Parameters

- **rf_net_t1** (`npt.NDArray[np.float64]`) – local contrail net radiative forcing at the start of the time step, [Wm^{-2}]
- **rf_net_t2** (`npt.NDArray[np.float64]`) – local contrail net radiative forcing at the end of the time step, [Wm^{-2}]
- **width_t1** (`npt.NDArray[np.float64]`) – contrail width at the start of the time step, [m]
- **width_t2** (`npt.NDArray[np.float64]`) – contrail width at the end of the time step, [m]

Returns
npt.NDArray[np.float64] – Mean radiative flux between time steps, [Wm^{-1}]

pycontrails.models.cocip.contrail_properties.**new_contrail_dimensions**(*sigma_yy_t2*, *sigma_zz_t2*)

Calculate the new contrail width and depth.

Parameters

- **sigma_yy_t2** (npt.NDArray[np.float64]) – element yy, covariance matrix of the Gaussian concentration field, Eq. (6) of Schumann (2012)
- **sigma_zz_t2** (npt.NDArray[np.float64]) – element zz, covariance matrix of the Gaussian concentration field, Eq. (6) of Schumann (2012)

Returns

- **width_t2** (npt.NDArray[np.float64]) – Contrail width at the end of the time step, [m]
- **depth_t2** (npt.NDArray[np.float64]) – Contrail depth at the end of the time step, [m]

pycontrails.models.cocip.contrail_properties.**new_effective_area_from_sigma**(*sigma_yy*, *sigma_zz*, *sigma_yz*)

Calculate effective cross-sectional area of contrail plume (area_eff) from sigma parameters.

This method calculates the same output as :func`plume_effective_cross_sectional_area`, but calculated with different input parameters.

Parameters

- **sigma_yy** (npt.NDArray[np.float64]) – element yy, covariance matrix of the Gaussian concentration field, Eq. (6) of Schumann (2012)
- **sigma_zz** (npt.NDArray[np.float64]) – element zz, covariance matrix of the Gaussian concentration field, Eq. (6) of Schumann (2012)
- **sigma_yz** (npt.NDArray[np.float64] | float) – element yz, covariance matrix of the Gaussian concentration field, Eq. (6) of Schumann (2012)

Returns
npt.NDArray[np.float64] – Effective cross-sectional area of the contrail plume (area_eff)

pycontrails.models.cocip.contrail_properties.**new_ice_particle_number**(*n_ice_per_m_t1*, *dn_dt_agg*, *dn_dt_turb*, *seg_ratio*, *dt*)

Calculate the number of ice particles per distance at the end of the time step.

Parameters

- **n_ice_per_m_t1** (npt.NDArray[np.float64]) – number of contrail ice particles per distance at the start of the time step, [m^{-1}]
- **dn_dt_agg** (npt.NDArray[np.float64]) – rate of ice particle losses due to sedimentation-induced aggregation, [s^{-1}]
- **dn_dt_turb** (npt.NDArray[np.float64]) – rate of contrail ice particle losses due to plume-internal turbulence, [s^{-1}]
- **seg_ratio** (npt.NDArray[np.float64] | float) – Segment length ratio before and after it is advected to the new location.
- **dt** (npt.NDArray[np.timedelta64] | np.timedelta64) – integrate contrails with time steps of dt, [s]

Returns

npt.NDArray[np.float64] – number of ice particles per distance at the end of the time step, $[m^{-1}]$

pycontrails.models.cocip.contrail_properties.**new_ice_water_content**(*iwc_t1*, *q_t1*, *q_t2*, *q_sat_t1*, *q_sat_t2*, *mass_plume_t1*, *mass_plume_t2*)

Calculate the new contrail ice water content after the time integration step (iwc_t2).

Parameters

- **iwc_t1** (npt.NDArray[np.float64]) – contrail ice water content, i.e., contrail ice mass per kg of air, at the start of the time step, $[kg_{H_2O}/kg_{air}]$
- **q_t1** (npt.NDArray[np.float64]) – specific humidity for each waypoint at the start of the time step, $[kg_{H_2O}/kg_{air}]$
- **q_t2** (npt.NDArray[np.float64]) – specific humidity for each waypoint at the end of the time step, $[kg_{H_2O}/kg_{air}]$
- **q_sat_t1** (npt.NDArray[np.float64]) – saturation humidity for each waypoint at the start of the time step, $[kg_{H_2O}/kg_{air}]$
- **q_sat_t2** (npt.NDArray[np.float64]) – saturation humidity for each waypoint at the end of the time step, $[kg_{H_2O}/kg_{air}]$
- **mass_plume_t1** (npt.NDArray[np.float64]) – contrail plume mass per unit length at the start of the time step, $[kg_{air}m^{-1}]$
- **mass_plume_t2** (npt.NDArray[np.float64]) – contrail plume mass per unit length at the end of the time step, $[kg_{air}m^{-1}]$

Returns

npt.NDArray[np.float64] – Contrail ice water content at the end of the time step, $[kg_{ice}kg_{air}^{-1}]$

Notes

(1) The ice water content is fully conservative.

(2) mass_h2o_t2: the total H2O mass (ice + vapour) per unit of contrail plume [Units of kg-H2O/m]

(3) q_sat is used to calculate mass_h2o because air inside the contrail is assumed to be ice saturated.

(4) (mass_plume_t2 - mass_plume) * q_mean: contrail absorbs (releases) H2O from (to) surrounding air.

(5) iwc_t2 = mass_h2o_t2 / mass_plume_t2 - q_sat_t2: H2O in the gas phase is removed (- q_sat_t2).

pycontrails.models.cocip.contrail_properties.**particle_losses_aggregation**(*r_ice_vol*, *terminal_fall_speed*, *area_eff*, *agg_efficiency=1.0*)

Calculate the rate of contrail ice particle losses due to sedimentation-induced aggregation.

Parameters

- **r_ice_vol** (npt.NDArray[np.float64]) – Ice particle volume mean radius, $[m]$

- **terminal_fall_speed** (`npt.NDArray[np.float64]`) – Terminal fall speed of contrail ice particles, $[ms^{-1}]$
- **area_eff** (`npt.NDArray[np.float64]`) – Effective cross-sectional area of the contrail plume, $[m^2]$
- **agg_efficiency** (`float`, *optional*) – Aggregation efficiency

Returns

`npt.NDArray[np.float64]` – Rate of contrail ice particle losses due to sedimentation-induced aggregation, $[s^{-1}]$

Notes

The aggregation efficiency (`agg_efficiency = 1`) was calibrated based on the observed lifetime and optical properties from the Contrail Library (COLI) database ([Schumann *et al.*, 2017]).

References

- [Schumann *et al.*, 2017]

pycontrails.models.cocip.contrail_properties.**particle_losses_turbulence**(*width*, *depth*, *depth_eff*, *diffuse_h*, *diffuse_v*, *turb_efficiency=0.1*)

Calculate the rate of contrail ice particle losses due to plume-internal turbulence.

Parameters

- **width** (`npt.NDArray[np.float64]`) – Contrail width at each waypoint, $[m]$
- **depth** (`npt.NDArray[np.float64]`) – Contrail depth at each waypoint, $[m]$
- **depth_eff** (`npt.NDArray[np.float64]`) – Effective depth of the contrail plume, $[m]$
- **diffuse_h** (`npt.NDArray[np.float64]`) – Horizontal diffusivity, $[m^2 s^{-1}]$
- **diffuse_v** (`npt.NDArray[np.float64]`) – Vertical diffusivity, $[m^2 s^{-1}]$
- **turb_efficiency** (`float`, *optional*) – Turbulence sublimation efficiency

Returns

`npt.NDArray[np.float64]` – Rate of contrail ice particle losses due to plume-internal turbulence, $[s^{-1}]$

Notes

The turbulence sublimation efficiency (`turb_efficiency = 0.1`) was calibrated based on the observed lifetime and optical properties from the Contrail Library (COLI) database ([Schumann *et al.*, 2017]).

References

- [Schumann *et al.*, 2017]

pycontrails.models.cocip.contrail_properties.**plume_effective_cross_sectional_area**(*width*, *depth*, *sigma_yz*)

Calculate the effective cross-sectional area of the contrail plume (`area_eff`).

`sigma_yy`, `sigma_zz` and `sigma_yz` are the parameters governing the contrail plume's temporal evolution.

Parameters

- **width** (`npt.NDArray[np.float64]`) – contrail width at each waypoint, [m]
- **depth** (`npt.NDArray[np.float64]`) – contrail depth at each waypoint, [m]
- **sigma_yz** (`npt.NDArray[np.float64] | float`) – temporal evolution of the contrail plume parameters

Returns

`npt.NDArray[np.float64]` – effective cross-sectional area of the contrail plume, [m^2]

pycontrails.models.cocip.contrail_properties.**plume_effective_depth**(*width*, *area_eff*)

Calculate the effective depth of the contrail plume (`depth_eff`).

`depth_eff` is calculated from the effective cross-sectional area (`area_eff`) and the contrail width.

Parameters

- **width** (`npt.NDArray[np.float64]`) – contrail width at each waypoint, [m]
- **area_eff** (`npt.NDArray[np.float64]`) – effective cross-sectional area of the contrail plume, [m^2]

Returns

`npt.NDArray[np.float64]` – effective depth of the contrail plume, [m]

pycontrails.models.cocip.contrail_properties.**plume_mass_per_distance**(*area_eff*, *rho_air*)

Calculate the contrail plume mass per unit length.

Parameters

- **area_eff** (`npt.NDArray[np.float64]`) – effective cross-sectional area of the contrail plume, [m^2]
- **rho_air** (`npt.NDArray[np.float64]`) – density of air for each waypoint, [kgm^{-3}]

Returns

`npt.NDArray[np.float64]` – contrail plume mass per unit length, [kgm^{-1}]

pycontrails.models.cocip.contrail_properties.**plume_temporal_evolution**(*width_t1*, *depth_t1*, *sigma_yz_t1*, *dsn_dz_t1*, *diffuse_h_t1*, *diffuse_v_t1*, *seg_ratio*, *dt*, *max_depth*)

Calculate the temporal evolution of the contrail plume parameters.

Refer to equation (6) of Schumann (2012). See also equations (29), (30), and (31).

Parameters

- **width_t1** (`npt.NDArray[np.float64]`) – contrail width at the start of the time step, [m]

- **depth_t1** (npt.NDArray[np.float64]) – contrail depth at the start of the time step, [m]
- **sigma_yz_t1** (npt.NDArray[np.float64]) – sigma_yz governs the contrail plume's temporal evolution at the start of the time step
- **dsn_dz_t1** (npt.NDArray[np.float64]) – vertical gradient of the horizontal velocity (wind shear) normal to the contrail axis at the start of the time step, [ms^{-1}/Pa]:

```
X----------------------X              X
             ^                        |
             | (dsn_dz)               |  <-- (dsn_dz)
             |                        |
                                      X
```

- **diffuse_h_t1** (npt.NDArray[np.float64]) – horizontal diffusivity at the start of the time step, [$m^2 s^{-1}$]
- **diffuse_v_t1** (npt.NDArray[np.float64]) – vertical diffusivity at the start of the time step, [$m^2 s^{-1}$]
- **seg_ratio** (npt.NDArray[np.float64 | float]) – Segment length ratio before and after it is advected to the new location. See *segment_length_ratio()*.
- **dt** (npt.NDArray[np.timedelta64] | np.timedelta64) – integrate contrails with time steps of dt, [s]
- **max_depth** (float | None) – Constrain maximum plume depth to prevent unrealistic values, [m]. If None is passed, the maximum plume depth is not constrained.

Returns

- **sigma_yy_t2** (npt.NDArray[np.float64]) – The yy component of convariance matrix, [m^2]
- **sigma_zz_t2** (npt.NDArray[np.float64]) – The zz component of convariance matrix, [m^2]
- **sigma_yz_t2** (npt.NDArray[np.float64]) – The yz component of convariance matrix, [m^2]

pycontrails.models.cocip.contrail_properties.**q_exhaust**(*air_temperature*, *air_pressure*, *fuel_dist*, *width*, *depth*, *ei_h2o*)

Calculate the specific humidity released by water vapor from aircraft emissions.

Parameters

- **air_temperature** (npt.NDArray[np.float64]) – ambient temperature for each waypoint, [K]
- **air_pressure** (npt.NDArray[np.float64]) – initial pressure altitude at each waypoint, before the wake vortex phase, [Pa]
- **fuel_dist** (npt.NDArray[np.float64]) – fuel consumption of the flight segment per distance travelled, [kgm^{-1}]
- **width** (npt.NDArray[np.float64]) – initial contrail width, [m]
- **depth** (npt.NDArray[np.float64]) – initial contrail depth, [m]
- **ei_h2o** (float) – water vapor emissions index of fuel, [$kg_{H_2O}\ kg_{fuel}^{-1}$]

Returns
npt.NDArray[np.float64] – Humidity released by water vapour from aircraft emissions, $[kg_{H_2O}/kg_{air}]$

pycontrails.models.cocip.contrail_properties.**scattering_extinction_efficiency**(*r_ice_vol*)

Calculate the scattering extinction efficiency (q_ext) based on Mie-theory.

Parameters
r_ice_vol (npt.NDArray[np.float64]) – ice particle volume mean radius, $[m]$

Returns
npt.NDArray[np.float64] – scattering extinction efficiency

References

- https://en.wikipedia.org/wiki/Mie_scattering

pycontrails.models.cocip.contrail_properties.**segment_length_ratio**(*seg_length_t1*, *seg_length_t2*)

Calculate the ratio of contrail segment length pre-advection to post-advection.

Parameters

- **seg_length_t1** (npt.NDArray[np.float64]) – Segment length of contrail waypoint at the start of the time step, $[m]$
- **seg_length_t2** (npt.NDArray[np.float64]) – Segment length of contrail waypoint after time step and advection, $[m]$

Returns
npt.NDArray[np.float64] – Ratio of segment length before advection to segment length after advection.

Notes

This implementation differs from the original fortran implementation. Instead of taking a geometric mean between the previous and following segments, a simple ratio is computed.

For terminal waypoints along a flight trajectory, the associated segment length is 0. In this case, the segment ratio is set to 1 (the naive ratio 0 / 0 is undefined). According to CoCiP conventions, terminus waypoints are "discontinuous" within the flight trajectory, and will not contribute to contrail calculations.

More broadly, any undefined (nan values, or division by 0) segment ratio is set to 1. This convention ensures that the contrail calculations are not affected by undefined segment-based properties.

Presently, the output of this function is only used by *plume_temporal_evolution()* and *new_ice_particle_number()* as a scaling term.

A *seg_ratio* value of 1 is the same as not applying any scaling in these two functions.

pycontrails.models.cocip.contrail_properties.**temperature_adiabatic_heating**(*air_temperature*, *air_pressure*, *air_pressure_1*)

Calculate the ambient air temperature for each waypoint after the wake vortex phase.

This calculation accounts for adiabatic heating.

Parameters

- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature for each waypoint, $[K]$
- **air_pressure** (`npt.NDArray[np.float64]`) – initial pressure altitude at each waypoint, before the wake vortex phase, $[Pa]$
- **air_pressure_1** (`npt.NDArray[np.float64]`) – pressure altitude at each waypoint, after the wake vortex phase, $[Pa]$

Returns

`npt.NDArray[np.float64]` – ambient air temperature after the wake vortex phase, $[K]$

Notes

Level 1, see Figure 1 of [Schumann, 2012]

References

- [Schumann, 2012]

pycontrails.models.cocip.contrail_properties.**vertical_diffusivity**(*air_pressure*, *air_temperature*, *dT_dz*, *depth_eff*, *terminal_fall_speed*, *sedimentation_impact_factor*, *eff_heat_rate*)

Calculate contrail vertical diffusivity.

Parameters

- **air_pressure** (`npt.NDArray[np.float64]`) – Pressure altitude at each waypoint, $[Pa]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **dT_dz** (`npt.NDArray[np.float64]`) – Temperature gradient with respect to altitude (dz), $[Km^{-1}]$
- **depth_eff** (`npt.NDArray[np.float64]`) – Effective depth of the contrail plume, $[m]$
- **terminal_fall_speed** (`npt.NDArray[np.float64]`) – Terminal fall speed of contrail ice particles, $[ms^{-1}]$
- **sedimentation_impact_factor** (`float`) – Enhancement parameter denoted by f_T in eq. (35) Schumann (2012).
- **eff_heat_rate** (`npt.NDArray[np.float64] | None`) – Effective heating rate, i.e., rate of which the contrail plume is heated, $[Ks^{-1}]$. If None is passed, the radiative heating effects on contrail cirrus properties are not included.

Returns

`npt.NDArray[np.float64]` – vertical diffusivity, $[m^2s^{-1}]$

References

- [Schumann, 2012]
- [Schumann and Graf, 2013]

Notes

Accounts for the turbulence-induced diffusive contrail spreading in the vertical direction. See eq. (35) of [Schumann, 2012].

The first term in Eq. (35) of [Schumann, 2012] is (c_V * w'_N^2 / N_BV, where c_V = 0.2 and w'_N^2 = 0.1) is different than outlined below. Here, a constant of 0.01 is used when radiative heating effects are not activated. This update comes from [Schumann and Graf, 2013] , which found that the original formulation estimated thinner contrails relative to satellite observations. The vertical diffusivity was enlarged so that the simulated contrails are more consistent with observations.

pycontrails.models.cocip.radiative_forcing

Module for calculating radiative forcing of contrail cirrus.

References

- [Schumann *et al.*, 2011]
- [Schumann *et al.*, 2012]

Functions

albedo(sdr, rsr)	Calculate albedo along contrail waypoint.
contrail_albedo(tau_contrail, mue, r_eff_um, ...)	Calculate the contrail albedo, `alpha_c`.
contrail_contrail_overlap_radiative_effects(	Calculate radiative properties after accounting for contrail overlapping.
contrail_effective_emissivity(r_eff_um, delta_lr)	Calculate the effective emissivity of the contrail, `f_lw`.
effective_radius_by_habit(r_vol_um, habit_idx)	Calculate the effective radius `r_eff_um` via the mean ice particle radius and habit type.
effective_radius_droxtal(r_vol_um)	Calculate the effective radius of contrail ice particles assuming a droxtal particle habit.
effective_radius_hollow_column(r_vol_um)	Calculate the effective radius of ice particles assuming a hollow column particle habit.
effective_radius_myhre(r_vol_um)	Calculate the effective radius of contrail ice particles assuming a sphere particle habit.
effective_radius_plate(r_vol_um)	Calculate the effective radius of contrail ice particles assuming a plate particle habit.
effective_radius_rosette(r_vol_um)	Calculate the effective radius of contrail ice particles assuming a rosette particle habit.
effective_radius_rough_aggregate(r_vol_um)	Calculate the effective radius of ice particles assuming a rough aggregate particle habit.
effective_radius_solid_column(r_vol_um)	Calculate the effective radius of contrail ice particles assuming a solid column particle habit.
effective_radius_sphere(r_vol_um)	Calculate the effective radius of contrail ice particles assuming a sphere particle habit.
effective_tau_cirrus(tau_cirrus, mue, ...)	Calculate the effective optical depth of natural cirrus above the contrail, `e_sw`.
habit_weight_regime_idx(r_vol_um, ...)	Determine regime of ice particle habits based on contrail ice particle volume mean radius.
habit_weights(r_vol_um, habit_distributions, ...)	Assign weights to different ice particle habits for each waypoint.
longwave_radiative_forcing(r_vol_um, olr, ...)	Calculate the local contrail longwave radiative forcing (RF_{LW}).
net_radiative_forcing(rf_lw, rf_sw)	Calculate the local contrail net radiative forcing (rf_net).
olr_reduction_natural_cirrus(tau_cirrus, ...)	Calculate reduction in outgoing longwave radiation (OLR) due to the presence of natural cirrus.
shortwave_radiative_forcing(r_vol_um, sdr, ...)	Calculate the local contrail shortwave radiative forcing (RF_{SW}).

Classes

RFConstants()	Constants that are used to calculate the local contrail radiative forcing.

class pycontrails.models.cocip.radiative_forcing.**RFConstants**

Bases: object

Constants that are used to calculate the local contrail radiative forcing.

See Table 1 of [Schumann *et al.*, 2012].

Each coefficient has 8 elements, one corresponding to each contrail ice particle habit (shape):

```
[
    Sphere,
    Solid column,
    Hollow column,
    Rough aggregate,
    Rosette-6,
    Plate,
    Droxtal,
    Myhre,
]
```

For each waypoint, the distinct mix of ice particle habits are approximated using the mean contrail ice particle radius (`r_vol_um`) relative to `radius_threshold_um`.

For example:

- if `r_vol_um` for a waypoint < 5 um, the mix of ice particle habits will be 100% droxtals.
- if `r_vol_um` for a waypoint between 5 and 9.5 um, the mix of ice particle habits will be 30% solid columns, 70% droxtals.

See Table 2 from [Schumann *et al.*, 2011].

References

- [Schumann *et al.*, 2011]
- [Schumann *et al.*, 2012]

A_mu = array([0.361226, 0.294072, 0.343894, 0.317866, 0.337227, 0.310978, 0.342593, 0.269179])

A_μ in Eq. (6) in [Schumann *et al.*, 2012]

B_mu = array([1.67592, 1.55687, 1.71065, 1.55843, 1.70782, 1.71789, 1.56399, 1.59015])

Approximate the SZA-dependent contrail sideward scattering B_μ in Eq. (10) in [Schumann *et al.*, 2012]

C_mu = array([0.7093 , 0.678016, 0.687546, 0.675315, 0.712041, 0.713317, 0.660267, 0.545716])

C_μ in Eq. (6) in [Schumann *et al.*, 2012]

F_r = array([0.511852, 0.576911, 0.597351, 0.22575 , 0.550734, 0.817858, 0.249004, 0.])

Approximates the effective contrail optical depth F_r in Eq. (7) and (8) in [Schumann *et al.*, 2012]

T_0 = array([152.237, 152.724, 152.923, 152.36 , 151.879, 152.318, 165.692, 153.073])

Approximates the temperature of the atmosphere without contrails T_0 in Eq. (2) in [Schumann *et al.*, 2012]

delta_lc = array([0.159942 , 0.0958129, 0.092485 , 0.0462023, 0.132925 , 0.0870067, 0.0626339, 0.0665289])

Optical depth scaling factor for reduction of the OLR at the contrail level due to existing cirrus above the contrail $\delta_{lc} in Eq.(4) in : cite : `schumann Parametric Radiative Forcing 2012$

delta_lr = array([0.211276, 0.341194, 0.325496, 0.255921, 0.170265, 1.65441 , 0.201949, 0.])

Effective radius scaling factor for optical properties (extinction relative to scattering) $\delta_{lr} in Eq.(3) in : cite : `schumannParametricRadiativeForcing2012$

delta_sc = array([0.157017, 0.143274, 0.167995, 0.148547, 0.173036, 0.162442, 0.171855, 0.213488])

Account for the optical depth of natural cirrus above the contrail δ_{sc} in Eq. (11) in [Schumann *et al.*, 2012]

delta_sc_aps = array([0.229574, 0.197611, 0.245036, 0.204875, 0.248328, 0.254029, 0.244051, 0.302246])

delta_sr = array([0.149851 , 0.025427 , 0.0238836, 0.0463724, 0.0478892, 0.0700234, 0.0517942, 0.])

Approximates the effective contrail optical depth $delta_s r$ in Eq. (7) and (8) in [Schumann *et al.*, 2012]

delta_t = array([0.940846, 0.808397, 0.736222, 0.675591, 0.748757, 0.708515, 0.927592, 0.795527])

Approximate the effective emmissivity factor $\delta_\tau in : cite : `schumannParametricRadiativeForcing2012$

gamma_lower = array([0.323166, 0.392598, 0.356189, 0.34504 , 0.407515, 0.523604, 0.310853, 0.274741])

Approximates the contrail reflectances γ in Eq. (9) in [Schumann *et al.*, 2012]

gamma_upper = array([0.241507, 0.347023, 0.288452, 0.296813, 0.327857, 0.43756 , 0.27471 , 0.208154])

Approximates the contrail reflectances Γ in Eq. (9) in [Schumann *et al.*, 2012]

k_t = array([1.93466, 1.95456, 1.95994, 1.95906, 1.94397, 1.95123, 2.30363, 1.94611])

Linear approximation of Stefan-Boltzmann Law k_t in Eq. (2) in [Schumann *et al.*, 2012]

t_a = array([0.879119, 0.901701, 0.881812, 0.899144, 0.879896, 0.883212, 0.899096, 1.00744])

Approximates the dependence on the effective albedo t_a: Eq. (5) in [Schumann *et al.*, 2012]

pycontrails.models.cocip.radiative_forcing.**albedo**(*sdr*, *rsr*)

Calculate albedo along contrail waypoint.

Albedo, the diffuse reflection of solar radiation out of the total solar radiation, is computed based on the solar direct radiation (*sdr*) and reflected solar radiation (*rsr*).

Output values range between 0 (corresponding to a black body that absorbs all incident radiation) and 1 (a body that reflects all incident radiation).

Parameters

- **sdr** (`npt.NDArray[np.float64]`) – Solar direct radiation, $[Wm^{-2}]$
- **rsr** (`npt.NDArray[np.float64]`) – Reflected solar radiation, $[Wm^{-2}]$

Returns

`npt.NDArray[np.float64]` – Albedo value, $[[0 - 1]]$

pycontrails.models.cocip.radiative_forcing.**contrail_albedo**(*tau_contrail*, *mue*, *r_eff_um*, *A_mu*, *B_mu*, *C_mu*, *delta_sr*, *F_r*, *gamma_lower*, *gamma_upper*)

Calculate the contrail albedo, `alpha_c`.

Refer to Eq. (6) of Schumann et al. (2012),

Parameters

- **tau_contrail** (`npt.NDArray[np.float64]`) – Contrail optical depth for each waypoint
- **mue** (`npt.NDArray[np.float64]`) – Cosine of the solar zenith angle (theta), mue = cos(theta) = sdr/sd0
- **r_eff_um** (`npt.NDArray[np.float64]`) – Effective radius for each waypoint, n_waypoints x 8 (habit) columns, [μm] See `effective_radius_habit()`.
- **A_mu** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the albedo of the contrail
- **B_mu** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the SZA-dependent contrail sideward scattering
- **C_mu** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the albedo of the contrail
- **delta_sr** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the effective contrail optical depth
- **F_r** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the effective contrail optical depth
- **gamma_lower** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the contrail reflectances
- **gamma_upper** (`npt.NDArray[np.float64]`) – Habit-specific parameter to approximate the contrail reflectances

Returns

`npt.NDArray[np.float64]` – Contrail albedo for each waypoint and ice particle habit

pycontrails.models.cocip.radiative_forcing.**contrail_contrail_overlap_radiative_effects**(*contrails*, *habit_distributions*, *radius_threshold_um*, ***, *min_altitude_m=600*, *max_altitude_m=130*, *dz_overlap_m=500.0*, *spatial_grid_res=0.25*)

Calculate radiative properties after accounting for contrail overlapping.

This function mutates the `contrails` parameter.

Parameters

- **contrails** (*GeoVectorDataset*) – Contrail waypoints at a given time. Must include the following variables: - segment_length - width - r_ice_vol - tau_contrail - tau_cirrus - air_temperature - sdr - rsr - olr
- **habit_distributions** (`npt.NDArray[np.float64]`) – Habit weight distributions. See `CocipParams.habit_distributions`

- **radius_threshold_um** (npt.NDArray[np.float64]) – Radius thresholds for habit distributions. See CocipParams.radius_threshold_um
- **min_altitude_m** (float) – Minimum altitude domain in simulation, [m] See CocipParams.min_altitude_m
- **max_altitude_m** – Maximum altitude domain in simulation, [m] See CocipParams.min_altitude_m
- **dz_overlap_m** (float) – Altitude interval used to segment contrail waypoints, [m] See CocipParams.dz_overlap_m
- **spatial_grid_res** (float) – Spatial grid resolution, [deg]

Returns

GeoVectorDataset – Contrail waypoints at a given time with additional variables attached, including - rsr_overlap - olr_overlap - tau_cirrus_overlap - rf_sw_overlap - rf_lw_overlap - rf_net_overlap

References

- **Schumann et al. (2021) Air traffic and contrail changes over Europe during COVID-19:** A model study, Atmos. Chem. Phys., 21, 7429–7450, https://doi.org/10.5194/ACP-21-7429-2021.
- Teoh et al. (2023) Global aviation contrail climate effects from 2019 to 2021.

Notes

- The radiative effects of contrail-contrail overlapping is approximated by changing the background RSR and OLR fields, and the overlying cirrus optical depth above the contrail.
- All contrail segments within each altitude interval are treated as one contrail layer, where they do not overlap. Contrail layers are processed starting from the bottom to the top.
- Refer to the Supporting Information (S4.3) of Teoh et al. (2023)

pycontrails.models.cocip.radiative_forcing.**contrail_effective_emissivity**(*r_eff_um*, *delta_lr*)

Calculate the effective emissivity of the contrail, f_lw.

Refer to Eq. (3) of Schumann et al. (2012).

Parameters

- **r_eff_um** (npt.NDArray[np.float64]) – Effective radius for each waypoint, n_waypoints x 8 (habit) columns, [μm] See effective_radius_habit().
- **delta_lr** (npt.NDArray[np.float64]) – Habit specific parameter to approximate the effective emissivity of the contrail.

Returns

npt.NDArray[np.float64] – Effective emissivity of the contrail

pycontrails.models.cocip.radiative_forcing.**effective_radius_by_habit**(*r_vol_um*, *habit_idx*)

Calculate the effective radius r_eff_um via the mean ice particle radius and habit type.

The habit_idx corresponds to the habit types in rf_const.habits. Each habit type has a specific parameterization to calculate r_eff_um based on r_vol_um. derived from [Schumann *et al.*, 2011].

Parameters

- **r_vol_um** (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]

- **habit_idx** (npt.NDArray[np.intp]) – Habit type index for the contrail ice particle, corresponding to the habits in rf_const.habits.

Returns
npt.NDArray[np.float64] – Effective radius of ice particles for each combination of r_vol_um and habit_idx, [μm]

References

- [Schumann *et al.*, 2011]

pycontrails.models.cocip.radiative_forcing.**effective_radius_droxtal**(*r_vol_um*)

Calculate the effective radius of contrail ice particles assuming a droxtal particle habit.

Parameters
r_vol_um (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]

Returns
npt.NDArray[np.float64] – Effective radius, [μm]

pycontrails.models.cocip.radiative_forcing.**effective_radius_hollow_column**(*r_vol_um*)

Calculate the effective radius of ice particles assuming a hollow column particle habit.

Parameters
r_vol_um (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]

Returns
npt.NDArray[np.float64] – Effective radius, [μm]

pycontrails.models.cocip.radiative_forcing.**effective_radius_myhre**(*r_vol_um*)

Calculate the effective radius of contrail ice particles assuming a sphere particle habit.

Parameters
r_vol_um (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]

Returns
npt.NDArray[np.float64] – Effective radius, [μm]

pycontrails.models.cocip.radiative_forcing.**effective_radius_plate**(*r_vol_um*)

Calculate the effective radius of contrail ice particles assuming a plate particle habit.

Parameters
r_vol_um (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]

Returns
npt.NDArray[np.float64] – Effective radius, [μm]

pycontrails.models.cocip.radiative_forcing.**effective_radius_rosette**(*r_vol_um*)

Calculate the effective radius of contrail ice particles assuming a rosette particle habit.

Parameters
r_vol_um (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]

Returns
npt.NDArray[np.float64] – Effective radius, [μm]

pycontrails.models.cocip.radiative_forcing.**effective_radius_rough_aggregate**(*r_vol_um*)

Calculate the effective radius of ice particles assuming a rough aggregate particle habit.

Parameters
r_vol_um (`npt.NDArray[np.float64]`) – Contrail ice particle volume mean radius, [μm]

Returns
`npt.NDArray[np.float64]` – Effective radius, [μm]

`pycontrails.models.cocip.radiative_forcing.`**`effective_radius_solid_column`**(*r_vol_um*)

Calculate the effective radius of contrail ice particles assuming a solid column particle habit.

Parameters
r_vol_um (`npt.NDArray[np.float64]`) – Contrail ice particle volume mean radius, [μm]

Returns
`npt.NDArray[np.float64]` – Effective radius, [μm]

`pycontrails.models.cocip.radiative_forcing.`**`effective_radius_sphere`**(*r_vol_um*)

Calculate the effective radius of contrail ice particles assuming a sphere particle habit.

Parameters
r_vol_um (`npt.NDArray[np.float64]`) – Contrail ice particle volume mean radius, [μm]

Returns
`npt.NDArray[np.float64]` – Effective radius, [μm]

`pycontrails.models.cocip.radiative_forcing.`**`effective_tau_cirrus`**(*tau_cirrus*, *mue*, *delta_sc*, *delta_sc_aps*)

Calculate the effective optical depth of natural cirrus above the contrail, `e_sw`.

Refer to Eq. (11) of [Schumann *et al.*, 2012]. See Notes for a correction to the equation.

Parameters

- **tau_cirrus** (`npt.NDArray[np.float64]`) – Optical depth of numerical weather prediction (NWP) cirrus above the contrail for each waypoint.
- **mue** (`npt.NDArray[np.float64]`) – Cosine of the solar zenith angle (theta), mue = cos(theta) = sdr/sd0
- **delta_sc** (`npt.NDArray[np.float64]`) – Habit-specific parameter to account for the optical depth of natural cirrus above the contrail
- **delta_sc_aps** (`npt.NDArray[np.float64]`) – Habit-specific parameter to account for the optical depth of natural cirrus above the contrail

Returns
`npt.NDArray[np.float64]` – Effective optical depth of natural cirrus above the contrail, `n_waypoints x 8` (habit) columns.

Notes

- In a personal correspondence, Dr. Schumann identified a print error in Eq. (11) in [Schumann *et al.*, 2012], where the positions of `delta_sc_aps` and `delta_sc` should be swapped. The correct function is provided below.

`pycontrails.models.cocip.radiative_forcing.`**`habit_weight_regime_idx`**(*r_vol_um*, *radius_threshold_um*)

Determine regime of ice particle habits based on contrail ice particle volume mean radius.

Parameters

- **r_vol_um** (`npt.NDArray[np.float64]`) – Contrail ice particle volume mean radius, [μm]

- **radius_threshold_um** (npt.NDArray[np.float64]) – Radius thresholds for habit distributions. See CocipParams.radius_threshold_um

Returns

npt.NDArray[np.intp] – Row index of the habit distribution in array CocipParams().habit_distributions

pycontrails.models.cocip.radiative_forcing.**habit_weights**(*r_vol_um*, *habit_distributions*, *radius_threshold_um*)

Assign weights to different ice particle habits for each waypoint.

For each waypoint, the distinct mix of ice particle habits are approximated using the mean contrail ice particle radius (r_vol_um) binned by radius_threshold_um.

For example:

- For waypoints with r_vol_um < 5 um, the mix of ice particle habits will be from Group 1 (100% Droxtals, refer to CocipParams().habit_distributions).
- For waypoints with 5 um <= r_vol_um < 9.5 um, the mix of ice particle habits will be from Group 2 (30% solid columns, 70% droxtals)

Parameters

- **r_vol_um** (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]
- **habit_distributions** (npt.NDArray[np.float64]) – Habit weight distributions. See CocipParams().habit_distributions
- **radius_threshold_um** (npt.NDArray[np.float64]) – Radius thresholds for habit distributions. See CocipParams.radius_threshold_um

Returns

npt.NDArray[np.float64] – Array with shape n_waypoints x 8 columns, where each column is the weights to the ice particle habits, [$[0 - 1]$], and the sum of each column should be equal to 1.

Raises

ValueError – Raises when habit_distributions do not sum to 1 across columns or if there is a size mismatch with radius_threshold_um.

pycontrails.models.cocip.radiative_forcing.**longwave_radiative_forcing**(*r_vol_um*, *olr*, *air_temperature*, *tau_contrail*, *tau_cirrus*, *habit_weights_*, *r_eff_um=None*)

Calculate the local contrail longwave radiative forcing (RF_{LW}).

All returned values are positive.

Parameters

- **r_vol_um** (npt.NDArray[np.float64]) – Contrail ice particle volume mean radius, [μm]
- **olr** (npt.NDArray[np.float64]) – Outgoing longwave radiation at each waypoint, [Wm^{-2}]
- **air_temperature** (npt.NDArray[np.float64]) – Ambient temperature at each waypoint, [K]
- **tau_contrail** (npt.NDArray[np.float64]) – Contrail optical depth at each waypoint

- **tau_cirrus** (`npt.NDArray[np.float64]`) – Optical depth of numerical weather prediction (NWP) cirrus above the contrail at each waypoint
- **habit_weights_** (`npt.NDArray[np.float64]`) – Weights to different ice particle habits for each waypoint, `n_waypoints x 8` (habit) columns, $[[0 - 1]]$
- **r_eff_um** (`npt.NDArray[np.float64]`, *optional*) – Provide effective radius corresponding to elements in `r_vol_um`, $[\mu m]$. Defaults to None, which means the effective radius will be calculated using `r_vol_um` and habit types in *effective_radius_by_habit()*.

Returns
`npt.NDArray[np.float64]` – Local contrail longwave radiative forcing (positive), $[Wm^{-2}]$

Raises
`ValueError` – If *r_eff_um* and *olr* have different shapes.

References

- [Schumann *et al.*, 2012]

pycontrails.models.cocip.radiative_forcing.**net_radiative_forcing**(*rf_lw*, *rf_sw*)

Calculate the local contrail net radiative forcing (rf_net).

RF Net = Longwave RF (positive) + Shortwave RF (negative)

Parameters
- **rf_lw** (`npt.NDArray[np.float64]`) – local contrail longwave radiative forcing, $[Wm^{-2}]$
- **rf_sw** (`npt.NDArray[np.float64]`) – local contrail shortwave radiative forcing, $[Wm^{-2}]$

Returns
`npt.NDArray[np.float64]` – local contrail net radiative forcing, $[Wm^{-2}]$

pycontrails.models.cocip.radiative_forcing.**olr_reduction_natural_cirrus**(*tau_cirrus*, *delta_lc*)

Calculate reduction in outgoing longwave radiation (OLR) due to the presence of natural cirrus.

Natural cirrus has optical depth `tau_cirrus` above the contrail. See `e_lw` in Eq. (4) of Schumann et al. (2012).

Parameters
- **tau_cirrus** (`npt.NDArray[np.float64]`) – Optical depth of numerical weather prediction (NWP) cirrus above the contrail for each waypoint.
- **delta_lc** (`npt.NDArray[np.float64]`) – Habit specific parameter to approximate the reduction of the outgoing longwave radiation at the contrail level due to natural cirrus above the contrail.

Returns
`npt.NDArray[np.float64]` – Reduction of outgoing longwave radiation

pycontrails.models.cocip.radiative_forcing.**shortwave_radiative_forcing**(*r_vol_um*, *sdr*, *rsr*, *sd0*, *tau_contrail*, *tau_cirrus*, *habit_weights_*, *r_eff_um=None*)

Calculate the local contrail shortwave radiative forcing (RF_{SW}).

All returned values are negative.

Parameters

- **r_vol_um** (`npt.NDArray[np.float64]`) – Contrail ice particle volume mean radius, [μm]
- **sdr** (`npt.NDArray[np.float64]`) – Solar direct radiation, [Wm^{-2}]
- **rsr** (`npt.NDArray[np.float64]`) – Reflected solar radiation, [Wm^{-2}]
- **sd0** (`npt.NDArray[np.float64]`) – Solar constant, [Wm^{-2}]
- **tau_contrail** (`npt.NDArray[np.float64]`) – Contrail optical depth for each waypoint
- **tau_cirrus** (`npt.NDArray[np.float64]`) – Optical depth of numerical weather prediction (NWP) cirrus above the contrail for each waypoint.
- **habit_weights_** (`npt.NDArray[np.float64]`) – Weights to different ice particle habits for each waypoint, `n_waypoints x 8` (habit) columns, [$[0 - 1]$]
- **r_eff_um** (`npt.NDArray[np.float64]`, *optional*) – Provide effective radius corresponding to elements in `r_vol_um`, [μm]. Defaults to None, which means the effective radius will be calculated using `r_vol_um` and habit types in *`effective_radius_by_habit()`*.

Returns

`npt.NDArray[np.float64]` – Local contrail shortwave radiative forcing (negative), [Wm^{-2}]

Raises

`ValueError` – If *r_eff_um* and *sdr* have different shapes.

References

- [Schumann *et al.*, 2012]

pycontrails.models.cocip.wake_vortex

Wave-vortex downwash functions.

This module includes equations from the original CoCiP model [Schumann, 2012]. An alternative set of equations based on [Unterstrasser, 2016] is available in `unterstrasser_wake_vortex`.

Unterstrasser Notes

Improved estimation of the survival fraction of the contrail ice crystal number `f_surv` during the wake-vortex phase. This is a parameterised model that is developed based on outputs provided by large eddy simulations.

For comparison, CoCiP assumes that `f_surv` is equal to the change in the contrail ice water content (by mass) before and after the wake vortex phase. However, for larger (smaller) ice particles, their survival fraction by number could be smaller (larger) than their survival fraction by mass. This is particularly important in the "soot-poor" scenario, for example, in cleaner lean-burn engines where their soot emissions can be 3-4 orders of magnitude lower than conventional RQL engines.

Functions

downward_displacement_strongly_stratified(...)	Calculate the maximum contrail downward displacement under strongly stratified conditions.
downward_displacement_weakly_stratified(...)	Calculate the maximum contrail downward displacement under weakly/stably stratified conditions.
effective_time_scale(wingspan, ...)	Calculate the effective time scale of the wake vortex.
initial_contrail_depth(dz_max, ...)	Calculate the initial contrail depth.
initial_contrail_width(wingspan, dz_max)	Calculate the initial contrail width.
max_downward_displacement(wingspan, ...)	Calculate the maximum contrail downward displacement after the wake vortex phase.
normalized_dissipation_rate(epsilon, ...)	Calculate the normalized dissipation rate of the sinking wake vortex.
turbulent_kinetic_energy_dissipation_rate(ds	Calculate the turbulent kinetic energy dissipation rate (epsilon).
wake_vortex_separation(wingspan)	Calculate the wake vortex separation.

pycontrails.models.cocip.wake_vortex.**downward_displacement_strongly_stratified**(*wingspan*, *true_airspeed*, *aircraft_mass*, *rho_air*, *n_bv*)

Calculate the maximum contrail downward displacement under strongly stratified conditions.

Parameters

- **wingspan** (`npt.NDArray[np.float64] | float`) – aircraft wingspan, [m]
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, [ms^{-1}]
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – aircraft mass for each waypoint, [kg]
- **rho_air** (`npt.NDArray[np.float64]`) – density of air for each waypoint, [kgm^{-3}]
- **n_bv** (`npt.NDArray[np.float64]`) – Brunt-Vaisaila frequency, [s^{-1}]

Returns

`npt.NDArray[np.float64]` – Maximum contrail downward displacement, strongly stratified conditions, [m]

Notes

See section 2.5 (pg 547 - 548) of [Schumann, 2012].

References

- [Schumann, 2012]

pycontrails.models.cocip.wake_vortex.**downward_displacement_weakly_stratified**(*wingspan*, *true_airspeed*, *aircraft_mass*, *rho_air*, *n_bv*, *dz_max_strong*, *ds_dz*, *t_0*, *effective_vertical_resolution*, *wind_shear_enhancement_expone*

Calculate the maximum contrail downward displacement under weakly/stably stratified conditions.

Parameters

- **wingspan** (`npt.NDArray[np.float64] | float`) – aircraft wingspan, [m]
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, [ms^{-1}]
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – aircraft mass for each waypoint, [kg]
- **rho_air** (`npt.NDArray[np.float64]`) – density of air for each waypoint, [kgm^{-3}]
- **n_bv** (`npt.NDArray[np.float64]`) – Brunt-Vaisaila frequency, [s^{-1}]
- **dz_max_strong** (`npt.NDArray[np.float64]`) – Max contrail downward displacement under strongly stratified conditions, [m]
- **ds_dz** (`npt.NDArray[np.float64]`) – Difference in wind speed over dz in the atmosphere, [ms^{-1}/m]
- **t_0** (`npt.NDArray[np.float64]`) – Wake vortex effective time scale, [s]
- **effective_vertical_resolution** (`float`) – Passed through to `wind_shear.wind_shear_enhancement_factor()`, [m]
- **wind_shear_enhancement_exponent** (`npt.NDArray[np.float64] | float`) – Passed through to `wind_shear.wind_shear_enhancement_factor()`

Returns

`npt.NDArray[np.float64]` – Maximum contrail downward displacement, weakly/stably stratified conditions, [m]

Notes

See section 2.5 (pg 548) of [Schumann, 2012].

References

- [Schumann, 2012]

pycontrails.models.cocip.wake_vortex.**effective_time_scale**(*wingspan*, *true_airspeed*, *aircraft_mass*, *rho_air*)

Calculate the effective time scale of the wake vortex.

Parameters

- **wingspan** (`npt.NDArray[np.float64]`) – aircraft wingspan, [m]
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, [$m\ s^{-1}$]
- **aircraft_mass** (`npt.NDArray[np.float64]`) – aircraft mass for each waypoint, [kg]
- **rho_air** (`npt.NDArray[np.float64]`) – density of air for each waypoint, [$kg\ m^{-3}$]

Returns

`npt.NDArray[np.float64]` – Wake vortex effective time scale, [s]

Notes

See section 2.5 (pg 547) of [Schumann, 2012].

References

- [Schumann, 2012]

pycontrails.models.cocip.wake_vortex.**initial_contrail_depth**(*dz_max*, *initial_wake_vortex_depth*)

Calculate the initial contrail depth.

Parameters

- **dz_max** (`npt.NDArray[np.float64]`) – Max contrail downward displacement after the wake vortex phase, [m]
- **initial_wake_vortex_depth** (`float | npt.NDArray[np.float64]`) – Initial wake vortex depth scaling factor. Denoted *C_D0* in eq (14) in [Schumann, 2012].

Returns

`npt.NDArray[np.float64]` – Initial contrail depth, [m]

pycontrails.models.cocip.wake_vortex.**initial_contrail_width**(*wingspan*, *dz_max*)

Calculate the initial contrail width.

Parameters

- **wingspan** (`npt.NDArray[np.float64] | float`) – aircraft wingspan, [m]
- **dz_max** (`npt.NDArray[np.float64]`) – Max contrail downward displacement after the wake vortex phase, [m] Only the size of this array is used; the values are ignored.

Returns

`npt.NDArray[np.float64]` – Initial contrail width, [m]

pycontrails.models.cocip.wake_vortex.**max_downward_displacement**(*wingspan*, *true_airspeed*, *aircraft_mass*, *air_temperature*, *dT_dz*, *ds_dz*, *air_pressure*, *effective_vertical_resolution*, *wind_shear_enhancement_exponent*)

Calculate the maximum contrail downward displacement after the wake vortex phase.

Parameters

- **wingspan** (`npt.NDArray[np.float64] | float`) – aircraft wingspan, $[m]$
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, $[ms^{-1}]$
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – aircraft mass for each waypoint, $[kg]$
- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature for each waypoint, $[K]$
- **dT_dz** (`npt.NDArray[np.float64]`) – potential temperature gradient, $[Km^{-1}]$
- **ds_dz** (`npt.NDArray[np.float64]`) – Difference in wind speed over dz in the atmosphere, $[ms^{-1}/m]$
- **air_pressure** (`npt.NDArray[np.float64]`) – pressure altitude at each waypoint, $[Pa]$
- **effective_vertical_resolution** (`float`) – Passed through to `wind_shear.wind_shear_enhancement_factor()`, $[m]$
- **wind_shear_enhancement_exponent** (`npt.NDArray[np.float64] | float`) – Passed through to `wind_shear.wind_shear_enhancement_factor()`

Returns

`npt.NDArray[np.float64]` – Max contrail downward displacement after the wake vortex phase, $[m]$

References

- [Holzäpfel, 2003]
- [Schumann, 2012]

pycontrails.models.cocip.wake_vortex.**normalized_dissipation_rate**(*epsilon*, *wingspan*, *true_airspeed*, *aircraft_mass*, *rho_air*)

Calculate the normalized dissipation rate of the sinking wake vortex.

Parameters

- **epsilon** (`npt.NDArray[np.float64]`) – turbulent kinetic energy dissipation rate, $[m^2s^{-3}]$
- **wingspan** (`npt.NDArray[np.float64] | float`) – aircraft wingspan, $[m]$
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, $[ms^{-1}]$
- **aircraft_mass** (`npt.NDArray[np.float64]`) – aircraft mass for each waypoint, $[kg]$
- **rho_air** (`npt.NDArray[np.float64]`) – density of air for each waypoint, $[kgm^{-3}]$

Returns

`npt.NDArray[np.float64]` – Normalized dissipation rate of the sinking wake vortex

Notes

See page 548 of [Schumann, 2012].

References

- [Schumann, 2012]

pycontrails.models.cocip.wake_vortex.**turbulent_kinetic_energy_dissipation_rate**(*ds_dz*, *shear_enhancement_factor=1.0*

Calculate the turbulent kinetic energy dissipation rate (epsilon).

The shear enhancement factor is used to account for any sub-grid scale turbulence.

Parameters

- **ds_dz** (`npt.NDArray[np.float64]`) – Difference in wind speed over dz in the atmosphere, $[ms^{-1}/m]$
- **shear_enhancement_factor** (`npt.NDArray[np.float64] | float`) – Multiplication factor to enhance the wind shear

Returns

`npt.NDArray[np.float64]` – turbulent kinetic energy dissipation rate, $[m^2s^{-3}]$

Notes

- See eq. (37) in [Schumann, 2012].
- In a personal correspondence, Dr. Schumann identified a print error in Eq. (37) of the 2012 paper where the shear term should not be squared. The correct equation is listed in Eq. (13) [Schumann and Gerz, 1995].

References

- [Schumann, 2012]
- [Schumann and Gerz, 1995]

pycontrails.models.cocip.wake_vortex.**wake_vortex_separation**(*wingspan*)

Calculate the wake vortex separation.

Parameters

wingspan (`npt.NDArray[np.float64] | float`) – aircraft wingspan, $[m]$

Returns

`npt.NDArray[np.float64]` – wake vortex separation, $[m]$

pycontrails.models.cocip.wind_shear

Wind shear functions.

Functions

wind_shear(u_wind_top, u_wind_btm, ...)	Calculate the total wind shear.
wind_shear_enhancement_factor(...)	Calculate the multiplication factor to enhance the wind shear based on contrail depth.
wind_shear_normal(u_wind_top, u_wind_btm, ...)	Calculate the total wind shear normal to an axis.

pycontrails.models.cocip.wind_shear.**wind_shear**(*u_wind_top*, *u_wind_btm*, *v_wind_top*, *v_wind_btm*, *dz*)

Calculate the total wind shear.

The total wind shear is the vertical gradient of the horizontal velocity.

Parameters

- **u_wind_top** (*ArrayScalarLike*) – u wind speed in the top layer, $[m\ s^{-1}]$
- **u_wind_btm** (*ArrayScalarLike*) – u wind speed in the bottom layer, $[m\ s^{-1}]$
- **v_wind_top** (*ArrayScalarLike*) – v wind speed in the top layer, $[m\ s^{-1}]$
- **v_wind_btm** (*ArrayScalarLike*) – v wind speed in the bottom layer, $[m\ s^{-1}]$
- **dz** (`float`) – Difference in altitude between measurements, $[m]$

Returns

ArrayScalarLike – Total wind shear, $[s^{-1}]$

pycontrails.models.cocip.wind_shear.**wind_shear_enhancement_factor**(*contrail_depth*, *effective_vertical_resolution*, *wind_shear_enhancement_exponent*)

Calculate the multiplication factor to enhance the wind shear based on contrail depth.

This factor accounts for any subgrid-scale that is not captured by the resolution of the meteorological datasets.

Parameters

- **contrail_depth** (`npt.NDArray[np.float64]`) – Contrail depth , $[m]$. Expected to be positive and bounded away from 0.
- **effective_vertical_resolution** (`float | npt.NDArray[np.float64]`) – Vertical resolution of met data , $[m]$
- **wind_shear_enhancement_exponent** (`float | npt.NDArray[np.float64]`) – Exponent used in calculation. Expected to be nonnegative. Discussed in paragraphs following eq. (39) in Schumann 2012 and referenced as *n*. When this parameter is 0, no enhancement occurs.

Returns

`npt.NDArray[np.float64]` – Wind shear enhancement factor

Notes

Implementation based on eq (39) in [Schumann, 2012].

References

- [Schumann, 2012]

pycontrails.models.cocip.wind_shear.**wind_shear_normal**(*u_wind_top*, *u_wind_btm*, *v_wind_top*, *v_wind_btm*, *cos_a*, *sin_a*, *dz*)

Calculate the total wind shear normal to an axis.

The total wind shear is the vertical gradient of the horizontal velocity.

Parameters

- **u_wind_top** (*ArrayScalarLike*) – u wind speed in the top layer, [$m\ s^{-1}$]
- **u_wind_btm** (*ArrayScalarLike*) – u wind speed in the bottom layer, [$m\ s^{-1}$]
- **v_wind_top** (*ArrayScalarLike*) – v wind speed in the top layer, [$m\ s^{-1}$]
- **v_wind_btm** (*ArrayScalarLike*) – v wind speed in the bottom layer, [$m\ s^{-1}$]
- **cos_a** (*ArrayScalarLike*) – Cosine component of segment
- **sin_a** (*ArrayScalarLike*) – Sine component of segment
- **dz** (`float`) – Difference in altitude between measurements, [m]

Returns

ArrayScalarLike – Wind shear normal to axis, [s^{-1}]

11.3.4 Gridded CoCiP

models.cocipgrid.CocipGrid(met, rad[, params])	Run CoCiP simulation on a grid.
models.cocipgrid.CocipGridParams([...])	Default parameters for `CocipGrid`.

pycontrails.models.cocipgrid.CocipGrid

class pycontrails.models.cocipgrid.**CocipGrid**(*met*, *rad*, *params=None*, ***params_kwargs*)

Bases: *Model*

Run CoCiP simulation on a grid.

See *eval()* for a description of model evaluation `source` parameters.

Parameters

- **met, rad** (*MetDataset*) – CoCiP-specific met data to interpolate against
- **params** (`dict[str, Any]`, *optional*) – Override *CocipGridParams* defaults. Most notably, the model is highly dependent on the parameter `dt_integration`. Memory usage is also affected by parameters `met_slice_dt` and `target_split_size`.
- **param_kwargs** (`Any`) – Override CocipGridParams defaults with arbitrary keyword arguments.

Notes

- If `rad` contains accumulated radiative fluxes, differencing to obtain time-averaged fluxes will reduce the time coverage of `rad` by half a forecast step. A warning will be produced during *eval()* if the time coverage of `rad` (after differencing) is too short given the model evaluation parameters. If this occurs, provide an additional step of radiation data at the start or end of `rad`.

References

- [Schumann *et al.*, 2011]
- [Schumann *et al.*, 2012]

See also:

CocipGridParams, `Cocip`, `wake_vortex`, `contrail_properties`, `radiative_forcing`, `humidity_scaling`, `Emissions`, `sac`, `tau_cirrus`

__init__(*met*, *rad*, *params=None*, ***params_kwargs*)

Methods

__init__(met, rad[, params])	
create_source(level, time[, longitude, ...])	Shortcut to create a `MetDataset` source from coordinate arrays.
`downselect_met`()	Downselect `met` domain to the max/min bounds of `source`.
eval([source])	Run CoCiP simulation on a 4d coordinate grid or arbitrary set of 4d points.
`get_source_param`(key[, default, set_attr])	Get source data with default set by parameter key.
`require_met`()	Ensure that `met` is a MetDataset.
`require_source_type`(type_)	Ensure that `source` is `type_`.
`set_source`([source])	Attach original or copy of input `source` to `source`.
`set_source_met`([optional, variable])	Ensure or interpolate each required *met_variables* on `source` .
`transfer_met_source_attrs`([source])	Transfer met source metadata from `met` to `source`.
`update_params`([params])	Update model parameters on `params`.

Attributes

rad	
timesteps	
contrail	
contrail_list	Artifacts attached when parameter `verbose_outputs_evolution` is True These allow for some additional information and parity with the approach taken by `Cocip`.
`hash`	Generate a unique hash for model instance.
`interp_kwargs`	Shortcut to create interpolation arguments from `params`.
long_name	
`met`	Met data is not optional
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
`params`	Instantiated model parameters, in dictionary form
processed_met_variables	Set of required parameters if processing already complete on `met` input.
rad_variables	
`source`	Last evaluated input source
source_time	Return the time array of the `source` data.
`optional_met_variables`	Optional meteorology variables

contrail

contrail_list

Artifacts attached when parameter `verbose_outputs_evolution` is True These allow for some additional information and parity with the approach taken by `Cocip`.

static create_source(*level*, *time*, *longitude=None*, *latitude=None*, *lon_step=1.0*, *lat_step=1.0*)

Shortcut to create a `MetDataset` source from coordinate arrays.

Parameters

- **level** (`level`: `npt.NDArray[np.float64] | list[float] | float`) – Pressure levels for gridded cocip. To avoid interpolating outside of the passed `met` and `rad` data, this parameter should avoid the extreme values of the `met` and *rad* levels. If `met` is already defined, a good choice for `level` is `met.data['level'].values[1: -1]`.
- **time** (`npt.NDArray[np.datetime64 | list[np.datetime64] | np.datetime64,`) – One or more time values for gridded cocip.
- **longitude, latitude** (`npt.NDArray[np.float64] | list[float]`, *optional*) – Longitude and latitude arrays, by default None. If not specified, values of `lon_step` and `lat_step` are used to define `longitude` and `latitude`. To avoid model degradation at the poles, latitude values are expected to be between -80 and 80 degrees.

- **lon_step, lat_step** (float, *optional*) – Longitude and latitude resolution, by default 1.0. Only used if parameter `longitude` (respective `latitude`) not specified.

Returns

MetDataset – MetDataset that can be used as `source` input to `CocipGrid.eval(source=..)()`

See also:

`MetDataset.from_coords()`

default_params

alias of *CocipGridParams*

eval(*source=None*, ***params*)

Run CoCiP simulation on a 4d coordinate grid or arbitrary set of 4d points.

If the `params` `verbose_outputs_evolution` is True, the model holds *contrail_list* and *contrail* attributes for viewing intermediate artifacts. If `source` data is large, these intermediate vectors may consume substantial memory.

Changed in version 0.25.0: No longer explicitly support `Flight` as a source. Any flight source will be viewed as a `GeoVectorDataset`. In order to evaluate CoCiP predictions over a flight trajectory, it is best to use the `Cocip` model. It's also possible to pre-compute segment azimuth and true airspeed before passing the flight trajectory in here.

Parameters

- **source** (`GeoVectorDataset | MetDataset | None`) – Input `GeoVectorDataset` or `MetDataset`. If None, a `NotImplementedError` is raised. If any subclass of `GeoVectorDataset` is passed (e.g., `Flight`), the additional structure is forgotten and the model is evaluated as if it were a `GeoVectorDataset`. Additional variables may be passed as `source` data or attrs. These include:
  - `aircraft_type`: This overrides any value in `params`. Must be included in the source attrs (not data).
  - `fuel_flow`, `engine_efficiency`, `true_airspeed`, `wingspan`, `aircraft_mass`: These override any value in `params`.
  - `azimuth`: This overrides any value in `params`.
  - `segment_length`: This overrides any value in `params`.
- ****params** (`Any`) – Overwrite model parameters before eval

Returns

`GeoVectorDataset | MetDataset` – CoCiP predictions for each point in `source`. Output data contains variables `contrail_age` and `ef_per_m`. Additional variables specified by the model `params` `verbose_outputs_formation` are also included.

Raises

`NotImplementedError` – If `source` is None

Notes

At a high level, the model is broken down into the following steps:

- Convert any `MetDataset` `source` to `GeoVectorDataset`.
- Split the `source` into chunks of size `params["target_split_size"]`.
- For each timestep in *timesteps*:
  - Generate any new waypoints from the source data. Calculate aircraft performance and run the CoCiP downwash routine over the new waypoints.
  - For each "active" contrail (i.e., a contrail that has been initialized but has not yet reach its end of life), evolve the contrail forward one step. Filter any waypoint that has reached its end of life.
- Aggregate contrail age and energy forcing predictions to a single output variable to return.

long_name = 'Gridded Contrail Cirrus Prediction Model'

met_required = True

Require meteorology is not None on __init__()

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg**-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'), MetVariable(short_name='u', standard_name='eastward_wind', long_name='Eastward Wind', level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m s**-1', amip='ua', description='"Eastward" indicates a vector component which is positive when directed eastward (negative westward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3), units='m s**-1', amip='va', description='"Northward" indicates a vector component which is positive when directed northward (negative southward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='w', standard_name='lagrangian_tendency_of_air_pressure', long_name='Vertical Velocity (omega)', level_type='isobaricInhPa', ecmwf_id=135, grib1_id=39, grib2_id=(0, 2, 8), units='Pa s**-1', amip='wap', description='The Lagrangian tendency of air pressure, often called "omega", plays the role of the upward component of air velocity when air pressure is being used as the vertical coordinate. If the vertical air velocity is upwards, it is negative when expressed as a tendency of air pressure; downwards is positive. Air pressure is the force per unit area which would be exerted when the moving gas molecules of which the air is composed strike a theoretical surface of any orientation.'), (MetVariable(short_name='ciwc', standard_name='specific_cloud_ice_water_content', long_name='Specific cloud ice water content', level_type='isobaricInhPa', ecmwf_id=247, grib1_id=None, grib2_id=(0, 1, 84), units='kg kg**-1', amip=None, description="This parameter is the mass of cloud ice particles per kilogram of the total mass of moist air. The 'total mass of moist air' is the sum of the dry air, water vapour, cloud liquid, cloud ice, rain and falling snow. This parameter represents the average value for a grid box."), MetVariable(short_name='icmr', standard_name='ice_water_mixing_ratio', long_name='Cloud ice water mixing ratio', level_type='isobaricInhPa', ecmwf_id=260019, grib1_id=None, grib2_id=(0, 1, 23), units='kg kg**-1', amip=None, description='This parameter is the mass of cloud ice particles per kilogram of the total mass of dry air. ')))

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

name = 'contrail_grid'

processed_met_variables = (MetVariable(short_name='t',
standard_name='air_temperature', long_name='Air Temperature',
level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0),
units='K', amip='ta', description='Air temperature is the bulk temperature of the
air, not the surface (skin) temperature.'), MetVariable(short_name='q',
standard_name='specific_humidity', long_name='Specific Humidity',
level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg
kg**-1', amip='hus', description='Specific means per unit mass. Specific humidity is
the mass fraction of water vapor in (moist) air.'), MetVariable(short_name='u',
standard_name='eastward_wind', long_name='Eastward Wind',
level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m
s**-1', amip='ua', description='"Eastward" indicates a vector component which is
positive when directed eastward (negative westward). Wind is defined as a
two-dimensional (horizontal) air velocity vector, with no vertical component.'),
MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward
Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3),
units='m s**-1', amip='va', description='"Northward" indicates a vector component
which is positive when directed northward (negative southward). Wind is defined as a
two-dimensional (horizontal) air velocity vector, with no vertical component.'),
MetVariable(short_name='w', standard_name='lagrangian_tendency_of_air_pressure',
long_name='Vertical Velocity (omega)', level_type='isobaricInhPa', ecmwf_id=135,
grib1_id=39, grib2_id=(0, 2, 8), units='Pa s**-1', amip='wap', description='The
Lagrangian tendency of air pressure, often called "omega", plays the role of the
upward component of air velocity when air pressure is being used as the vertical
coordinate. If the vertical air velocity is upwards, it is negative when expressed
as a tendency of air pressure; downwards is positive. Air pressure is the force per
unit area which would be exerted when the moving gas molecules of which the air is
composed strike a theoretical surface of any orientation.'),
MetVariable(short_name='tau_cirrus', standard_name='tau_cirrus', long_name='Cirrus
optical depth', level_type=None, ecmwf_id=None, grib1_id=None, grib2_id=None,
units='dimensionless', amip=None, description=None))

Set of required parameters if processing already complete on `met` input.

rad

```
rad_variables = ((MetVariable(short_name='tsr',
standard_name='top_net_solar_radiation', long_name='Top of atmosphere net solar
(shortwave) radiation', level_type='nominalTop', ecmwf_id=178, grib1_id=None,
grib2_id=(0, 4, 1), units='J m**-2', amip=None, description="This parameter is the
incoming solar radiation (also known as shortwave radiation) minus the outgoing
solar radiation at the top of the atmosphere. It is the amount of radiation passing
through a horizontal plane. The incoming solar radiation is the amount received from
the Sun. The outgoing solar radiation is the amount reflected and scattered by the
Earth's atmosphere and surfaceSee https://www.ecmwf.int/sites/default/files/
elibrary/2015/18490-radiation-quantities-ecmwf-model-and-mars.pdf"),
MetVariable(short_name='uswrf', standard_name='toa_upward_shortwave_flux',
long_name='Top of atmosphere upward shortwave radiation', level_type='nominalTop',
ecmwf_id=None, grib1_id=None, grib2_id=(0, 4, 193), units='W m**-2', amip=None,
description='This parameter is the outgoing shortwave (solar) radiation at the
nominal top of the atmosphere.')), (MetVariable(short_name='ttr',
standard_name='top_net_thermal_radiation', long_name='Top of atmosphere net thermal
(longwave) radiation', level_type='nominalTop', ecmwf_id=179, grib1_id=None,
grib2_id=(0, 5, 5), units='J m**-2', amip=None, description='The thermal (also known
as terrestrial or longwave) radiation emitted to space at the top of the atmosphere
is commonly known as the Outgoing Longwave Radiation (OLR). The top net thermal
radiation (this parameter) is equal to the negative of OLR.See
https://www.ecmwf.int/sites/default/files/elibrary/2015/
18490-radiation-quantities-ecmwf-model-and-mars.pdf'),
MetVariable(short_name='ulwrf', standard_name='toa_upward_longwave_flux',
long_name='Top of atmosphere upward longwave radiation', level_type='nominalTop',
ecmwf_id=None, grib1_id=None, grib2_id=(0, 5, 193), units='W m**-2', amip=None,
description='This parameter is the outgoing longwave (thermal) radiation at the
nominal top of the atmosphere.')))
```

property source_time

Return the time array of the source data.

timesteps

pycontrails.models.cocipgrid.CocipGridParams

class pycontrails.models.cocipgrid.**CocipGridParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(10.0, 10.0)*, *met_latitude_buffer=(10.0, 10.0)*, *met_level_buffer=(40.0, 40.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h')*, *numpy.timedelta64(0, 'h'))*, *process_emissions=True*, *dt_integration=numpy.timedelta64(30, 'm')*, *dz_m=200.0*, *effective_vertical_resolution=2000.0*, *smooth_true_airspeed=True*, *smooth_true_airspeed_window_length=7*, *smooth_true_airspeed_polyorder=1*, *humidity_scaling=None*, *compute_tau_cirrus_in_model_init='auto'*, *filter_sac=True*, *filter_initially_persistent=True*, *persistent_buffer=None*, *verbose_outputs=False*, *compute_atr20=False*, *global_rf_to_atr20_factor=0.0151*, *initial_wake_vortex_depth=0.5*, *sedimentation_impact_factor=0.5*, *default_nvpm_ei_n=1000000000000000.0*, *wind_shear_enhancement_exponent=0.5*, *nvpm_ei_n_enhancement_factor=1.0*, *min_ice_particle_number_nvpm_ei_n=10000000000000.0*, *max_depth=1500.0*, *unterstrasser_ice_survival_fraction=False*, *radiative_heating_effects=False*, *contrail_contrail_overlapping=False*, *dz_overlap_m=500.0*, *radius_threshold_um=<factory>*, *habits=<factory>*, *habit_distributions=<factory>*, *rf_sw_enhancement_factor=1.0*, *rf_lw_enhancement_factor=1.0*, *min_altitude_m=6000.0*, *max_altitude_m=13000.0*, *max_seg_length_m=40000.0*, *max_age=numpy.timedelta64(20, 'h')*, *min_tau=1e-06*, *max_tau=10000000000.0*, *min_n_ice_per_m3=1000.0*, *max_n_ice_per_m3=1e+20*, *target_split_size=100000*, *target_split_size_pre_SAC_boost=3.0*, *show_progress=True*, *segment_length=1000.0*, *fuel=<factory>*, *aircraft_type='B737'*, *engine_uid=None*, *azimuth=0.0*, *dsn_dz_factor=0.0*, *wingspan=None*, *aircraft_mass=None*, *true_airspeed=None*, *engine_efficiency=None*, *fuel_flow=None*, *aircraft_performance=None*, *verbose_outputs_formation=False*, *verbose_outputs_evolution=False*)

Bases: *CocipParams*

Default parameters for *CocipGrid*.

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(10.0, 10.0)*, *met_latitude_buffer=(10.0, 10.0)*, *met_level_buffer=(40.0, 40.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *process_emissions=True*, *dt_integration=numpy.timedelta64(30, 'm')*, *dz_m=200.0*, *effective_vertical_resolution=2000.0*, *smooth_true_airspeed=True*, *smooth_true_airspeed_window_length=7*, *smooth_true_airspeed_polyorder=1*, *humidity_scaling=None*, *compute_tau_cirrus_in_model_init='auto'*, *filter_sac=True*, *filter_initially_persistent=True*, *persistent_buffer=None*, *verbose_outputs=False*, *compute_atr20=False*, *global_rf_to_atr20_factor=0.0151*, *initial_wake_vortex_depth=0.5*, *sedimentation_impact_factor=0.5*, *default_nvpm_ei_n=1000000000000000.0*, *wind_shear_enhancement_exponent=0.5*, *nvpm_ei_n_enhancement_factor=1.0*, *min_ice_particle_number_nvpm_ei_n=10000000000000.0*, *max_depth=1500.0*, *unterstrasser_ice_survival_fraction=False*, *radiative_heating_effects=False*, *contrail_contrail_overlapping=False*, *dz_overlap_m=500.0*, *radius_threshold_um=<factory>*, *habits=<factory>*, *habit_distributions=<factory>*, *rf_sw_enhancement_factor=1.0*, *rf_lw_enhancement_factor=1.0*, *min_altitude_m=6000.0*, *max_altitude_m=13000.0*, *max_seg_length_m=40000.0*, *max_age=numpy.timedelta64(20, 'h')*, *min_tau=1e-06*, *max_tau=10000000000.0*, *min_n_ice_per_m3=1000.0*, *max_n_ice_per_m3=1e+20*, *target_split_size=100000*, *target_split_size_pre_SAC_boost=3.0*, *show_progress=True*, *segment_length=1000.0*, *fuel=<factory>*, *aircraft_type='B737'*, *engine_uid=None*, *azimuth=0.0*, *dsn_dz_factor=0.0*, *wingspan=None*, *aircraft_mass=None*, *true_airspeed=None*, *engine_efficiency=None*, *fuel_flow=None*, *aircraft_performance=None*, *verbose_outputs_formation=False*, *verbose_outputs_evolution=False*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

aircraft_mass	Nominal aircraft mass, [kg].
aircraft_performance	Aircraft performance model.
aircraft_type	ICAO code designating simulated aircraft type.
azimuth	Navigation bearing [deg] measured in clockwise direction from true north, by default 0.0.
compute_atr20	Add additional metric of ATR20 and global yearly mean RF to model output.
compute_tau_cirrus_in_model_init	Compute "tau_cirrus" variable in pressure-level met data during model initialization.
contrail_contrail_overlapping	Experimental.
copy_source	Copy input source data on eval

continues on next page

Table 7 – continued from previous page

default_nvpm_ei_n	Default nvpm_ei_n value if no data provided and emissions calculations fails.
downselect_met	Downselect input MetDataset` to region around source.
dsn_dz_factor	Experimental parameter used to approximate dsn_dz from ds_dz via dsn_dz = ds_dz * dsn_dz_factor.
dt_integration	Apply Euler's method with a fixed step size of dt_integration.
dz_m	Difference in altitude between top and bottom layer for stratification calculations, $[m]$.
dz_overlap_m	Experimental.
effective_vertical_resolution	Vertical resolution (m) associated to met data.
engine_efficiency	Nominal engine efficiency, $[0 - 1]$.
engine_uid	Engine unique identification number for the ICAO Aircraft Emissions Databank (EDB) If None, an assumed engine_uid is used in Emissions.
filter_initially_persistent	Filter out waypoints if they don't satisfy the initial persistent criteria Passing in a non-default value is unusual, but is included to allow for false negative calibration and model uncertainty studies.
filter_sac	Filter out waypoints if the don't satisfy the SAC criteria Note that the SAC algorithm will still be run to calculate T_critical_sac for use estimating initial ice particle number.
fuel_flow	Nominal fuel flow, $[kg\ s^{-1}]$.
global_rf_to_atr20_factor	Constant factor used to convert global- and year-mean RF, $[Wm^{-2}]$, to ATR20, $[K]$, given by [Yin *et al.*, 2023].
humidity_scaling	Humidity scaling
initial_wake_vortex_depth	Initial wake vortex depth scaling factor.
interpolation_bounds_error	If True, points lying outside interpolation will raise an error
interpolation_fill_value	Used for outside interpolation value if interpolation_bounds_error is False
interpolation_localize	Experimental.
interpolation_method	Interpolation method.
interpolation_q_method	Experimental.
interpolation_use_indices	Experimental.
max_age	Max age of contrail evolution.
max_altitude_m	Maximum altitude domain in simulation, $[m]$ If set to None, this check is disabled.
max_depth	Upper bound for contrail plume depth, constraining it to realistic values.
max_n_ice_per_m3	Maximum contrail ice particle number per volume of air to prevent unrealistic values.
max_seg_length_m	Maximum contrail segment length in simulation to prevent unrealistic values, $[m]$.
max_tau	Maximum contrail optical depth to prevent unrealistic values.
met_latitude_buffer	Met latitude buffer [WGS84] for Cocip evolution.

continues on next page

Table 7 – continued from previous page

met_level_buffer	Met level buffer [hPa] for Cocip initialization and evolution.
met_longitude_buffer	Met longitude [WGS84] buffer for Cocip evolution.
met_time_buffer	Met time buffer for input to `Flight.downselect_met()` Only applies when `downselect_met` is True.
min_altitude_m	Minimum altitude domain in simulation, [m] If set to `None`, this check is disabled.
min_ice_particle_number_nvpm_ei_n	Lower bound for `nvpm_ei_n` to account for ambient aerosol particles for newer engines, [kg^{-1}]
min_n_ice_per_m3	Minimum contrail ice particle number per volume of air.
min_tau	Minimum contrail optical depth.
nvpm_ei_n_enhancement_factor	Multiply flight black carbon number by enhancement factor.
persistent_buffer	Continue evolving contrail waypoints `persistent_buffer` beyond end of contrail life.
process_emissions	Determines whether `Cocip.process_emissions()` runs on model `Cocip.eval()` Set to `False` when input Flight includes emissions data.
radiative_heating_effects	Experimental.
rf_lw_enhancement_factor	Scale longwave radiative forcing.
rf_sw_enhancement_factor	Scale shortwave radiative forcing.
sedimentation_impact_factor	Sedimentation impact factor.
segment_length	Nominal segment length to place at each grid point [m].
show_progress	Display `tqdm` progress bar showing batch evaluation progress.
smooth_true_airspeed	Smoothing parameters for true airspeed.
smooth_true_airspeed_polyorder	
smooth_true_airspeed_window_length	
target_split_size	Approximate size of a typical `numpy.ndarray` used with in CoCiP calculations.
target_split_size_pre_SAC_boost	Additional boost to target split size before SAC is computed.
true_airspeed	Cruising true airspeed, [$m\ s^{-1}$].
unterstrasser_ice_survival_fraction	Experimental.
verbose_outputs	Add additional values to the flight and contrail that are not explicitly necessary for calculation.
verbose_outputs_evolution	Attach contrail evolution data to *CocipGrid.contrail_list*.
verbose_outputs_formation	Attach additional formation specific data to the output.
verify_met	Call `_verify_met()` on model instantiation.
wind_shear_enhancement_exponent	Parameter denoted by n in eq.
wingspan	Aircraft wingspan, [m].
fuel	Fuel type

continues on next page

Table 7 – continued from previous page

radius_threshold_um	Radius threshold for regime bins, [μm] This is the row index label for habit_distributions.
habits	Particle habit (shape) types.
habit_distributions	Mix of ice particle habits in each radius regime.

aircraft_mass = None

Nominal aircraft mass, [kg]. If included in CocipGrid.source, this parameter is unused. Otherwise, if this parameter is None, the *aircraft_performance* model is used to estimate the aircraft mass.

aircraft_performance = None

Aircraft performance model. Required unless source or params provide all of the following variables:

- wingspan
- true_airspeed (or mach_number)
- fuel_flow
- engine_efficiency
- aircraft_mass

If None and CocipGrid.source or *CocipGridParams* do not provide the above variables, a ValueError is raised. See PSGrid for an open-source implementation of a AircraftPerformanceGrid model.

aircraft_type = 'B737'

ICAO code designating simulated aircraft type. Needed for the *aircraft_performance* and Emissions models.

azimuth = 0.0

Navigation bearing [deg] measured in clockwise direction from true north, by default 0.0.

New in version 0.32.2.

EXPERIMENTAL: If None, run CoCiP in "segment-free" mode. This mode does not include any terms involving segments (wind shear, segment_length, any derived terms), unless *dsn_dz_factor* is non-zero.

dsn_dz_factor = 0.0

Experimental parameter used to approximate dsn_dz from ds_dz via dsn_dz = ds_dz * dsn_dz_factor. A value of 0.0 disables any normal wind shear effects. An initial unpublished experiment suggests that dsn_dz_factor = 0.665 adequately approximates the mean EF predictions of *CocipGrid* over all azimuths.

New in version 0.32.2.

engine_efficiency = None

Nominal engine efficiency, $[0 - 1]$. If included in CocipGrid.source, this parameter is unused. Otherwise, if this parameter is None, the *aircraft_performance* model is used to estimate the engine efficiency.

engine_uid = None

Engine unique identification number for the ICAO Aircraft Emissions Databank (EDB) If None, an assumed engine_uid is used in Emissions.

fuel

Fuel type

fuel_flow = None

Nominal fuel flow, $[kg\ s^{-1}]$. If included in `CocipGrid.source`, this parameter is unused. Otherwise, if this parameter is None, the *aircraft_performance* model is used to estimate the fuel flow.

segment_length = 1000.0

Nominal segment length to place at each grid point $[m]$. Round-off error can be problematic with a small nominal segment length and a large `dt_integration` parameter. On the other hand, too large of a nominal segment length diminishes the "locality" of the grid point.

New in version 0.32.2.

EXPERIMENTAL: If None, run CoCiP in "segment-free" mode. This mode does not include any terms involving segments (wind shear, segment length, any derived terms). See *azimuth* and *dsn_dz_factor* for more details.

show_progress = True

Display `tqdm` progress bar showing batch evaluation progress.

target_split_size = 100000

Approximate size of a typical `numpy.ndarray` used with in CoCiP calculations. The 4-dimensional array defining the waypoint is raveled and split into batches of this size. A smaller number for this parameter will reduce memory footprint at the expense of a longer compute time.

target_split_size_pre_SAC_boost = 3.0

Additional boost to target split size before SAC is computed. For typical meshes, only 10% of waypoints will survive SAC and initial downwash filtering. Accordingly, this parameter magnifies mesh split size before SAC is computed. See *target_split_size*.

true_airspeed = None

Cruising true airspeed, $[m\ s^{-1}]$. If included in `CocipGrid.source`, this parameter is unused. Otherwise, if this parameter is None, the *aircraft_performance* model is used to estimate the true airspeed.

verbose_outputs_evolution = False

Attach contrail evolution data to *CocipGrid.contrail_list*. Requires substantial memory overhead.

verbose_outputs_formation = False

Attach additional formation specific data to the output. If True, attach all possible formation data. See `pycontrails.models.cocipgrid.cocip_grid` for a list of supported formation data.

wingspan = None

Aircraft wingspan, $[m]$. If included in `CocipGrid.source`, this parameter is unused. Otherwise, if this parameter is None, the *aircraft_performance* model is used to estimate the wingspan.

11.3.5 ACCF

This model is a this interface over the DLR / UMadrid climaccf package. See *ACCF* for more information.

models.accf.ACCF(met[, surface, params])	Compute Algorithmic Climate Change Functions (ACCF).
models.accf.ACCFParams([copy_source, ...])	Default ACCF model parameters.

pycontrails.models.accf.ACCF

class pycontrails.models.accf.**ACCF**(*met*, *surface=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Compute Algorithmic Climate Change Functions (ACCF).

This class is a wrapper over the DLR / UMadrid library climaccf, DOI: 10.5281/zenodo.6977272

Parameters

- **met** (*MetDataset*) – Dataset containing "air_temperature" and "specific_humidity" variables
- **surface** (*MetDataset*, *optional*) – Dataset containing "surface_solar_downward_radiation" and "top_net_thermal_radiation" variables

References

- [Dietmüller, 2022]
- [Dietmüller *et al.*, 2022]

__init__(*met*, *surface=None*, *params=None*, ***params_kwargs*)

Methods

__init__(met[, surface, params])	
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
eval([source])	Evaluate accfs along flight trajectory or on meteorology grid.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is type_.
set_source([source])	Attach original or copy of input source to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to source.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Data evaluated in model
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
path_lib	
short_vars	
sur_variables	
processed_met_variables	Set of required parameters if processing already complete on `met` input.
optional_met_variables	Optional meteorology variables

default_params

alias of *ACCFParams*

eval(*source=None*, ***params*)

Evaluate accfs along flight trajectory or on meteorology grid.

Parameters

- **source** (`GeoVectorDataset | Flight | MetDataset | None`, *optional*) – Input GeoVectorDataset or Flight. If None, evaluates at the *met* grid points.
- ****params** (`Any`) – Overwrite model parameters before eval

Returns

`GeoVectorDataset | Flight | MetDataArray` – Returns *np.nan* if interpolating outside meteorology grid.

Raises

NotImplementedError – Raises if input `source` is not supported.

long_name = 'algorithmic climate change functions'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg**-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'), MetVariable(short_name='pv', standard_name='potential_vorticity', long_name='Potential vorticity (K m^2 / kg s)', level_type='isobaricInhPa', ecmwf_id=60, grib1_id=128, grib2_id=(0, 2, 14), units='K m**2 kg**-1 s**-1', amip='pvu', description='Potential vorticity is a measure of the capacity for air to rotate in the atmosphere.If we ignore the effects of heating and friction, potential vorticity is conserved following an air parcel.It is used to look for places where large wind storms are likely to originate and develop.Potential vorticity increases strongly above the tropopause and therefore, it can also be used in studiesrelated to the stratosphere and stratosphere-troposphere exchanges. Large wind storms develop when a columnof air in the atmosphere starts to rotate. Potential vorticity is calculated from the wind, temperature andpressure across a column of air in the atmosphere.'), MetVariable(short_name='z', standard_name='geopotential', long_name='Geopotential', level_type='isobaricInhPa', ecmwf_id=129, grib1_id=6, grib2_id=(0, 3, 4), units='m**2 s**-2', amip=None, description='Geopotential is the sum of the specific gravitational potential energy relative to the geoid and the specific centripetal potential energy.'), MetVariable(short_name='r', standard_name='relative_humidity', long_name='Relative Humidity', level_type='isobaricInhPa', ecmwf_id=157, grib1_id=52, grib2_id=(0, 1, 1), units='1', amip='hur', description='This parameter is the water vapour pressure as a percentage of the value at which the air becomes saturated liquid.'), MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3), units='m s**-1', amip='va', description='"Northward" indicates a vector component which is positive when directed northward (negative southward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='u', standard_name='eastward_wind', long_name='Eastward Wind', level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m s**-1', amip='ua', description='"Eastward" indicates a vector component which is positive when directed eastward (negative westward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='pv', standard_name='potential_vorticity', long_name='Potential vorticity (K m^2 / kg s)', level_type='isobaricInhPa', ecmwf_id=60, grib1_id=128, grib2_id=(0, 2, 14), units='K m**2 kg**-1 s**-1', amip='pvu', description='Potential vorticity is a measure of the capacity for air to rotate in the atmosphere.If we ignore the effects of heating and friction, potential vorticity is conserved following an air parcel.It is used to look for places where large wind storms are likely to originate and develop.Potential vorticity increases strongly above the tropopause and therefore, it can also be used in studiesrelated to the stratosphere and stratosphere-troposphere exchanges. Large wind storms develop when a columnof air in the atmosphere starts to rotate. Potential vorticity is calculated from the wind, temperature andpressure across a column of air in the atmosphere.'))

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

name = 'accr'

params

Instantiated model parameters, in dictionary form

path_lib = './'

short_vars = {'pv', 'q', 'r', 'ssrd', 't', 'ttr', 'u', 'v', 'z'}

source

Data evaluated in model

sur_variables = (MetVariable(short_name='ssrd', standard_name='surface_solar_downward_radiation', long_name='Surface Solar Downward Radiation', level_type='surface', ecmwf_id=169, grib1_id=None, grib2_id=(0, 4, 7), units='J m-2', amip=None, description='This parameter is the amount of solar radiation (also known as shortwave radiation) that reaches a horizontal plane at the surface of the Earth. This parameter comprises both direct and diffuse solar radiation.'), MetVariable(short_name='ttr', standard_name='top_net_thermal_radiation', long_name='Top of atmosphere net thermal (longwave) radiation', level_type='nominalTop', ecmwf_id=179, grib1_id=None, grib2_id=(0, 5, 5), units='J m**-2', amip=None, description='The thermal (also known as terrestrial or longwave) radiation emitted to space at the top of the atmosphere is commonly known as the Outgoing Longwave Radiation (OLR). The top net thermal radiation (this parameter) is equal to the negative of OLR.See https://www.ecmwf.int/sites/default/files/elibrary/2015/18490-radiation-quantities-ecmwf-model-and-mars.pdf'))**

pycontrails.models.accf.ACCFParams

class pycontrails.models.accf.**ACCFParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *lat_bound=None*, *lon_bound=None*, *efficacy=True*, *efficacy_option='lee_2021'*, *accf_v='V1.0'*, *ch4_scaling=1.0*, *co2_scaling=1.0*, *cont_scaling=1.0*, *h2o_scaling=1.0*, *o3_scaling=1.0*, *forecast_step=6.0*, *sep_ri_rw=False*, *climate_indicator='ATR'*, *horizontal_resolution=0.5*, *emission_scenario='pulse'*, *time_horizon=20*, *pfca='PCFA-ISSR'*, *merged=True*, *issr_rhi_threshold=0.9*, *issr_temp_threshold=235*, *sac_ei_h2o=1.25*, *sac_q=43000000.0*, *sac_eta=0.3*, *nox_ei='TTV'*, *PMO=False*)

Bases: *ModelParams*

Default ACCF model parameters.

See *config-user.yml* definition at https://github.com/dlr-pa/climaccf

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *lat_bound=None*, *lon_bound=None*, *efficacy=True*, *efficacy_option='lee_2021'*, *accf_v='V1.0'*, *ch4_scaling=1.0*, *co2_scaling=1.0*, *cont_scaling=1.0*, *h2o_scaling=1.0*, *o3_scaling=1.0*, *forecast_step=6.0*, *sep_ri_rw=False*, *climate_indicator='ATR'*, *horizontal_resolution=0.5*, *emission_scenario='pulse'*, *time_horizon=20*, *pfca='PCFA-ISSR'*, *merged=True*, *issr_rhi_threshold=0.9*, *issr_temp_threshold=235*, *sac_ei_h2o=1.25*, *sac_q=43000000.0*, *sac_eta=0.3*, *nox_ei='TTV'*, *PMO=False*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

PMO	
accf_v	
ch4_scaling	
climate_indicator	
co2_scaling	
cont_scaling	
copy_source	Copy input source data on eval
downselect_met	Downselect input MetDataset` to region around source.
efficacy	
efficacy_option	
emission_scenario	
forecast_step	
h2o_scaling	
horizontal_resolution	
interpolation_bounds_error	If True, points lying outside interpolation will raise an error

continues on next page

Table 8 – continued from previous page

interpolation_fill_value	Used for outside interpolation value if interpolation_bounds_error is False
interpolation_localize	Experimental.
interpolation_method	Interpolation method.
interpolation_q_method	Experimental.
interpolation_use_indices	Experimental.
issr_rhi_threshold	RHI Threshold
issr_temp_threshold	
lat_bound	
lon_bound	
merged	
met_latitude_buffer	Met latitude buffer for input to Flight.downselect_met(), in WGS84 coordinates.
met_level_buffer	Met level buffer for input to Flight.downselect_met(), in $[hPa]$.
met_longitude_buffer	Met longitude buffer for input to Flight.downselect_met(), in WGS84 coordinates.
met_time_buffer	Met time buffer for input to Flight.downselect_met() Only applies when downselect_met is True.
nox_ei	
o3_scaling	
pfca	
sac_ei_h2o	
sac_eta	
sac_q	
sep_ri_rw	
time_horizon	
verify_met	Call _verify_met() on model instantiation.

PMO = False

accf_v = 'V1.0'

ch4_scaling = 1.0

climate_indicator = 'ATR'

co2_scaling = 1.0

cont_scaling = 1.0

efficacy = True

efficacy_option = 'lee_2021'

emission_scenario = 'pulse'

forecast_step = 6.0

h2o_scaling = 1.0

horizontal_resolution = 0.5

issr_rhi_threshold = 0.9

RHI Threshold

issr_temp_threshold = 235

lat_bound = None

lon_bound = None

merged = True

nox_ei = 'TTV'

o3_scaling = 1.0

pfca = 'PCFA-ISSR'

sac_ei_h2o = 1.25

sac_eta = 0.3

sac_q = 43000000.0

sep_ri_rw = False

time_horizon = 20

11.3.6 Aircraft Performance

core.aircraft_performance	Abstract interfaces for aircraft performance models.
models.ps_model.PSFlightParams([...])	Default parameters for `PSFlight`.
models.ps_model.PSFlight([met, params])	Simulate aircraft performance using Poll-Schumann (PS) model.
models.ps_model.PSGrid([met, params])	Compute nominal Poll-Schumann aircraft performance over a grid.
models.ps_model.PSAircraftEngineParams(...)	Store extracted aircraft and engine parameters for each aircraft type.
models.ps_model.ps_nominal_grid(aircraft_type, *)	Calculate the nominal performance grid for a given aircraft type.

pycontrails.core.aircraft_performance

Abstract interfaces for aircraft performance models.

Module Attributes

DEFAULT_LOAD_FACTOR	Default load factor for aircraft performance models.

Classes

AircraftPerformance([met, params])	Support for standardizing aircraft performance methodologies.
AircraftPerformanceData(fuel_flow, ...)	Store the computed aircraft performance metrics.
AircraftPerformanceGrid([met, params])	Support for standardizing aircraft performance methodologies on a grid.
AircraftPerformanceGridData(fuel_flow, ...)	Store the computed aircraft performance metrics for nominal cruise conditions.
AircraftPerformanceGridParams([copy_source, ...])	Parameters for *AircraftPerformanceGrid*.
AircraftPerformanceParams([copy_source, ...])	Parameters for *AircraftPerformance*.

class pycontrails.core.aircraft_performance.**AircraftPerformance**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Support for standardizing aircraft performance methodologies.

This class provides a *simulate_fuel_and_performance()* method for iteratively calculating aircraft mass and fuel flow rate.

The implementing class must bring *eval()* and *calculate_aircraft_performance()* methods. At runtime, these methods are intended to be chained together as follows:

1. The *eval()* method is called with a `Flight`
2. The *simulate_fuel_and_performance()* method is called inside *eval()* to iteratively calculate aircraft mass and fuel flow rate. If an aircraft mass is provided, the fuel flow rate is calculated once directly with a single call to *calculate_aircraft_performance()*. If an aircraft mass is not provided, the fuel flow rate is calculated iteratively with multiple calls to *calculate_aircraft_performance()*.

abstract calculate_aircraft_performance(***, *aircraft_type*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *engine_efficiency*, *fuel_flow*, *thrust*, *q_fuel*, ***kwargs*)

Calculate aircraft performance along a trajectory.

When `time` is not None, this method should be used for a single flight trajectory. Waypoints are coupled via the `time` parameter.

This method computes the rate of climb and descent (ROCD) to determine flight phases: “cruise”, “climb”, and “descent”. Performance metrics depend on this phase.

When `time` is None, this method can be used to simulate flight performance over an arbitrary sequence of flight waypoints by assuming nominal flight characteristics. In this case, each point is treated independently and all points are assumed to be in a “cruise” phase of the flight.

Parameters

- **aircraft_type** (`str`) – Used to query the underlying model database for aircraft engine parameters.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude at each waypoint, [ft]
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, [K]
- **time** (`npt.NDArray[np.datetime64] | None`) – Waypoint time in `np.datetime64` format. If None, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (`npt.NDArray[np.float64] | float | None`) – True airspeed for each waypoint, [ms^{-1}]. If None, a nominal value is used.
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – Aircraft mass for each waypoint, [kg].
- **engine_efficiency** (`npt.NDArray[np.float64] | float | None`) – Override the engine efficiency at each waypoint.
- **fuel_flow** (`npt.NDArray[np.float64] | float | None`) – Override the fuel flow at each waypoint, [kgs^{-1}].
- **thrust** (`npt.NDArray[np.float64] | float | None`) – Override the thrust setting at each waypoint, [:math: N].
- **q_fuel** (`float`) – Lower calorific value (LCV) of fuel, [$J\ kg_{fuel}^{-1}$].
- ****kwargs** (`Any`) – Additional keyword arguments to pass to the model.

Returns

AircraftPerformanceData – Derived performance metrics at each waypoint.

ensure_true_airspeed_on_source()

Add `true_airspeed` field to *source* data if not already present.

Returns

`npt.NDArray[np.float64]` – True airspeed, [ms^{-1}]. If `true_airspeed` is already present on *source*, this is returned directly. Otherwise, it is calculated using `Flight.segment_true_airspeed()`.

abstract eval(*source=None*, ***params*)

Evaluate the aircraft performance model.

The implementing model adds the following fields to the source flight:

- `aircraft_mass`: aircraft mass at each waypoint, [kg]
- `fuel_flow`: fuel mass flow rate at each waypoint, [kgs^{-1}]
- `thrust`: thrust at each waypoint, [N]
- `engine_efficiency`: engine efficiency at each waypoint
- `rocd`: rate of climb or descent at each waypoint, [$ftmin^{-1}$]
- `fuel_burn`: fuel burn at each waypoint, [kg]

In addition, the following attributes are added to the source flight:

- `n_engine`: number of engines
- `wingspan`: wingspan, [m]

- `max_mach`: maximum Mach number
- `max_altitude`: maximum altitude, $[m]$
- `total_fuel_burn`: total fuel burn, $[kg]$

Parameters

- **source** (*Flight*) – Flight trajectory to evaluate.
- **params** (Any) – Override *params* with keyword arguments.

Returns

Flight – Flight trajectory with aircraft performance data.

met

Meteorology data

params

Instantiated model parameters, in dictionary form

simulate_fuel_and_performance(*, *aircraft_type*, *altitude_ft*, *time*, *true_airspeed*, *air_temperature*, *aircraft_mass*, *thrust*, *engine_efficiency*, *fuel_flow*, *q_fuel*, *n_iter*, *amass_oew*, *amass_mtow*, *amass_mpl*, *load_factor*, *takeoff_mass*, ***kwargs*)

Calculate aircraft mass, fuel mass flow rate, and overall propulsion efficiency.

This method performs `n_iter` iterations, each of which calls *calculate_aircraft_performance()*. Each successive iteration generates a better estimate for mass fuel flow rate and aircraft mass at each waypoint.

Parameters

- **aircraft_type** (`str`) – Aircraft type designator used to query the underlying model database.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude at each waypoint, $[ft]$
- **time** (`npt.NDArray[np.datetime64]`) – Waypoint time in `np.datetime64` format.
- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed for each waypoint, $[ms^{-1}]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **aircraft_mass** (`npt.NDArray[np.float64] | float | None`) – Override the aircraft_mass at each waypoint, $[kg]$.
- **thrust** (`npt.NDArray[np.float64] | float | None`) – Override the thrust setting at each waypoint, [:math: N].
- **engine_efficiency** (`npt.NDArray[np.float64] | float | None`) – Override the engine efficiency at each waypoint.
- **fuel_flow** (`npt.NDArray[np.float64] | float | None`) – Override the fuel flow at each waypoint, $[kgs^{-1}]$.
- **q_fuel** (`float`) – Lower calorific value (LCV) of fuel, $[J\ kg^{-1}_{fuel}]$.
- **amass_oew** (`float`) – Aircraft operating empty weight, $[kg]$. Used to determine the initial aircraft mass if `takeoff_mass` is not provided. This quantity is constant for a given aircraft type.

- **amass_mtow** (float) – Aircraft maximum take-off weight, [kg]. Used to determine the initial aircraft mass if `takeoff_mass` is not provided. This quantity is constant for a given aircraft type.
- **amass_mpl** (float) – Aircraft maximum payload, [kg]. Used to determine the initial aircraft mass if `takeoff_mass` is not provided. This quantity is constant for a given aircraft type.
- **load_factor** (float) – Aircraft load factor assumption (between 0 and 1). If unknown, a value of 0.7 is a reasonable default. Typically, this parameter is between 0.6 and 0.8. During the height of the COVID-19 pandemic, this parameter was often much lower.
- **takeoff_mass** (`float | None`, *optional*) – If known, the takeoff mass can be provided to skip the calculation in `jet.initial_aircraft_mass()`. In this case, the parameters `load_factor`, `amass_oew`, `amass_mtow`, and `amass_mpl` are ignored.
- ****kwargs** (Any) – Additional keyword arguments are passed to *calculate_aircraft_performance()*.

Returns

AircraftPerformanceData – Results from the final iteration is returned.

source

Data evaluated in model

class pycontrails.core.aircraft_performance.**AircraftPerformanceData**(*fuel_flow*, *aircraft_mass*, *true_airspeed*, *fuel_burn*, *thrust*, *engine_efficiency*, *rocd*)

Bases: object

Store the computed aircraft performance metrics.

Parameters

- **fuel_flow** (`npt.NDArray[np.float64]`) – Fuel mass flow rate for each waypoint, [kgs^{-1}]
- **aircraft_mass** (`npt.NDArray[np.float64]`) – Aircraft mass for each waypoint, [kg]
- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed at each waypoint, [:math: *m s^{-1}*]
- **fuel_burn** (`npt.NDArray[np.float64]`) – Fuel consumption for each waypoint, [kg]. Set to an array of all nan values if it cannot be computed (ie, working with gridpoints).
- **thrust** (`npt.NDArray[np.float64]`) – Thrust force, [N]
- **engine_efficiency** (`npt.NDArray[np.float64]`) – Overall propulsion efficiency for each waypoint
- **rocd** (`npt.NDArray[np.float64]`) – Rate of climb and descent, [$ftmin^{-1}$]

aircraft_mass

engine_efficiency

fuel_burn

fuel_flow

rocd

thrust

true_airspeed

class pycontrails.core.aircraft_performance.**AircraftPerformanceGrid**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Support for standardizing aircraft performance methodologies on a grid.

Currently just a container until additional models are implemented.

abstract eval(*source=None*, ***params*)

Evaluate the aircraft performance model.

met

Meteorology data

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

class pycontrails.core.aircraft_performance.**AircraftPerformanceGridData**(*fuel_flow*, *engine_efficiency*)

Bases: Generic[*ArrayOrFloat*]

Store the computed aircraft performance metrics for nominal cruise conditions.

engine_efficiency

Engine efficiency, $[0 - 1]$

fuel_flow

Fuel mass flow rate, $[kgs^{-1}]$

class pycontrails.core.aircraft_performance.**AircraftPerformanceGridParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64('h')*, *numpy.timedelta64(0, 'h'))*, *fuel=<factory>*, *aircraft_type='B737'*, *mach_number=None*, *aircraft_mass=None*)

Bases: *ModelParams*

Parameters for *AircraftPerformanceGrid*.

aircraft_mass = None

Aircraft mass, $[kg]$ If `None`, a nominal value is determined by the implementation. Can be overridden by including an `aircraft_mass` key in source data

aircraft_type = 'B737'

ICAO code designating simulated aircraft type. Can be overridden by including `aircraft_type` attribute in source data

fuel

Fuel type

mach_number = None

Mach number, $[Ma]$ If `None`, a nominal cruise value is determined by the implementation. Can be overridden by including a `mach_number` key in source data

class pycontrails.core.aircraft_performance.**AircraftPerformanceParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *correct_fuel_flow=True*, *n_iter=3*)

Bases: *ModelParams*

Parameters for *AircraftPerformance*.

correct_fuel_flow = True

Whether to correct fuel flow to ensure it remains within the operational limits of the aircraft type.

n_iter = 3

The number of iterations used to calculate aircraft mass and fuel flow. The default value of 3 is sufficient for most cases.

pycontrails.core.aircraft_performance.**DEFAULT_LOAD_FACTOR = 0.7**

Default load factor for aircraft performance models.

pycontrails.models.ps_model.PSFlightParams

class pycontrails.models.ps_model.**PSFlightParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *correct_fuel_flow=True*, *n_iter=3*, *eta_over_eta_b_min=0.5*)

Bases: *AircraftPerformanceParams*

Default parameters for *PSFlight*.

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *correct_fuel_flow=True*, *n_iter=3*, *eta_over_eta_b_min=0.5*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

copy_source	Copy input source data on eval
correct_fuel_flow	Whether to correct fuel flow to ensure it remains within the operational limits of the aircraft type.
downselect_met	Downselect input MetDataset` to region around source.
eta_over_eta_b_min	Clip the ratio of the overall propulsion efficiency to the maximum propulsion efficiency to always exceed this value.
interpolation_bounds_error	If True, points lying outside interpolation will raise an error
interpolation_fill_value	Used for outside interpolation value if interpolation_bounds_error is False
interpolation_localize	Experimental.
interpolation_method	Interpolation method.
interpolation_q_method	Experimental.
interpolation_use_indices	Experimental.
met_latitude_buffer	Met latitude buffer for input to Flight.downselect_met(), in WGS84 coordinates.
met_level_buffer	Met level buffer for input to Flight.downselect_met(), in $[hPa]$.
met_longitude_buffer	Met longitude buffer for input to Flight.downselect_met(), in WGS84 coordinates.
met_time_buffer	Met time buffer for input to Flight.downselect_met() Only applies when downselect_met is True.
n_iter	The number of iterations used to calculate aircraft mass and fuel flow.
verify_met	Call _verify_met() on model instantiation.

eta_over_eta_b_min = 0.5

Clip the ratio of the overall propulsion efficiency to the maximum propulsion efficiency to always exceed this value.

pycontrails.models.ps_model.PSFlight

class pycontrails.models.ps_model.**PSFlight**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *AircraftPerformance*

Simulate aircraft performance using Poll-Schumann (PS) model.

References

[Poll and Schumann, 2021] [Poll and Schumann, 2021]

Poll & Schumann (2022). An estimation method for the fuel burn and other performance characteristics of civil transport aircraft. Part 3 Generalisation to cover climb, descent and holding. Aero. J., submitted.

__init__(*met=None*, *params=None*, ***params_kwargs*)

Methods

__init__([met, params])	
calculate_aircraft_performance(*, ...)	Calculate aircraft performance along a trajectory.
check_aircraft_type_availability(aircraft_typ	Check if aircraft type designator is available in the PS model database.
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
ensure_true_airspeed_on_source()	Add true_airspeed field to *source* data if not already present.
eval([source])	Evaluate the aircraft performance model.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is type_.
set_source([source])	Attach original or copy of input source to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
simulate_fuel_and_performance(*, ...)	Calculate aircraft mass, fuel mass flow rate, and overall propulsion efficiency.
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to source.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Data evaluated in model
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
optional_met_variables	Optional meteorology variables
aircraft_engine_params	
processed_met_variables	Set of required parameters if processing already complete on `met` input.

aircraft_engine_params

calculate_aircraft_performance(*, *aircraft_type*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *engine_efficiency*, *fuel_flow*, *thrust*, *q_fuel*, ***kwargs*)

Calculate aircraft performance along a trajectory.

When `time` is not None, this method should be used for a single flight trajectory. Waypoints are coupled via the `time` parameter.

This method computes the rate of climb and descent (ROCD) to determine flight phases: "cruise", "climb", and "descent". Performance metrics depend on this phase.

When `time` is None, this method can be used to simulate flight performance over an arbitrary sequence of flight waypoints by assuming nominal flight characteristics. In this case, each point is treated independently and all points are assumed to be in a "cruise" phase of the flight.

Parameters

- **aircraft_type** (`str`) – Used to query the underlying model database for aircraft engine parameters.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude at each waypoint, $[ft]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **time** (`npt.NDArray[np.datetime64] | None`) – Waypoint time in `np.datetime64` format. If None, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (`npt.NDArray[np.float64] | float | None`) – True airspeed for each waypoint, $[ms^{-1}]$. If None, a nominal value is used.
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – Aircraft mass for each waypoint, $[kg]$.
- **engine_efficiency** (`npt.NDArray[np.float64] | float | None`) – Override the engine efficiency at each waypoint.

- **fuel_flow** (npt.NDArray[np.float64] | float | None) – Override the fuel flow at each waypoint, [kgs^{-1}].
- **thrust** (npt.NDArray[np.float64] | float | None) – Override the thrust setting at each waypoint, [:math: N].
- **q_fuel** (float) – Lower calorific value (LCV) of fuel, [$J\ kg_{fuel}^{-1}$].
- ****kwargs** (Any) – Additional keyword arguments to pass to the model.

Returns
AircraftPerformanceData – Derived performance metrics at each waypoint.

check_aircraft_type_availability(*aircraft_type*, *raise_error=True*)

Check if aircraft type designator is available in the PS model database.

Parameters

- **aircraft_type** (str) – ICAO aircraft type designator
- **raise_error** (bool, *optional*) – Optional flag for raising an error, by default True.

Returns
bool – Aircraft found in the PS model database.

Raises
KeyError – raises KeyError if the aircraft type is not covered by database

default_params

alias of *PSFlightParams*

eval(*source=None*, ***params*)

Evaluate the aircraft performance model.

The implementing model adds the following fields to the source flight:

- aircraft_mass: aircraft mass at each waypoint, [kg]
- fuel_flow: fuel mass flow rate at each waypoint, [kgs^{-1}]
- thrust: thrust at each waypoint, [N]
- engine_efficiency: engine efficiency at each waypoint
- rocd: rate of climb or descent at each waypoint, [$ftmin^{-1}$]
- fuel_burn: fuel burn at each waypoint, [kg]

In addition, the following attributes are added to the source flight:

- n_engine: number of engines
- wingspan: wingspan, [m]
- max_mach: maximum Mach number
- max_altitude: maximum altitude, [m]
- total_fuel_burn: total fuel burn, [kg]

Parameters

- **source** (*Flight*) – Flight trajectory to evaluate.
- **params** (Any) – Override *params* with keyword arguments.

Returns
Flight – Flight trajectory with aircraft performance data.

long_name = 'Poll-Schumann Aircraft Performance Model'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'),)

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

name = 'PSFlight'

optional_met_variables = (MetVariable(short_name='u', standard_name='eastward_wind', long_name='Eastward Wind', level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m s-1', amip='ua', description='"Eastward" indicates a vector component which is positive when directed eastward (negative westward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3), units='m s**-1', amip='va', description='"Northward" indicates a vector component which is positive when directed northward (negative southward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'))**

Optional meteorology variables

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

pycontrails.models.ps_model.PSGrid

class pycontrails.models.ps_model.**PSGrid**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *AircraftPerformanceGrid*

Compute nominal Poll-Schumann aircraft performance over a grid.

For a given aircraft type, altitude, aircraft mass, air temperature, and mach number, the PS model computes a theoretical engine efficiency and fuel flow rate for an aircraft under cruise conditions. Letting the aircraft mass vary and fixing the other parameters, the engine efficiency curve attains a single maximum at a particular aircraft mass. By solving this implicit equation, the PS model can be used to compute the aircraft mass that maximizes engine efficiency for a given set of parameters. This is the "nominal" aircraft mass computed by this model.

This nominal aircraft mass is not always realizable. For example, the maximum engine efficiency may be attained at an aircraft mass that is less than the operating empty mass of the aircraft. This model determines the minimum and maximum possible aircraft mass for a given set of parameters using a simple heuristic. The nominal aircraft mass is then clipped to this range.

__init__(*met=None*, *params=None*, ***params_kwargs*)

Methods

__init__([met, params])	
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
eval([source])	Evaluate the PS model over a MetDataset or GeoVectorDataset.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is type_.
set_source([source])	Attach original or copy of input source to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to source.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Data evaluated in model
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
processed_met_variables	Set of required parameters if processing already complete on met input.
optional_met_variables	Optional meteorology variables

default_params

alias of PSGridParams

eval(*source=None*, ***params*)

Evaluate the PS model over a MetDataset or GeoVectorDataset.

Parameters

- **source** (GeoVectorDataset | MetDataset | None, *optional*) – The source data to use for the evaluation. If None, the source is taken from the *met* attribute of the *PSGrid* instance. The aircraft type is taken from source.attrs["aircraft_type"]. If this field is not present, params["aircraft_type"] is used instead. See the static CSV file ps-aircraft-params-20231117.csv for a list of supported aircraft types.
- ****params** (Any) – Override the default parameters of the *PSGrid* instance.

Returns

GeoVectorDataset | MetDataset –

The source data with the following variables added:

- aircraft_mass
- fuel_flow
- engine_efficiency

Raises

NotImplementedError – If "true_airspeed" or "aircraft_mass" fields are included in *source*.

See also:

ps_nominal_grid()

long_name = 'Poll-Schumann Aircraft Performance evaluated at arbitrary points'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'),)

Required meteorology pressure level variables. Each element in the list is a MetVariable or a tuple[MetVariable]. If element is a tuple[MetVariable], the variable depends on the data source. Only one variable in the tuple is required.

name = 'PSGrid'

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

pycontrails.models.ps_model.PSAircraftEngineParams

class pycontrails.models.ps_model.**PSAircraftEngineParams**(*manufacturer*, *aircraft_type*, *n_engine*, *winglets*, *amass_mtow*, *amass_mlw*, *amass_mzfw*, *amass_oew*, *amass_mpl*, *wing_surface_area*, *wing_span*, *fuselage_width*, *delta_2*, *cos_sweep*, *wing_aspect_ratio*, *psi_0*, *x_ref*, *wing_constant*, *j_1*, *j_2*, *j_3*, *c_l_do*, *f_00*, *ff_max_sls*, *ff_idle_sls*, *m_des*, *c_t_des*, *eta_1*, *eta_2*, *tr_ec*, *m_ec*, *tet_mto*, *tet_mcc*, *nominal_opr*, *nominal_bpr*, *nominal_fpr*, *fl_max*, *max_mach_num*, *p_i_max*, *p_inf_co*)

Bases: object

Store extracted aircraft and engine parameters for each aircraft type.

AIRCRAFT INFORMATION

- `manufacturer` Aircraft manufacturer name
- `aircraft_type` Specific aircraft type variant
- `n_engine` Number of engines
- `winglets` Does the aircraft type contain winglets? (True/False)

AIRCRAFT MASS PARAMETERS

- `amass_mtow` Aircraft maximum take-off weight, [kg]
- `amass_mlw` Aircraft maximum landing weight, [kg]
- `amass_mzfw` Aircraft maximum zero fuel weight, [kg]
- `amass_oew` Aircraft operating empty weight, [kg]
- `amass_mpl` Aircraft maximum payload, [kg]

AIRCRAFT GEOMETRY

- `wing_surface_area` Reference wing surface area, [m^2]
- `wing_span` Wing span, [m]
- `fuselage_width` Aircraft fuselage width, [m]
- `delta_2` Induced drag wing-fuselage interference factor
- `cos_sweep` Cosine of wing sweep angle measured at the 1/4 chord line
- `wing_aspect_ratio` Wing aspect ratio, `wing_span**2 / wing_surface_area`

AERODYNAMIC PARAMETERS

- `psi_0` Aircraft geometry drag parameter
- **`x_ref` Threshold condition where the terminating shockwave moves onto** the rear part of the wing
- **`wing_constant` A constant used in the wave drag model, capturing the aerofoil** technology factor and wing geometry
- `j_1` Wave drag parameter 1
- `j_2` Wave drag parameter 2
- `j_3` Wave drag parameter 3
- `c_l_do` Design optimum lift coefficient

ENGINE PARAMETERS

- **f_00 Maximum thrust force that can be supplied by the engine at sea level static** conditions, summed over all engines, [N].
- **ff_max_sls Fuel mass flow rate at take-off and sea level static conditions, summed** over all engines, [kgs^{-1}]
- **ff_idle_sls Fuel mass flow rate under engine idle and sea level static conditions,** summed over all engines, [kgs^{-1}]
- m_des Design optimum Mach number where the fuel mass flow rate is at a minimum.
- **c_t_des Design optimum engine thrust coefficient where the fuel mass flow rate is** at a minimum.
- eta_1 Multiplier for maximum overall propulsion efficiency model
- eta_2 Exponent for maximum overall propulsion efficiency model
- **tr_ec Engine characteristic ratio of total turbine-entry-temperature to** the total freestream temperature for maximum overall efficiency.
- m_ec Engine characteristic Mach number associated with *tr_ec*.
- tet_mto Turbine entry temperature at maximum take-off rating, [K]
- tet_mcc Turbine entry temperature at maximum continuous climb rating, [K]
- nominal_opr Nominal engine operating pressure ratio.
- nominal_bpr Nominal engine bypass ratio.
- nominal_fpr Nominal engine fan pressure ratio.

HEIGHT AND SPEED LIMITS

- fl_max Maximum flight level
- max_mach_num Maximum operational Mach number
- p_i_max Maximum operational impact pressure, [Pa]
- p_inf_co Crossover pressure altitude, [Pa]

__init__(*manufacturer*, *aircraft_type*, *n_engine*, *winglets*, *amass_mtow*, *amass_mlw*, *amass_mzfw*, *amass_oew*, *amass_mpl*, *wing_surface_area*, *wing_span*, *fuselage_width*, *delta_2*, *cos_sweep*, *wing_aspect_ratio*, *psi_0*, *x_ref*, *wing_constant*, *j_1*, *j_2*, *j_3*, *c_l_do*, *f_00*, *ff_max_sls*, *ff_idle_sls*, *m_des*, *c_t_des*, *eta_1*, *eta_2*, *tr_ec*, *m_ec*, *tet_mto*, *tet_mcc*, *nominal_opr*, *nominal_bpr*, *nominal_fpr*, *fl_max*, *max_mach_num*, *p_i_max*, *p_inf_co*)

Methods

__init__(manufacturer, aircraft_type, ...)

Attributes

manufacturer
aircraft_type
n_engine
winglets
amass_mtow
amass_mlw
amass_mzfw
amass_oew
amass_mpl
wing_surface_area
wing_span
fuselage_width
delta_2
cos_sweep
wing_aspect_ratio
psi_0
x_ref
wing_constant
j_1
j_2
j_3

continues on next page

Table 9 – continued from previous page

c_l_do
f_00
ff_max_sls
ff_idle_sls
m_des
c_t_des
eta_1
eta_2
tr_ec
m_ec
tet_mto
tet_mcc
nominal_opr
nominal_bpr
nominal_fpr
fl_max
max_mach_num
p_i_max
p_inf_co

aircraft_type

amass_mlw

amass_mpl

amass_mtow

amass_mzfw

amass_oew

c_l_do

c_t_des

cos_sweep

delta_2

eta_1

eta_2

f_00

ff_idle_sls

ff_max_sls

fl_max

fuselage_width

j_1

j_2

j_3

m_des

m_ec

manufacturer

max_mach_num

n_engine

nominal_bpr

nominal_fpr

nominal_opr

p_i_max

p_inf_co

psi_0

tet_mcc

tet_mto

tr_ec

wing_aspect_ratio

wing_constant

wing_span

wing_surface_area

winglets

x_ref

pycontrails.models.ps_model.ps_nominal_grid

pycontrails.models.ps_model.**ps_nominal_grid**(*aircraft_type*, *, *level=None*, *air_temperature=None*, *q_fuel=43130000.0*, *mach_number=None*, *maxiter=10*)

Calculate the nominal performance grid for a given aircraft type.

This function is similar to the *PSGrid* model, but it doesn't require meteorological data. Instead, the ambient air temperature can be computed from the ISA model or passed as an argument.

Parameters

- **aircraft_type** (`str`) – The aircraft type.
- **level** (`npt.NDArray[np.float64] | None`, *optional*) – The pressure level, $[hPa]$. If None, the `air_temperature` argument must be a `xarray.DataArray` with a `level` coordinate.
- **air_temperature** (`xr.DataArray | npt.NDArray[np.float64] | None`, *optional*) – The ambient air temperature, $[K]$. If None (default), the ISA temperature is computed from the `level` argument. If a `xarray.DataArray`, the `level` coordinate must be present and the `level` argument must be None to avoid ambiguity. If a `numpy.ndarray` is passed, it is assumed to be 1 dimensional with the same shape as the `level` argument.
- **q_fuel** (`float`, *optional*) – The fuel heating value, by default `JetA.q_fuel`
- **mach_number** (`float | None`, *optional*) – The Mach number. If None (default), the PS design Mach number is used.
- **maxiter** (`int`, *optional*) – Passed into `scipy.optimize.newton()`.

Returns

`xarray.Dataset` – The nominal performance grid. The grid is indexed by altitude and Mach number. Contains the following variables:

- `"fuel_flow"` : Fuel flow rate, $[kg/s]$
- `"engine_efficiency"` : Engine efficiency
- `"aircraft_mass"` : Aircraft mass at which the engine efficiency is maximized, $[kg]$

Raises

`KeyError` – If "aircraft_type" is not supported by the PS model.

Examples

```
>>> level = np.arange(200, 300, 10, dtype=float)
```

```
>>> # Compute nominal aircraft performance assuming ISA conditions
>>> # and the design Mach number
>>> perf = ps_nominal_grid("A320", level=level)
>>> perf.attrs["mach_number"]
0.753
```

```
>>> perf.to_dataframe()
       aircraft_mass  engine_efficiency  fuel_flow
level
200.0   58416.230844           0.308675   0.561244
210.0   61617.676623           0.308675   0.589307
220.0   64829.702583           0.308675   0.617369
230.0   68026.415693           0.308675   0.646423
240.0   71187.897058           0.308675   0.677265
250.0   71818.514829           0.308542   0.686129
260.0   71809.073819           0.308073   0.690896
270.0   71796.095265           0.307367   0.696978
280.0   71780.225852           0.306500   0.704144
290.0   71761.957365           0.305529   0.712217
```

```
>>> # Now compute it for a higher Mach number
>>> perf = ps_nominal_grid("A320", level=level, mach_number=0.78)
>>> perf.to_dataframe()
       aircraft_mass  engine_efficiency  fuel_flow
level
200.0   57857.463750           0.314462   0.580472
210.0   60626.062062           0.314467   0.605797
220.0   63818.498305           0.314467   0.634645
230.0   66993.691515           0.314467   0.664512
240.0   70129.930503           0.314467   0.696217
250.0   71747.933832           0.314423   0.714997
260.0   71735.433936           0.314098   0.721147
270.0   71719.057287           0.313542   0.728711
280.0   71699.595538           0.312830   0.737412
290.0   71677.628782           0.312016   0.747045
```

11.3.7 Emissions

models.emissions.Emissions([met, params])	Emissions handling using ICAO Emissions Databank (EDB) and black carbon correlations.
models.emissions.black_carbon	Non-volatile particulate matter (nvPM) calculations.
models.emissions.ffm2	Calculate nitrogen oxide (NOx), carbon monoxide (CO) and hydrocarbon (HC) emissions.

pycontrails.models.emissions.Emissions

class pycontrails.models.emissions.**Emissions**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Emissions handling using ICAO Emissions Databank (EDB) and black carbon correlations.

Parameters

- **met** (`MetDataset | None`, *optional*) – Met data, by default None.
- **params** (`dict[str, Any] | None`, *optional*) – Model parameters, by default None.
- **params_kwargs** (`Any`) – Model parameters passed as keyword arguments.

References

- [Lee *et al.*, 2021]
- [Schumann *et al.*, 2015]
- [Stettler *et al.*, 2013]
- [Wilkerson *et al.*, 2010]

See also:

pycontrails.models.emissions.black_carbon, *pycontrails.models.emissions.ffm2*

__init__(*met=None*, *params=None*, ***params_kwargs*)

Methods

__init__([met, params])	
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
eval([source])	Calculate the emissions data for source.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is type_.
set_source([source])	Attach original or copy of input source to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to source.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Data evaluated in model
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
processed_met_variables	Set of required parameters if processing already complete on met input.
optional_met_variables	Optional meteorology variables

default_params

alias of EmissionsParams

eval(*source=None*, ***params*)

Calculate the emissions data for source.

Parameter source must contain each of the variables:

- air_temperature
- specific_humidity
- true_airspeed
- fuel_flow

If 'engine_uid' is not provided in source.attrs or not available in the ICAO EDB, constant emission indices will be assumed for NOx, CO, HC, and nvPM mass and number.

The computed pollutants include carbon dioxide (CO2), nitrogen oxide (NOx), carbon monoxide (CO), hydrocarbons (HC), non-volatile particulate matter (nvPM) mass and number, sulphur oxides (SOx), sulphates (S) and organic carbon (OC).

Changed in version 0.47.0: Support GeoVectorDataset for the source parameter.

Parameters

- **source** (*GeoVectorDataset*) – Flight to evaluate
- ****params** (Any) – Overwrite model parameters before eval

Returns

GeoVectorDataset – Flight with attached emissions data

long_name = 'ICAO Emissions Databank (EDB)'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'), MetVariable(short_name='q', standard_name='specific_humidity', long_name='Specific Humidity', level_type='isobaricInhPa', ecmwf_id=133, grib1_id=51, grib2_id=(0, 1, 0), units='kg kg-1', amip='hus', description='Specific means per unit mass. Specific humidity is the mass fraction of water vapor in (moist) air.'))**

Required meteorology pressure level variables. Each element in the list is a MetVariable or a tuple[MetVariable]. If element is a tuple[MetVariable], the variable depends on the data source. Only one variable in the tuple is required.

name = 'emissions'

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

pycontrails.models.emissions.black_carbon

Non-volatile particulate matter (nvPM) calculations.

Functions

air_to_fuel_ratio_imfox(thrust_setting)	Calculate the air-to-fuel ratio at cruise conditions via Abrahamson's method.
bc_mass_concentration_cruise_fox(c_bc_ref, ...)	Calculate the black carbon mass concentration for cruise conditions (`c_bc_cru`).
bc_mass_concentration_fox(fuel_flow, t_fl, afr)	Calculate the black carbon mass concentration for ground conditions (`c_bc_ref`).
bc_mass_concentration_imfox(...)	Calculate the BC mass concentration for ground and cruise conditions with ImFOX methodology.
bc_mass_emissions_index(c_bc, q_exhaust)	Calculate the black carbon mass emissions index.
dopelheuer_lecht_scaling_factor(t_fl_cru, ...)	Estimate scaling factor to convert the reference BC mass concentration from ground to cruise.
exhaust_gas_volume_per_kg_fuel(afr)	Calculate the volume of exhaust gas per mass of fuel burnt.
flame_temperature(t_3)	Calculate the flame temperature at the combustion chamber (t_fl).
geometric_mean_diameter_sac(air_pressure, ...)	Calculate the BC GMD for singular annular combustor (SAC) engines.
mass_emissions_index_fox(air_pressure, ...)	Calculate the black carbon mass emissions index using the Formation and Oxidation Method (FOX).
mass_emissions_index_imfox(...)	Calculate the BC mass EI using the "Improved" Formation and Oxidation Method (ImFOX).
number_emissions_index_fractal_aggregates(...)	Estimate the black carbon number emission index using the fractal aggregates (FA) model.
nvpm_mass_ei_pct_reduction_due_to_saf(...)	Adjust nvPM mass emissions index to account for the effects of sustainable aviation fuels.
nvpm_number_ei_pct_reduction_due_to_saf(...)	Adjust nvPM number emissions index to account for the effects of sustainable aviation fuels.
turbine_inlet_temperature_imfox(afr)	Calculate the turbine inlet temperature using Abrahamson's method.

pycontrails.models.emissions.black_carbon.**air_to_fuel_ratio_imfox**(*thrust_setting*)

Calculate the air-to-fuel ratio at cruise conditions via Abrahamson's method.

See Eq. (11) in [Abrahamson *et al.*, 2016].

Parameters

thrust_setting (`npt.NDArray[np.float64]`) – Engine thrust setting, which is the fuel mass flow rate divided by the maximum fuel mass flow rate

Returns

`npt.NDArray[np.float64]` – Air-to-fuel ratio at cruise conditions

References

- [Abrahamson *et al.*, 2016]

pycontrails.models.emissions.black_carbon.**bc_mass_concentration_cruise_fox**(*c_bc_ref*, *t_fl_cru*, *t_fl_ref*, *p_3_cru*, *p_3_ref*, *afr_cru*, *afr_ref*)

Calculate the black carbon mass concentration for cruise conditions (`c_bc_cru`).

This quantity is computed at the instrument sampling point without correcting for particle line losses.

Parameters

- **c_bc_ref** (`npt.NDArray[np.float64]`) – Black carbon mass concentration at reference conditions, $[mgm^{-3}]$
- **t_fl_cru** (`npt.NDArray[np.float64]`) – Flame temperature at cruise conditions, $[K]$
- **t_fl_ref** (`npt.NDArray[np.float64] | float`) – Flame temperature at reference conditions, $[K]$
- **p_3_cru** (`npt.NDArray[np.float64]`) – Combustor inlet pressure at cruise conditions, $[Pa]$
- **p_3_ref** (`npt.NDArray[np.float64] | float`) – Combustor inlet pressure at reference conditions, $[Pa]$
- **afr_cru** (`npt.NDArray[np.float64]`) – Air-to-fuel ratio at cruise conditions
- **afr_ref** (`npt.NDArray[np.float64] | float`) – Air-to-fuel ratio at reference conditions

Returns

`npt.NDArray[np.float64]` – Black carbon mass concentration for cruise conditions, $[mgm^{-3}]$

pycontrails.models.emissions.black_carbon.**bc_mass_concentration_fox**(*fuel_flow*, *t_fl*, *afr*)

Calculate the black carbon mass concentration for ground conditions (`c_bc_ref`).

This quantity is computed at the instrument sampling point without correcting for particle line losses.

Parameters

- **fuel_flow** (`npt.NDArray[np.float64]`) – Fuel mass flow rate, $[kgs^{-1}]$
- **t_fl** (`npt.NDArray[np.float64] | float`) – Flame temperature at the combustion chamber, $[K]$
- **afr** (`npt.NDArray[np.float64] | float`) – Air-to-fuel ratio

Returns

`npt.NDArray[np.float64]` – Black carbon mass concentration for ground conditions, $[mgm^{-3}]$

pycontrails.models.emissions.black_carbon.**bc_mass_concentration_imfox**(*fuel_flow_per_engine*, *afr*, *t_4*, *fuel_hydrogen*)

Calculate the BC mass concentration for ground and cruise conditions with ImFOX methodology.

This quantity is computed at the instrument sampling point without correcting for particle line losses.

Parameters

- **fuel_flow_per_engine** (`npt.NDArray[np.float64]`) – fuel mass flow rate per engine, $[kgs^{-1}]$

- **afr** (npt.NDArray[np.float64]) – air-to-fuel ratio
- **t_4** (npt.NDArray[np.float64]) – turbine inlet temperature, $[K]$
- **fuel_hydrogen** (float) – percentage of hydrogen mass content in the fuel (13.8% for conventional Jet A-1 fuel)

Returns
npt.NDArray[np.float64] – Black carbon mass concentration, $[mgm^{-3}]$

pycontrails.models.emissions.black_carbon.**bc_mass_emissions_index**(*c_bc*, *q_exhaust*)

Calculate the black carbon mass emissions index.

Parameters

- **c_bc** (npt.NDArray[np.float64]) – Black carbon mass concentration, $[mgm^{-3}]$
- **q_exhaust** (npt.NDArray[np.float64]) – Volume of exhaust gas per mass of fuel burnt, $[m^3/kg_{fuel}]$

Returns
npt.NDArray[np.float64] – Black carbon mass emissions index, $[mg/kg_{fuel}]$

References

- [Stettler *et al.*, 2013]

pycontrails.models.emissions.black_carbon.**dopelheuer_lecht_scaling_factor**(*t_fl_cru*, *t_fl_ref*, *p_3_cru*, *p_3_ref*, *afr_cru*, *afr_ref*)

Estimate scaling factor to convert the reference BC mass concentration from ground to cruise.

Parameters

- **t_fl_cru** (npt.NDArray[np.float64]) – Flame temperature at cruise conditions, $[K]$
- **t_fl_ref** (npt.NDArray[np.float64] | float) – Flame temperature at reference conditions, $[K]$
- **p_3_cru** (npt.NDArray[np.float64]) – Combustor inlet pressure at cruise conditions, $[Pa]$
- **p_3_ref** (npt.NDArray[np.float64] | float) – Combustor inlet pressure at reference conditions, $[Pa]$
- **afr_cru** (npt.NDArray[np.float64]) – Air-to-fuel ratio at cruise conditions
- **afr_ref** (npt.NDArray[np.float64] | float) – Air-to-fuel ratio at reference conditions

Returns
npt.NDArray[np.float64] – Dopelheuer & Lecht scaling factor

References

- [Dopelheuer and Lecht, 1998]

pycontrails.models.emissions.black_carbon.**exhaust_gas_volume_per_kg_fuel**(*afr*)

Calculate the volume of exhaust gas per mass of fuel burnt.

Parameters
afr (`npt.NDArray[np.float64]`) – Air-to-fuel ratio

Returns
`npt.NDArray[np.float64]` – Volume of exhaust gas per mass of fuel burnt, $[m^3/kg_{fuel}]$

References

- [Stettler *et al.*, 2013]

pycontrails.models.emissions.black_carbon.**flame_temperature**(*t_3*)

Calculate the flame temperature at the combustion chamber (t_fl).

Parameters
t_3 (*ArrayScalarLike*) – Combustor inlet temperature, $[K]$

Returns
ArrayScalarLike – Flame temperature at the combustion chamber, $[K]$

pycontrails.models.emissions.black_carbon.**geometric_mean_diameter_sac**(*air_pressure*, *air_temperature*, *true_airspeed*, *thrust_setting*, *pressure_ratio*, *q_fuel*, ***, *comp_efficiency=0.9*, *delta_loss=5.75*, *cruise=True*)

Calculate the BC GMD for singular annular combustor (SAC) engines.

The BC (black carbon) GMD (geometric mean diameter) is estimated using the non-dimensionalized engine thrust setting, the ratio of turbine inlet to the compressor inlet temperature (`t4_t2`).

Parameters

- **air_pressure** (`npt.NDArray[np.float64]`) – Pressure altitude at each waypoint, $[Pa]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed for each waypoint, $[ms^{-1}]$
- **thrust_setting** (`npt.NDArray[np.float64]`) – Engine thrust setting, which is the fuel mass flow rate divided by the maximum fuel mass flow rate
- **pressure_ratio** (`float`) – Engine pressure ratio from the ICAO EDB
- **q_fuel** (`float`) – Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$
- **comp_efficiency** (`float`) – Engine compressor efficiency (assumed to be 0.9)
- **delta_loss** (`float`) – Correction factor accounting for particle line losses (assumed to be 5.75 nm), $[nm]$

- **cruise** (bool) – Set to true when the aircraft is not on the ground.

Returns

npt.NDArray[np.float64] – black carbon geometric mean diameter, [nm]

References

- [Teoh *et al.*, 2020]

pycontrails.models.emissions.black_carbon.**mass_emissions_index_fox**(*air_pressure*, *air_temperature*, *true_airspeed*, *fuel_flow_per_engine*, *thrust_setting*, *pressure_ratio*, *, *comp_efficiency=0.9*)

Calculate the black carbon mass emissions index using the Formation and Oxidation Method (FOX).

Parameters

- **air_pressure** (npt.NDArray[np.float64]) – Pressure altitude at each waypoint, [Pa]
- **air_temperature** (npt.NDArray[np.float64]) – Ambient temperature for each waypoint, [K]
- **true_airspeed** (npt.NDArray[np.float64]) – True airspeed for each waypoint, [ms^{-1}]
- **fuel_flow_per_engine** (npt.NDArray[np.float64]) – Fuel mass flow rate per engine, [kgs^{-1}]
- **thrust_setting** (npt.NDArray[np.float64]) – Engine thrust setting, which is the fuel mass flow rate divided by the maximum fuel mass flow rate
- **pressure_ratio** (float) – Engine pressure ratio from the ICAO EDB
- **comp_efficiency** (float) – Engine compressor efficiency, assumed to be 0.9

Returns

npt.NDArray[np.float64] – Black carbon mass emissions index, [$mg\ kg_{fuel}^{-1}$]

References

- [Stettler *et al.*, 2013]

pycontrails.models.emissions.black_carbon.**mass_emissions_index_imfox**(*fuel_flow_per_engine*, *thrust_setting*, *fuel_hydrogen*)

Calculate the BC mass EI using the "Improved" Formation and Oxidation Method (ImFOX).

Parameters

- **fuel_flow_per_engine** (npt.NDArray[np.float64]) – Fuel mass flow rate per engine, [kgs^{-1}]
- **thrust_setting** (npt.NDArray[np.float64]) – Engine thrust setting, which is the fuel mass flow rate divided by the maximum fuel mass flow rate
- **fuel_hydrogen** (float) – Percentage of hydrogen mass content in the fuel (13.8% for conventional Jet A-1 fuel)

Returns

`npt.NDArray[np.float64]` – Black carbon mass emissions index, $[mg\ kg_{fuel}^{-1}]$

References

- [Abrahamson *et al.*, 2016]

pycontrails.models.emissions.black_carbon.**number_emissions_index_fractal_aggregates**(*nvpm_ei_m*, *gmd*, ***, *gsd=1.8*, *rho_bc=1770*, *k_tem=1.621e-05*, *d_tem=0.39*, *d_fm=2.76*)

Estimate the black carbon number emission index using the fractal aggregates (FA) model.

The FA model estimates the number emissions index from the mass emissions index, particle size distribution, and morphology.

Parameters

- **nvpm_ei_m** (`npt.NDArray[np.float64]`) – Black carbon mass emissions index, $[kg/kg_{fuel}]$
- **gmd** (`npt.NDArray[np.float64]`) – Black carbon geometric mean diameter, $[m]$
- **gsd** (`float`) – Black carbon geometric standard deviation (assumed to be 1.80)
- **rho_bc** (`float`) – Black carbon material density (1770 kg/m**3), $[kgm^{-3}]$
- **k_tem** (`float`) – Transmission electron microscopy prefactor coefficient (assumed to be 1.621e-5)
- **d_tem** (`float`) – Transmission electron microscopy exponent coefficient (assumed to be 0.39)
- **d_fm** (`float`) – Mass-mobility exponent (assumed to be 2.76)

Returns

`npt.NDArray[np.float64]` – Black carbon number emissions index, $[kg_{fuel}^{-1}]$

References

- FA model: [Teoh *et al.*, 2019]
- `gmd`, `gsd`, `d_fm`: [Teoh *et al.*, 2020]
- `rho_bc`: [Park *et al.*, 2004]
- `k_tem`, `d_tem`: [Dastanpour and Rogak, 2014]

pycontrails.models.emissions.black_carbon.**nvpm_mass_ei_pct_reduction_due_to_saf**(*hydrogen_content*, *thrust_setting*)

Adjust nvPM mass emissions index to account for the effects of sustainable aviation fuels.

For fuel with hydrogen mass content > 14.3, the adjustment factor is adopted from Teoh et al. (2022), which was used to calculate the change in nvPM EIn.

Parameters

- **hydrogen_content** (`float`) – The percentage of hydrogen mass content in the fuel.
- **thrust_setting** (`npt.NDArray[np.float64]`) – Engine thrust setting, where the equivalent fuel mass flow rate per engine at sea level, $[0 - 1]$.

Returns

`npt.NDArray[np.float64]` – Percentage reduction in nvPM number emissions index

References

- [Teoh *et al.*, 2022]
- [Brem *et al.*, 2015]

pycontrails.models.emissions.black_carbon.**nvpm_number_ei_pct_reduction_due_to_saf**(*hydrogen_content*, *thrust_setting*)

Adjust nvPM number emissions index to account for the effects of sustainable aviation fuels.

Parameters

- **hydrogen_content** (`float`) – The percentage of hydrogen mass content in the fuel.
- **thrust_setting** (`npt.NDArray[np.float64]`) – Engine thrust setting, where the equivalent fuel mass flow rate per engine at sea level, $[0 - 1]$.

Returns

`npt.NDArray[np.float64]` – Percentage reduction in nvPM number emissions index

References

- [Teoh *et al.*, 2022]
- [Brem *et al.*, 2015]

pycontrails.models.emissions.black_carbon.**turbine_inlet_temperature_imfox**(*afr*)

Calculate the turbine inlet temperature using Abrahamson's method.

See Eq. (13) in [Abrahamson *et al.*, 2016].

Parameters

afr (`npt.NDArray[np.float64]`) – air-to-fuel ratio at cruise conditions

Returns

`npt.NDArray[np.float64]` – turbine inlet temperature, $[K]$

References

- [Abrahamson *et al.*, 2016]

pycontrails.models.emissions.ffm2

Calculate nitrogen oxide (NOx), carbon monoxide (CO) and hydrocarbon (HC) emissions.

This modules applies the Fuel Flow Method 2 (FFM2) from DuBois & Paynter (2006) for a given aircraft-engine pair.

References

- [DuBois and Paynter, 2006]

Functions

co_hc_emissions_index_profile(ff_idle, ...)	Create carbon monoxide (CO) and hydrocarbon (HC) emissions index (EI) profile.
ei_at_cruise(ei_sl, theta_amb, delta_amb, ...)	Convert the estimated EI at sea level to cruise conditions.
estimate_ei(log_ei_profile, ...)	Estimate carbon monoxide (CO) or hydrocarbon (HC) emissions index (EI).
estimate_nox(log_ei_nox_profile, ...[, ...])	Estimate the nitrogen oxide (NOx) emissions index (EI) at cruise conditions.
nitrogen_oxide_emissions_index_profile(...)	Create the nitrogen oxide (NOx) emissions index (EI) profile for the given engine type.

pycontrails.models.emissions.ffm2.**co_hc_emissions_index_profile**(*ff_idle*, *ff_approach*, *ff_climb*, *ff_take_off*, *ei_idle*, *ei_approach*, *ei_climb*, *ei_take_off*)

Create carbon monoxide (CO) and hydrocarbon (HC) emissions index (EI) profile.

Parameters

- **ff_idle** (float) – ICAO EDB fuel mass flow rate at idle conditions (7% power), $[kgs^{-1}]$
- **ff_approach** (float) – ICAO EDB fuel mass flow rate at approach (30% power), $[kgs^{-1}]$
- **ff_climb** (float) – ICAO EDB fuel mass flow rate at climb out (85% power), $[kgs^{-1}]$
- **ff_take_off** (float) – ICAO EDB fuel mass flow rate at take-off (100% power), $[kgs^{-1}]$
- **ei_idle** (float) – ICAO EDB CO or HC emissions index at idle conditions (7% power), $[g_{pollutant}/kg_{fuel}]$
- **ei_approach** (float) – ICAO EDB CO or HC emissions index at approach (30% power), $[g_{pollutant}/kg_{fuel}]$
- **ei_climb** (float) – ICAO EDB CO or HC emissions index at climb out (85% power), $[g_{pollutant}/kg_{fuel}]$
- **ei_take_off** (float) – ICAO EDB CO or HC emissions index at take-off (100% power), $[g_{pollutant}/kg_{fuel}]$

Returns

EmissionsProfileInterpolator – log of CO or HC emissions index versus the log of fuel mass flow rate for a given engine type

pycontrails.models.emissions.ffm2.**ei_at_cruise**(*ei_sl*, *theta_amb*, *delta_amb*, *ei_type*)

Convert the estimated EI at sea level to cruise conditions.

Refer to Eqs. (15) and (16) in DuBois & Paynter (2006).

Parameters

- **ei_sl** (`npt.NDArray[np.float64]`) – Sea level EI values.
- **theta_amb** (`npt.NDArray[np.float64]`) – Ratio of the ambient temperature to the temperature at mean sea-level.
- **delta_amb** (`npt.NDArray[np.float64]`) – Ratio of the pressure altitude to the surface pressure.
- **ei_type** (`str`) – One of {"HC", "CO", "NOX"}

Returns

`npt.NDArray[np.float64]` – Estimated cruise EI values.

References

- [DuBois and Paynter, 2006]

`pycontrails.models.emissions.ffm2.`**`estimate_ei`**(*log_ei_profile*, *fuel_flow_per_engine*, *true_airspeed*, *air_pressure*, *air_temperature*)

Estimate carbon monoxide (CO) or hydrocarbon (HC) emissions index (EI).

Parameters

- **log_ei_profile** (`EmissionsProfileInterpolator`) – emissions profile containing the log of EI CO or EI HC versus log of fuel flow.
- **fuel_flow_per_engine** (`npt.NDArray[np.float64]`) – fuel mass flow rate per engine, $[kgs^{-1}]$
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, $[ms^{-1}]$
- **air_pressure** (`npt.NDArray[np.float64]`) – pressure altitude at each waypoint, $[Pa]$
- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature for each waypoint, $[K]$

`pycontrails.models.emissions.ffm2.`**`estimate_nox`**(*log_ei_nox_profile*, *fuel_flow_per_engine*, *true_airspeed*, *air_pressure*, *air_temperature*, *specific_humidity=None*)

Estimate the nitrogen oxide (NOx) emissions index (EI) at cruise conditions.

Parameters

- **log_ei_nox_profile** (`EmissionsProfileInterpolator`) – emissions profile containing the log of EI NOx versus log of fuel flow.
- **fuel_flow_per_engine** (`npt.NDArray[np.float64]`) – fuel mass flow rate per engine, $[kgs^{-1}]$
- **true_airspeed** (`npt.NDArray[np.float64]`) – true airspeed for each waypoint, $[ms^{-1}]$
- **air_pressure** (`npt.NDArray[np.float64]`) – pressure altitude at each waypoint, $[Pa]$
- **air_temperature** (`npt.NDArray[np.float64]`) – ambient temperature for each waypoint, $[K]$
- **specific_humidity** (`npt.NDArray[np.float64] | None`) – specific humidity for each waypoint, $[kg_{H_2O}/kg_{air}]$

pycontrails.models.emissions.ffm2.**nitrogen_oxide_emissions_index_profile**(*ff_idle*, *ff_approach*, *ff_climb*, *ff_take_off*, *ei_nox_idle*, *ei_nox_approach*, *ei_nox_climb*, *ei_nox_take_off*)

Create the nitrogen oxide (NOx) emissions index (EI) profile for the given engine type.

Parameters

- **ff_idle** (float) – ICAO EDB fuel mass flow rate at idle conditions (7% power), [kgs^{-1}]
- **ff_approach** (float) – ICAO EDB fuel mass flow rate at approach (30% power), [kgs^{-1}]
- **ff_climb** (float) – ICAO EDB fuel mass flow rate at climb out (85% power), [kgs^{-1}]
- **ff_take_off** (float) – ICAO EDB fuel mass flow rate at take-off (100% power), [kgs^{-1}]
- **ei_nox_idle** (float) – ICAO EDB NOx emissions index at idle conditions (7% power), [g_{NO_X}/kg_{fuel}]
- **ei_nox_approach** (float) – ICAO EDB NOx emissions index at approach (30% power), [g_{NO_X}/kg_{fuel}]
- **ei_nox_climb** (float) – ICAO EDB NOx emissions index at climb out (85% power), [g_{NO_X}/kg_{fuel}]
- **ei_nox_take_off** (float) – ICAO EDB NOx emissions index at take-off (100% power), [g_{NO_X}/kg_{fuel}]

Returns

EmissionsProfileInterpolator – log of NOx emissions index versus the log of fuel mass flow rate for a given engine type

Raises

ValueError – If any EI nox values are non-positive.

11.3.8 Cirrus Optical Depth (τ_{cirrus})

models.tau_cirrus	Calculate tau cirrus on Met data.

pycontrails.models.tau_cirrus

Calculate tau cirrus on Met data.

Functions

cirrus_effective_extinction_coef(ciwc, T, p)	Calculate the effective extinction coefficient for spectral range 0.2-0.69 um.
tau_cirrus(met)	Calculate the optical depth of NWP cirrus around each pressure level.

pycontrails.models.tau_cirrus.**cirrus_effective_extinction_coef**(*ciwc*, *T*, *p*)

Calculate the effective extinction coefficient for spectral range 0.2-0.69 um.

Parameters

- **ciwc** (*ArrayLike*) – Cloud ice water content, [$kg_{ice} kg^{-1}_{dry\ air}$]. Note that ECMWF provides specific ice water content per mass *moist* air.
- **T** (*ArrayLike*) – Air temperature, [K]
- **p** (*ArrayLike*) – Air pressure, [Pa]

Returns

ArrayLike – Effective extinction coefficient for spectral range 0.2-0.69 um, [m^{-1}]

References

- [Schumann, 2012]
- [Sun and Rikus, 1999]

Notes

References as noted in [Schumann, 2012]:

- Sun and Rikus QJRMS (1999), 125, 3037-3055
- Sun QJRMS (2001), 127, 267-271
- McFarquhar QJRMS (2001), 127, 261-265

pycontrails.models.tau_cirrus.**tau_cirrus**(*met*)

Calculate the optical depth of NWP cirrus around each pressure level.

Parameters

met (*MetDataset*) – A MetDataset with the following variables: - "air_temperature" - "specific_cloud_ice_water_content" or "ice_water_mixing_ratio"

Returns

xarray.DataArray – Array of tau cirrus values. Has the same dimensions as the input data.

Notes

Implementation differs from original Fortran implementation in computing the vertical derivative of geopotential height. In particular, the finite difference at the top-most and bottom-most layers different from original calculation by a factor of two. The implementation here is consistent with a numerical approximation of the derivative.

11.3.9 Humidity Scaling

models.humidity_scaling	Humidity scaling methodologies.

pycontrails.models.humidity_scaling

Humidity scaling methodologies.

class pycontrails.models.humidity_scaling.**ConstantHumidityScaling**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *HumidityScaling*

Scale specific humidity by applying a constant uniform scaling.

This scalar simply applies the transformation..

rhi -> rhi / rhi_adj

where `rhi_adj` is a constant specified by *params* or overridden by a variable or attribute on `source` in `eval()`.

The convention to divide by `rhi_adj` instead of considering the more natural product `rhi_adj * rhi` is somewhat arbitrary. In short, `rhi_adj` can be thought of as the critical threshold for supersaturation.

References

- [Schumann, 2012]
- [Reutter *et al.*, 2020]

default_params

alias of *ConstantHumidityScalingParams*

formula = 'rhi -> rhi / rhi_adj'

long_name = 'Constant humidity scaling'

met

Meteorology data

name = 'constant_scale'

params

Instantiated model parameters, in dictionary form

scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Compute scaled specific humidity and RHi.

See docstring for the implementing subclass for specific methodology.

Parameters

- **specific_humidity** (*ArrayLike*) – Unscaled specific relative humidity, $[kg\ kg^{-1}]$. Typically, this is interpolated meteorology data.
- **air_temperature** (*ArrayLike*) – Air temperature, $[K]$. Typically, this is interpolated meteorology data.
- **air_pressure** (*ArrayLike*) – Pressure, $[Pa]$

- **kwargs** (*ArrayLike*) – Other keyword-only variables and model parameters used by the formula.

Returns

- **specific_humidity** (*ArrayLike*) – Scaled specific humidity.
- **rhi** (*ArrayLike*) – Scaled relative humidity over ice.

See also:

`eval()`

scaler_specific_keys = ('rhi_adj',)

Variables required in addition to specific_humidity, air_temperature, and air_pressure These are either `ModelParams` specific to scaling, or variables that should be extracted from `eval()` parameter `source`.

source

Data evaluated in model

class pycontrails.models.humidity_scaling.**ConstantHumidityScalingParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *rhi_adj=0.97*)

Bases: *ModelParams*

Parameters for *ConstantHumidityScaling*.

rhi_adj = 0.97

Scale specific humidity by dividing it with adjustment factor per [Schumann, 2012] eq. (9). Set to a constant 0.9 in [Schumann, 2012] to account for sub-scale variability of specific humidity. A value of 1.0 provides no scaling.

class pycontrails.models.humidity_scaling.**ExponentialBoostHumidityScaling**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *HumidityScaling*

Scale humidity by composing constant scaling with exponential boosting.

This formula composes the transformations

1. constant scaling: `rhi -> rhi / rhi_adj`
2. exponential boosting: `rhi -> rhi ^ rhi_boost_exponent if rhi > 1`
3. clipping: `rhi -> min(rhi, clip_upper)`

where `rhi_adj`, `rhi_boost_exponent`, and `clip_upper` are model *params*.

References

- [Teoh *et al.*, 2022]

See also:

ExponentialBoostLatitudeCorrectionHumidityScaling

default_params

alias of *ExponentialBoostHumidityScalingParams*

formula = 'rhi -> (rhi / rhi_adj) ^ rhi_boost_exponent'

long_name = 'Constant humidity scaling composed with exponential boosting'

met

Meteorology data

name = 'exponential_boost'

params

Instantiated model parameters, in dictionary form

scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Compute scaled specific humidity and RHi.

See docstring for the implementing subclass for specific methodology.

Parameters

- **specific_humidity** (*ArrayLike*) – Unscaled specific relative humidity, $[kg\ kg^{-1}]$. Typically, this is interpolated meteorology data.
- **air_temperature** (*ArrayLike*) – Air temperature, $[K]$. Typically, this is interpolated meteorology data.
- **air_pressure** (*ArrayLike*) – Pressure, $[Pa]$
- **kwargs** (*ArrayLike*) – Other keyword-only variables and model parameters used by the formula.

Returns

- **specific_humidity** (*ArrayLike*) – Scaled specific humidity.
- **rhi** (*ArrayLike*) – Scaled relative humidity over ice.

See also:

`eval()`

scaler_specific_keys = ('rhi_adj', 'rhi_boost_exponent', 'clip_upper')

Variables required in addition to specific_humidity, air_temperature, and air_pressure These are either `ModelParams` specific to scaling, or variables that should be extracted from `eval()` parameter `source`.

source

Data evaluated in model

class pycontrails.models.humidity_scaling.**ExponentialBoostHumidityScalingParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *rhi_adj=0.97*, *rhi_boost_exponent=1.7*, *clip_upper=1.7*)

Bases: *ConstantHumidityScalingParams*

Parameters for *ExponentialBoostHumidityScaling*.

clip_upper = 1.7

Used to clip overinflated unrealistic RHi values.

rhi_boost_exponent = 1.7

Boost RHi values exceeding 1 as described in [Teoh *et al.*, 2022]. In `eval()`, this can be overridden by a keyword argument with the same name.

class pycontrails.models.humidity_scaling.**ExponentialBoostLatitudeCorrectionHumidityScaling**(*met=None*, *params=None*, ***params_kwa*

Bases: *HumidityScaling*

Correct RHi values derived from ECMWF ERA5 HRES.

Unlike other RHi correction factors, this function applies a custom latitude-based term and has been tuned for global application.

This formula composes the transformations

1. constant scaling: `rhi -> rhi / rhi_adj`
2. exponential boosting: `rhi -> rhi ^ rhi_boost_exponent if rhi > 1`
3. clipping: `rhi -> min(rhi, rhi_max)`

where `rhi_adj` and `rhi_boost_exponent` depend on `latitude` to minimize error between ERA5 HRES and IAGOS in-situ data.

For each waypoint, `rhi_max` ensures that the corrected RHi does not exceed the maximum value according to thermodynamics:

- `rhi_max = p_liq(T) / p_ice(T)` for `T > 235 K`, (Pruppacher and Klett, 1997)
- `rhi_max = 1.67 + (1.45 - 1.67) * (T - 190.) / (235. - 190.)` for `T < 235 K` (Karcher and Lohmann, 2002; Tompkins et al., 2007)

The RHi correction addresses the known limitations of the ERA5 HRES humidity fields, ensuring that the ISSR coverage area and RHi-distribution is consistent with in-situ measurements from the IAGOS dataset. Generally, the correction:

1. reduces the ISSR coverage area near the equator,
2. increases the ISSR coverage area at higher latitudes, and
3. accounts for localized regions with very high ice supersaturation (RHi > 120%).

This methodology is an extension of Teoh et al. (2022) and has not yet been peer-reviewed/published.

The ERA5 HRES <> IAGOS fitting uses a sigmoid curve to capture significant changes in tropopause height at 20 - 50 degrees latitude.

The method `eval()` requires a `latitude` keyword argument.

References

- [Teoh *et al.*, 2022]
- Kärcher, B. and Lohmann, U., 2002. A parameterization of cirrus cloud formation: Homogeneous freezing of supercooled aerosols. Journal of Geophysical Research: Atmospheres, 107(D2), pp.AAC-4.
- Pruppacher, H.R. and Klett, J.D. (1997) Microphysics of Clouds and Precipitation. 2nd Edition, Kluwer Academic, Dordrecht, 954 p.
- Tompkins, A.M., Gierens, K. and Rädel, G., 2007. Ice supersaturation in the ECMWF integrated forecast system. Quarterly Journal of the Royal Meteorological Society: A journal of the atmospheric sciences, applied meteorology and physical oceanography, 133(622), pp.53-63.

See also:

ExponentialBoostHumidityScaling

default_params

alias of *ModelParams*

formula = 'rhi -> (rhi / rhi_adj) ^ rhi_boost_exponent'

long_name = 'Latitude specific humidity scaling composed with exponential boosting'

met

Meteorology data

name = 'exponential_boost_latitude_customization'

params

Instantiated model parameters, in dictionary form

scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Compute scaled specific humidity and RHi.

See docstring for the implementing subclass for specific methodology.

Parameters

- **specific_humidity** (*ArrayLike*) – Unscaled specific relative humidity, [$kg\ kg^{-1}$]. Typically, this is interpolated meteorology data.
- **air_temperature** (*ArrayLike*) – Air temperature, [K]. Typically, this is interpolated meteorology data.
- **air_pressure** (*ArrayLike*) – Pressure, [Pa]
- **kwargs** (*ArrayLike*) – Other keyword-only variables and model parameters used by the formula.

Returns

- **specific_humidity** (*ArrayLike*) – Scaled specific humidity.
- **rhi** (*ArrayLike*) – Scaled relative humidity over ice.

See also:

`eval()`

scaler_specific_keys = ('latitude',)

Variables required in addition to specific_humidity, air_temperature, and air_pressure These are either `ModelParams` specific to scaling, or variables that should be extracted from `eval()` parameter `source`.

source

Data evaluated in model

class pycontrails.models.humidity_scaling.**HistogramMatching**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *HumidityScaling*

Scale humidity by histogram matching to IAGOS RHi quantiles.

default_params

alias of *HistogramMatchingParams*

formula = 'era5_quantiles -> iagos_quantiles'

long_name = 'IAGOS RHi histogram matching'

met

Meteorology data

name = 'histogram_matching'

params

Instantiated model parameters, in dictionary form

scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Compute scaled specific humidity and RHi.

See docstring for the implementing subclass for specific methodology.

Parameters

- **specific_humidity** (*ArrayLike*) – Unscaled specific relative humidity, $[kg\ kg^{-1}]$. Typically, this is interpolated meteorology data.
- **air_temperature** (*ArrayLike*) – Air temperature, $[K]$. Typically, this is interpolated meteorology data.
- **air_pressure** (*ArrayLike*) – Pressure, $[Pa]$
- **kwargs** (*ArrayLike*) – Other keyword-only variables and model parameters used by the formula.

Returns

- **specific_humidity** (*ArrayLike*) – Scaled specific humidity.
- **rhi** (*ArrayLike*) – Scaled relative humidity over ice.

See also:

`eval()`

source

Data evaluated in model

class pycontrails.models.humidity_scaling.**HistogramMatchingParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *product_type='reanalysis'*, *level_type='pressure'*, *member=None*)

Bases: *ModelParams*

Parameters for *HistogramMatching*.

level_type = 'pressure'

The ERA5 vertical level type. Must be one of `"pressure"` or `"model"`.

member = None

The ERA5 ensemble member to use. Must be in the range `[0, 10)`. Only used if `product_type` is `"ensemble_members"`.

product_type = 'reanalysis'

The ERA5 product. Must be one of `"reanalysis"` or `"ensemble_members"`.

class pycontrails.models.humidity_scaling.**HistogramMatchingWithEckel**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *HumidityScaling*

Scale humidity by histogram matching to IAGOS RHi quantiles.

This method also applies the Eckel scaling to the recalibrated RHi values.

Unlike other specific humidity scaling methods, this method requires met data and performs interpolation at evaluation time.

Warning: Experimental. This may change or be removed in a future release.

References

[Eckel and Walters, 1998]

default_params

alias of *HistogramMatchingWithEckelParams*

eval(*source=None*, ***params*)

Scale specific humidity by histogram matching to IAGOS RHi quantiles.

This method assumes `source` is equipped with the following variables:

- air_temperature
- specific_humidity: Humidity values for the `params["member"]` ERA5 ensemble member.

formula = 'era5_quantiles -> iagos_quantiles -> recalibrated_rhi'

long_name = 'IAGOS RHi histogram matching with Eckel scaling'

met

Meteorology data

n_members = 10

name = 'histogram_matching_with_eckel'

params

Instantiated model parameters, in dictionary form

scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Scale specific humidity values via histogram matching and Eckel scaling.

Unlike the method on the base class, the method assumes each of the input arrays are `np.ndarray` and not `xr.DataArray` objects.

Parameters

- **specific_humidity** (npt.NDArray[np.float64]) – A 2D array of specific humidity values for all ERA5 ensemble members. The shape of this array must be (n, 10), where n is the number of observations and 10 is the number of ERA5 ensemble members.
- **air_temperature** (npt.NDArray[np.float64]) – A 1D array of air temperature values with shape (n,).
- **air_pressure** (npt.NDArray[np.float64]) – A 1D array of air pressure values with shape (n,).
- **kwargs** (Any) – Unused, kept for compatibility with the base class.

Returns

- **specific_humidity** (npt.NDArray[np.float64]) – The recalibrated specific humidity values. A 1D array with shape (n,).
- **rhi** (npt.NDArray[np.float64]) – The recalibrated RHi values. A 1D array with shape (n,).

source

Data evaluated in model

class pycontrails.models.humidity_scaling.**HistogramMatchingWithEckelParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *ensemble_specific_humidity=None*, *member=None*, *log_applied=False*)

Bases: *ModelParams*

Parameters for *HistogramMatchingWithEckel*.

> **Warning:** Experimental. This may change or be removed in a future release.

ensemble_specific_humidity = None

A length-10 list of ERA5 ensemble members. Each element is a `MetDataArray` holding specific humidity values for a single ensemble member. If None, a ValueError will be raised at model instantiation time. The order of the list must be consistent with the order of the ERA5 ensemble members.

log_applied = False

If a log transform has already been applied to each member of `ensemble_specific_humidity`, set this to True.

member = None

The specific member used. Must be in the range [0, 10). If None, a ValueError will be raised at model instantiation time.

class pycontrails.models.humidity_scaling.**HumidityScaling**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *Model*

Support for standardizing humidity scaling methodologies.

The method *scale()* or *eval()* should be called immediately after interpolation over meteorology data.

New in version 0.27.0.

property description

Get description for instance.

eval(*source=None*, ***params*)

Scale specific humidity values on `source`.

This method mutates the parameter `source` by modifying its "specific_humidity" variable and by attaching an "rhi" variable. Set model parameter `copy_source=True` to avoid mutating `source`.

Parameters

- **source** (`GeoVectorDataset | MetDataset`) – Data with variables "specific_humidity", "air_temperature", and any variables defined by *scaler_specific_keys*.
- ****params** (`Any`) – Overwrite model parameters before eval

Returns

`GeoVectorDataset | MetDataset` – Source data with updated "specific_humidity" and "rhi". If `source` is `GeoVectorDataset`, "air_pressure" data is also attached.

See also:

scale()

abstract property formula

Serializable formula for humidity scaler.

met

Meteorology data

params

Instantiated model parameters, in dictionary form

abstract scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Compute scaled specific humidity and RHi.

See docstring for the implementing subclass for specific methodology.

Parameters

- **specific_humidity** (*ArrayLike*) – Unscaled specific relative humidity, [$kg\ kg^{-1}$]. Typically, this is interpolated meteorology data.
- **air_temperature** (*ArrayLike*) – Air temperature, [K]. Typically, this is interpolated meteorology data.
- **air_pressure** (*ArrayLike*) – Pressure, [Pa]
- **kwargs** (*ArrayLike*) – Other keyword-only variables and model parameters used by the formula.

Returns

- **specific_humidity** (*ArrayLike*) – Scaled specific humidity.
- **rhi** (*ArrayLike*) – Scaled relative humidity over ice.

See also:

eval()

scaler_specific_keys = ()

Variables required in addition to specific_humidity, air_temperature, and air_pressure These are either `ModelParams` specific to scaling, or variables that should be extracted from *eval()* parameter `source`.

source

Data evaluated in model

property to_json

Get description for instance.

class pycontrails.models.humidity_scaling.**HumidityScalingByLevel**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *HumidityScaling*

Apply custom scaling to specific_humidity by pressure level.

This implements the original humidity scaling scheme suggested in [Schumann, 2012]. In particular, see eq. (9) and the surrounding text, quoted below.

> Hence, the critical value RHic is usually taken different and below 100% in NWP models. In the ECMWF model, this value is..
>
> RHic = 0.8, (9)
>
> in the mid-troposphere, 1.0 in the stratosphere and follows a smooth transition with pressure altitude between these two values in the upper 20 % of the troposphere. For simplicity of further analysis, we divide the input value of q by RHic initially.

See *ConstantHumidityScaling* for the simple method described above.

The diagram below shows the piecewise-linear `rhi_adj` factor by level. In particular, `rhi_adj` is constant at the stratosphere and above, linearly changes from the mid-troposphere to the stratosphere, and is constant at the mid-troposphere and below.

```
             _________  stratosphere rhi_adj = 1.0
           /
          /
         /
_________/  mid-troposphere rhi_adj = 0.8
```

References

- [Schumann, 2012]

default_params

alias of *HumidityScalingByLevelParams*

formula = 'rhi -> rhi / rhi_adj'

long_name = 'Constant humidity scaling by level'

met

Meteorology data

name = 'constant_scale_by_level'

params

Instantiated model parameters, in dictionary form

scale(*specific_humidity*, *air_temperature*, *air_pressure*, ***kwargs*)

Compute scaled specific humidity and RHi.

See docstring for the implementing subclass for specific methodology.

Parameters

- **specific_humidity** (*ArrayLike*) – Unscaled specific relative humidity, [$kg\ kg^{-1}$]. Typically, this is interpolated meteorology data.
- **air_temperature** (*ArrayLike*) – Air temperature, [K]. Typically, this is interpolated meteorology data.
- **air_pressure** (*ArrayLike*) – Pressure, [Pa]
- **kwargs** (*ArrayLike*) – Other keyword-only variables and model parameters used by the formula.

Returns

- **specific_humidity** (*ArrayLike*) – Scaled specific humidity.
- **rhi** (*ArrayLike*) – Scaled relative humidity over ice.

See also:

`eval()`

scaler_specific_keys = ('rhi_adj_mid_troposphere', 'rhi_adj_stratosphere', 'mid_troposphere_threshold', 'stratosphere_threshold')

Variables required in addition to specific_humidity, air_temperature, and air_pressure These are either `ModelParams` specific to scaling, or variables that should be extracted from `eval()` parameter `source`.

source

Data evaluated in model

class pycontrails.models.humidity_scaling.**HumidityScalingByLevelParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *rhi_adj_mid_troposphere=0.8*, *rhi_adj_stratosphere=1.0*, *mid_troposphere_threshold=0.8*, *stratosphere_threshold=1.0*)

Bases: *ModelParams*

Parameters for *HumidityScalingByLevel*.

mid_troposphere_threshold = 0.8

Adjustment factor for mid-troposphere humidity scaling. Default value of 0.8 taken from [Schumann, 2012].

rhi_adj_mid_troposphere = 0.8

Fraction of troposphere for mid-troposphere humidity scaling. Default value suggested in [Schumann, 2012].

rhi_adj_stratosphere = 1.0

Fraction of troposphere for stratosphere humidity scaling. Default value suggested in [Schumann, 2012].

stratosphere_threshold = 1.0

Adjustment factor for stratosphere humidity scaling. Default value of 1.0 taken from [Schumann, 2012].

pycontrails.models.humidity_scaling.**eckel_scaling**(*ensemble_mean_rhi*, *ensemble_member_rhi*, *q_method*)

Apply Eckel scaling to the given RHi values.

Parameters

- **ensemble_mean_rhi** (`npt.NDArray[np.float64]`) – The ensemble mean RHi values. This should be a 1D array with the same shape as `ensemble_member_rhi`.
- **ensemble_member_rhi** (`npt.NDArray[np.float64]`) – The RHi values for a single ensemble member.
- **q_method** (`{None, "cubic-spline", "log-q-log-p"}`) – The interpolation method.

Returns

`npt.NDArray[np.float64]` – The scaled RHi values. Values are manually clipped at 0 to ensure only non-negative values are returned.

References

[Eckel and Walters, 1998]

pycontrails.models.humidity_scaling.**histogram_matching**(*era5_rhi*, *product_type*, *level_type*, *member*, *q_method*)

Map ERA5-derived RHi to its corresponding IAGOS quantile via histogram matching.

This matching is performed on a **single** ERA5 ensemble member.

Parameters

- **era5_rhi** (*ArrayLike*) – ERA5-derived RHi values for the given ensemble member.
- **product_type** (`{"reanalysis", "ensemble_members"}`) – The ERA5 product type.
- **level_type** (`{"pressure", "model"}`) – Select whether to perform quantile mapping based on quantiles from pressure- or model-level ERA5 data. Selecting `level_type == "model"` when `product_type == "ensemble_members"` will produce a warning and change `product_type` to `"reanalysis"`.
- **member** (`int | None`) – The ERA5 ensemble member to use. Must be in the range `[0, 10)`. Only used if `product_type == "ensemble_members"`.
- **q_method** (`{None, "cubic-spline", "log-q-log-p"}`) – The interpolation method.

Returns

`npt.NDArray[np.float64]` – The IAGOS quantiles corresponding to the ERA5-derived RHi values. Returned as a numpy array with the same shape and dtype as `era5_rhi`.

pycontrails.models.humidity_scaling.**histogram_matching_all_members**(*era5_rhi_all_members*, *member*, *q_method*)

Recalibrate ERA5-derived RHi values to IAGOS quantiles by histogram matching.

This recalibration requires values for **all** ERA5 ensemble members. Currently, the number of ensemble members is hard-coded to 10.

Parameters

- **era5_rhi_all_members** (`npt.NDArray[np.float64]`) – ERA5-derived RHi values for all ensemble members. This array should have shape `(n, 10)`.
- **member** (`int`) – The ERA5 ensemble member to use. Must be in the range `[0, 10)`.
- **q_method** (`{None, "cubic-spline", "log-q-log-p"}`) – The interpolation method.

Returns

- **ensemble_mean_rhi** (`npt.NDArray[np.float64]`) – The mean RHi values after histogram matching over all ensemble members. This is an array of shape `(n,)`.

- **ensemble_member_rhi** (`npt.NDArray[np.float64]`) – The RHi values after histogram matching for the given ensemble member. This is an array of shape `(n,)`.

11.4 Physics

physics.constants	Meteorological, thermophysical, and aircraft constants.
physics.thermo	Thermodynamic relationships.
physics.jet	Jet aircraft trajectory and performance parameters.
physics.geo	Tools for spherical geometry, solar radiation, and wind advection.
physics.units	Unit conversion support.

11.4.1 pycontrails.physics.constants

Meteorological, thermophysical, and aircraft constants.

Module Attributes

absolute_zero	Absolute zero value $[C]$
g	Gravitational acceleration $[m\ s^{-2}]$
radius_earth	Radius of Earth $[m]$
surface_area_earth	Surface area of Earth $[m^2]$
h_tropopause	ISA height of the tropopause $[m]$ [Wikipedia contributors, 2023]
seconds_per_year	Seconds in a Julian year $[s]$
p_surface	Surface pressure, international standard atmosphere $[Pa]$ [Wikipedia contributors, 2023]
c_pd	Isobaric heat capacity of dry air $[J\ kg^{-1}\ K^{-1}]$
c_pv	Isobaric heat capacity of water vapor $[J\ kg^{-1}\ K^{-1}]$
M_d	Molar mass of dry air $[kg\ mol^{-1}]$
M_v	Molar mass of water $[kg\ mol^{-1}]$
gamma	which heat capacities?
R	molar gas constant $[J\ mol^{-1}\ K^{-1}]$
R_d	Gas constant of dry air $[J\ kg^{-1}\ K^{-1}]$
R_v	Gas constant of water vapour $[J\ kg^{-1}\ K^{-1}]$
epsilon	Ratio of gas constant for dry air / gas constant for water vapor
rho_ice	Density of ice $[kg\ m^{-3}]$
kappa	Adiabatic index air
rho_msl	Standard atmospheric density at mean sea level (MSL) $[kg\ m^{-3}]$
T_msl	Standard atmospheric temperature at mean sea level (MSL) $[K]$
c_msl	Speed of sound at mean sea level (MSL) in standard atmosphere $[m\ s^{-1}]$
T_lapse_rate	The rate at which the ISA ambient temperature falls with altitude $[K\ m^{-1}]$ [Wikipedia contributors, 2023]
solar_constant	Average incident solar radiation, $[W\ m^{-2}]$ This value can range +/- 3% as the earth orbits the sun.
c_p_combustion	Isobaric heat capacity of combustion products $[J\ kg^{-1}\ K^{-1}]$
mu_ice	Real refractive index of ice
lambda_light	Wavelength of visible light (550 nm)
c_r	Ratio between the volume mean radius and the effective radius (Uncertainty +- 0.3)
nominal_rocd	Nominal rate of climb/descent of aircraft $[m\ s^{-1}$‘]

pycontrails.physics.constants.**M_d = 0.0289647**

Molar mass of dry air $[kg\ mol^{-1}]$

pycontrails.physics.constants.**M_v = 0.0180153**

Molar mass of water $[kg\ mol^{-1}]$

pycontrails.physics.constants.**R = 8.314462618**

molar gas constant $[J\ mol^{-1}\ K^{-1}]$

pycontrails.physics.constants.**R_d = 287.05**

Gas constant of dry air $[J\ kg^{-1}\ K^{-1}]$

pycontrails.physics.constants.**R_v = 461.51**
Gas constant of water vapour $[J\ kg^{-1}\ K^{-1}]$

pycontrails.physics.constants.**T_lapse_rate = -0.0065**
The rate at which the ISA ambient temperature falls with altitude $[K\ m^{-1}]$ [Wikipedia contributors, 2023]

pycontrails.physics.constants.**T_msl = 288.15**
Standard atmospheric temperature at mean sea level (MSL) $[K]$

pycontrails.physics.constants.**absolute_zero = -273.15**
Absolute zero value $[C]$

pycontrails.physics.constants.**c_msl = 340.294**
Speed of sound at mean sea level (MSL) in standard atmosphere $[m\ s^{-1}]$

pycontrails.physics.constants.**c_p_combustion = 1250.0**
Isobaric heat capacity of combustion products $[J\ kg^{-1}\ K^{-1}]$

pycontrails.physics.constants.**c_pd = 1004.0**
Isobaric heat capacity of dry air $[J\ kg^{-1}\ K^{-1}]$

pycontrails.physics.constants.**c_pv = 1870.0**
Isobaric heat capacity of water vapor $[J\ kg^{-1}\ K^{-1}]$

pycontrails.physics.constants.**c_r = 0.9**
Ratio between the volume mean radius and the effective radius (Uncertainty +- 0.3)

pycontrails.physics.constants.**epsilon = 0.6219800221013629**
Ratio of gas constant for dry air / gas constant for water vapor

pycontrails.physics.constants.**g = 9.80665**
Gravitational acceleration $[m\ s^{-2}]$

pycontrails.physics.constants.**gamma = 1.4**
which heat capacities?

Type
Ratio of heat capacities, TODO

pycontrails.physics.constants.**h_tropopause = 11000.0**
ISA height of the tropopause $[m]$ [Wikipedia contributors, 2023]

pycontrails.physics.constants.**kappa = 1.4**
Adiabatic index air

pycontrails.physics.constants.**lambda_light = 5.5e-07**
Wavelength of visible light (550 nm)

pycontrails.physics.constants.**mu_ice = 1.31**
Real refractive index of ice

pycontrails.physics.constants.**nominal_rocd = 12.7**
Nominal rate of climb/descent of aircraft [$m\ s^{-1}$‘]

pycontrails.physics.constants.**p_surface = 101325.0**
Surface pressure, international standard atmosphere $[Pa]$ [Wikipedia contributors, 2023]

pycontrails.physics.constants.**radius_earth = 6371229.0**
Radius of Earth $[m]$

pycontrails.physics.constants.**rho_ice = 917.0**

Density of ice $[kg\ m^{-3}]$

pycontrails.physics.constants.**rho_msl = 1.225**

Standard atmospheric density at mean sea level (MSL) $[kg\ m^{-3}]$

pycontrails.physics.constants.**seconds_per_year = 31557600.0**

Seconds in a Julian year $[s]$

pycontrails.physics.constants.**solar_constant = 1361.0**

Average incident solar radiation, $[W\ m^{-2}]$ This value can range +/- 3% as the earth orbits the sun. From [Laboratory, 2022] citing [Paltridge *et al.*, 1976]

pycontrails.physics.constants.**surface_area_earth = 510072000000000.0**

Surface area of Earth $[m^2]$

11.4.2 pycontrails.physics.thermo

Thermodynamic relationships.

Functions

T_potential(T, p)	Calculate potential temperature.
T_potential_gradient(T_top, p_top, T_btm, ...)	Calculate the potential temperature gradient between two altitudes.
brunt_vaisala_frequency(p, T, T_grad)	Calculate the Brunt-Vaisaila frequency.
c_pm(q)	Calculate isobaric heat capacity of moist air.
e_sat_ice(T)	Calculate saturation pressure of water vapor over ice.
e_sat_liquid(T)	Calculate saturation pressure of water vapor over liquid water.
p_vapor(q, p)	Calculate the vapor pressure.
pressure_dz(T, p, dz)	Calculate the pressure altitude `dz` meters below input pressure.
q_sat(T, p)	Calculate saturation specific humidity over liquid or ice.
q_sat_ice(T, p)	Calculate saturation specific humidity over ice.
q_sat_liquid(T, p)	Calculate saturation specific humidity over liquid.
rh(q, T, p)	Calculate the relative humidity with respect to to liquid water.
rhi(q, T, p)	Calculate the relative humidity with respect to ice (RHi).
rho_d(T, p)	Calculate air density for (T, p) assuming dry air.
rho_v(T, p)	Calculate the air density for (T, p) assuming all water vapor.

pycontrails.physics.thermo.**T_potential**(*T*, *p*)

Calculate potential temperature.

The potential temperature is the temperature that an air parcel would attain if adiabatically brought to a standard reference pressure, `constants.p_surface`.

Parameters

- **T** (*ArrayScalarLike*) – Temperature , $[K]$
- **p** (*ArrayScalarLike*) – Pressure, $[Pa]$

Returns
ArrayScalarLike – Potential Temperature, [K]

References

- https://en.wikipedia.org/wiki/Potential_temperature

pycontrails.physics.thermo.**T_potential_gradient**(*T_top*, *p_top*, *T_btm*, *p_btm*, *dz*)

Calculate the potential temperature gradient between two altitudes.

Parameters

- **T_top** (*ArrayScalarLike*) – Temperature at original altitude, [K]
- **p_top** (*ArrayScalarLike*) – Pressure at original altitude, [Pa]
- **T_btm** (*ArrayScalarLike*) – Temperature at lower altitude, [K]
- **p_btm** (*ArrayScalarLike*) – Pressure at lower altitude, [Pa]
- **dz** (float) – Difference in altitude between measurements, [m]

Returns
ArrayScalarLike – Potential Temperature gradient, [$K\ m^{-1}$]

pycontrails.physics.thermo.**brunt_vaisala_frequency**(*p*, *T*, *T_grad*)

Calculate the Brunt-Vaisaila frequency.

The Brunt-Vaisaila frequency is the frequency at which a vertically displaced parcel will oscillate within a statically stable environment.

Parameters

- **p** (numpy.ndarray) – Pressure, [Pa]
- **T** (numpy.ndarray) – Temperature , [K]
- **T_grad** (numpy.ndarray) – Potential Temperature gradient (see *T_potential_gradient()*), [$K\ m^{-1}$]

Returns
numpy.ndarray – Brunt-Vaisaila frequency, [s^{-1}]

References

- https://en.wikipedia.org/wiki/Brunt%E2%80%93V%C3%A4is%C3%A4l%C3%A4_frequency

pycontrails.physics.thermo.**c_pm**(*q*)

Calculate isobaric heat capacity of moist air.

Parameters
q (*ArrayScalarLike*) – Specific humidity, [$kg\ kg^{-1}$]

Returns
ArrayScalarLike – Isobaric heat capacity of moist air, [$J\ kg^{-1}\ K^{-1}$]

Notes

Some models (including CoCiP) use a constant value here (1004 $J\ kg^{-1}\ K^{-1}$)

pycontrails.physics.thermo.**e_sat_ice**(T)

Calculate saturation pressure of water vapor over ice.

Parameters
T (*ArrayScalarLike*) – Temperature, [K]

Returns
ArrayScalarLike – Saturation pressure of water vapor over ice, [Pa]

References

- [Sonntag, 1994]

pycontrails.physics.thermo.**e_sat_liquid**(T)

Calculate saturation pressure of water vapor over liquid water.

Parameters
T (*ArrayScalarLike*) – Temperature, [K]

Returns
ArrayScalarLike – Saturation pressure of water vapor over liquid water, [Pa]

References

- [Sonntag, 1994]

pycontrails.physics.thermo.**p_vapor**(q, p)

Calculate the vapor pressure.

Parameters

- **q** (ArrayScalarTypeVar) – Specific humidity, [$kg\ kg^{-1}$]
- **p** (ArrayScalarTypeVar) – Pressure, [Pa]

Returns
ArrayScalarTypeVar – Vapor pressure, [Pa]

pycontrails.physics.thermo.**pressure_dz**(T, p, dz)

Calculate the pressure altitude dz meters below input pressure.

Returns surface pressure if the calculated pressure altitude is greater than constants.p_surface.

Parameters

- **T** (*ArrayScalarLike*) – Temperature, [K]
- **p** (*ArrayScalarLike*) – Pressure, [Pa]
- **dz** (float) – Difference in altitude between measurements, [m]

Returns
ArrayScalarLike – Pressure at altitude, [Pa]

Notes

This is used to calculate the temperature gradient and wind shear.

pycontrails.physics.thermo.**q_sat**(T, p)

Calculate saturation specific humidity over liquid or ice.

When T is above 0 C, liquid saturation is computed. Otherwise, ice saturation is computed.

Parameters

- **T** (*ArrayScalarLike*) – Temperature, [K]
- **p** (*ArrayScalarLike*) – Pressure, [Pa]

Returns

ArrayScalarLike – Saturation specific humidity, [$kg\ kg^{-1}$]

Notes

Smith et al. (1999)

pycontrails.physics.thermo.**q_sat_ice**(T, p)

Calculate saturation specific humidity over ice.

Parameters

- **T** (*ArrayScalarLike*) – Temperature, [K]
- **p** (*ArrayScalarLike*) – Pressure, [Pa]

Returns

ArrayScalarLike – Saturation specific humidity, [$kg\ kg^{-1}$]

Notes

Smith et al. (1999)

pycontrails.physics.thermo.**q_sat_liquid**(T, p)

Calculate saturation specific humidity over liquid.

Parameters

- **T** (*ArrayScalarLike*) – Temperature, [K]
- **p** (*ArrayScalarLike*) – Pressure, [Pa]

Returns

ArrayScalarLike – Saturation specific humidity, [$kg\ kg^{-1}$]

Notes

Smith et al. (1999)

pycontrails.physics.thermo.**rh**(q, T, p)

Calculate the relative humidity with respect to to liquid water.

Parameters

- **q** (*ArrayScalarLike*) – Specific humidity, $[kg\ kg^{-1}]$
- **T** (*ArrayScalarLike*) – Temperature, $[K]$
- **p** (*ArrayScalarLike*) – Pressure, $[Pa]$

Returns

ArrayScalarLike – Relative Humidity, $[0 - 1]$

pycontrails.physics.thermo.**rhi**(q, T, p)

Calculate the relative humidity with respect to ice (RHi).

Parameters

- **q** (*ArrayScalarLike*) – Specific humidity, $[kg\ kg^{-1}]$
- **T** (*ArrayScalarLike*) – Temperature, $[K]$
- **p** (*ArrayScalarLike*) – Pressure, $[Pa]$

Returns

ArrayScalarLike – Relative Humidity over ice, $[0 - 1]$

pycontrails.physics.thermo.**rho_d**(T, p)

Calculate air density for (T, p) assuming dry air.

Parameters

- **T** (*ArrayScalarLike*) – Temperature, $[K]$
- **p** (*ArrayScalarLike*) – Pressure, $[Pa]$

Returns

ArrayScalarLike – Air density of dry air, $[kg\ m^{-3}]$

pycontrails.physics.thermo.**rho_v**(T, p)

Calculate the air density for (T, p) assuming all water vapor.

Parameters

- **T** (*ArrayScalarLike*) – Temperature, $[K]$
- **p** (*ArrayScalarLike*) – Pressure, $[Pa]$

Returns

ArrayScalarLike – Air density of water vapor, $[kg\ m^{-3}]$

11.4.3 pycontrails.physics.jet

Jet aircraft trajectory and performance parameters.

This module includes common functions to calculate jet aircraft trajectory and performance parameters, including fuel quantities, mass, thrust setting and propulsion efficiency.

Functions

acceleration(true_airspeed, segment_duration)	Calculate the acceleration/deceleration at each waypoint.
air_to_fuel_ratio(thrust_setting, *[, ...])	Calculate air-to-fuel ratio from thrust setting.
aircraft_weight(aircraft_mass)	Calculate the aircraft weight at each waypoint.
climb_descent_angle(true_airspeed, rocd)	Calculate angle between the horizontal plane and the actual flight path.
clip_mach_number(true_airspeed, ...)	Compute the Mach number from the true airspeed and ambient temperature.
combustor_inlet_pressure(pressure_ratio, ...)	Calculate combustor inlet pressure, P_3.
combustor_inlet_temperature(comp_efficiency, ...)	Calculate combustor inlet temperature, T_3.
compressor_inlet_pressure(p, mach_num)	Calculate compressor inlet pressure for Jet engine, P_2.
compressor_inlet_temperature(T, mach_num)	Calculate compressor inlet temperature for Jet engine, T_2.
density_ratio(rho)	Calculate the ratio of air density relative to the air density at mean-sea-level.
equivalent_fuel_flow_rate_at_cruise(...)	Convert fuel mass flow rate at sea level to equivalent fuel flow rate at cruise conditions.
equivalent_fuel_flow_rate_at_sea_level(...)	Convert fuel mass flow rate at cruise conditions to equivalent flow rate at sea level.
fuel_burn(fuel_flow, segment_duration)	Calculate the fuel consumption at each waypoint.
initial_aircraft_mass(*, ...)	Estimate initial aircraft mass as a function of load factor and fuel requirements.
overall_propulsion_efficiency(true_airspeed, ...)	Calculate the overall propulsion efficiency (OPE).
pressure_ratio(p)	Calculate the ratio of ambient pressure relative to the surface pressure.
reserve_fuel_requirements(rocd, altitude_ft, ...)	Estimate reserve fuel requirements.
temperature_ratio(T)	Calculate the ratio of ambient temperature relative to the temperature at mean sea level.
thrust_force(altitude, true_airspeed, ...)	Calculate the thrust force at each waypoint.
thrust_setting_nd(true_airspeed, ...[, ...])	Calculate the non-dimensionalized thrust setting of a Jet engine.
turbine_inlet_temperature(afr, T_comb_inlet, ...)	Calculate turbine inlet temperature, T_4.
update_aircraft_mass(*, ...)	Update aircraft mass based on the simulated total fuel consumption.

pycontrails.physics.jet.**acceleration**(*true_airspeed*, *segment_duration*)

Calculate the acceleration/deceleration at each waypoint.

Parameters

- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed, $[m\ s^{-1}]$
- **segment_duration** (`npt.NDArray[np.float64]`) – Time difference between waypoints, $[s]$

Returns
`npt.NDArray[np.float64]` – Acceleration/deceleration, [$m\ s^{-2}$]

See also:

`flight.segment_duration()`

pycontrails.physics.jet.**air_to_fuel_ratio**(*thrust_setting*, ***, *cruise=False*, *T_compressor_inlet=None*)

Calculate air-to-fuel ratio from thrust setting.

Parameters

- **thrust_setting** (*ArrayScalarLike*) – Engine thrust setting, unitless
- **cruise** (`bool`) – Estimate thrust setting for cruise conditions. Defaults to False.
- **T_compressor_inlet** (`None | ArrayScalarLike`) – Compressor inlet temperature, [K] Required if `cruise` is True. Defaults to None

Returns
ArrayScalarLike – Air-to-fuel ratio, unitless

References

- [Cumpsty and Heyes, 2015]
- AFR equation from [Stettler *et al.*, 2013]
- Scaling factor to cruise from Eq. (30) of [DuBois and Paynter, 2006]

pycontrails.physics.jet.**aircraft_weight**(*aircraft_mass*)

Calculate the aircraft weight at each waypoint.

Parameters
aircraft_mass (`ArrayOrFloat`) – Aircraft mass, [kg]

Returns
`ArrayOrFloat` – Aircraft weight, [N]

pycontrails.physics.jet.**climb_descent_angle**(*true_airspeed*, *rocd*)

Calculate angle between the horizontal plane and the actual flight path.

Parameters

- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed, [$m\ s^{-1}$]
- **rocd** (`npt.NDArray[np.float64]`) – Rate of climb/descent, [$ftmin^{-1}$]

Returns
`npt.NDArray[np.float64]` – Climb (positive value) or descent (negative value) angle, [deg]

See also:

`flight.segment_rocd()`, `flight.segment_true_airspeed()`

pycontrails.physics.jet.**clip_mach_number**(*true_airspeed*, *air_temperature*, *max_mach_number*)

Compute the Mach number from the true airspeed and ambient temperature.

This method clips the computed Mach number to the value of `max_mach_number`.

If no mach number exceeds `max_mach_number`, the original array `true_airspeed` and the computed Mach number are returned.

Parameters

- **true_airspeed** (`npt.NDArray[np.float64]`) – Array of true airspeed, $[m\ s^{-1}]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Array of ambient temperature, [:math: K]
- **max_mach_number** (`ArrayOrFloat`) – Maximum mach number associated to aircraft. If no clipping is desired, this can be set tp *np.inf*.

Returns

- **true_airspeed** (`npt.NDArray[np.float64]`) – Array of true airspeed, $[m\ s^{-1}]$. All values are clipped at `max_mach_number`.
- **mach_num** (`npt.NDArray[np.float64]`) – Array of Mach numbers, $[Ma]$. All values are clipped at `max_mach_number`.

pycontrails.physics.jet.**combustor_inlet_pressure**(*pressure_ratio*, *p_comp_inlet*, *thrust_setting*)

Calculate combustor inlet pressure, P_3.

Parameters

- **pressure_ratio** (`float`) – Engine pressure ratio, unitless
- **p_comp_inlet** (*ArrayScalarLike*) – Compressor inlet pressure, $[Pa]$
- **thrust_setting** (*ArrayScalarLike*) – Engine thrust setting, unitless

Returns

ArrayScalarLike – Combustor inlet pressure, $[Pa]$

References

- [Stettler *et al.*, 2013]
- [Cumpsty and Heyes, 2015]

pycontrails.physics.jet.**combustor_inlet_temperature**(*comp_efficiency*, *T_comp_inlet*, *p_comp_inlet*, *p_comb_inlet*)

Calculate combustor inlet temperature, T_3.

Parameters

- **comp_efficiency** (`float`) – Engine compressor efficiency, $[0 - 1]$
- **T_comp_inlet** (*ArrayScalarLike*) – Compressor inlet temperature, $[K]$
- **p_comp_inlet** (*ArrayScalarLike*) – Compressor inlet pressure, $[Pa]$
- **p_comb_inlet** (*ArrayScalarLike*) – Compressor inlet pressure, $[Pa]$

Returns

ArrayScalarLike – Combustor inlet temperature, $[K]$

References

- [Stettler *et al.*, 2013]
- [Cumpsty and Heyes, 2015]

pycontrails.physics.jet.**compressor_inlet_pressure**(*p*, *mach_num*)

Calculate compressor inlet pressure for Jet engine, P_2.

Parameters

- **p** (*ArrayScalarLike*) – Ambient pressure, [Pa]
- **mach_num** (*ArrayScalarLike*) – Mach number

Returns

ArrayScalarLike – Compressor inlet pressure, [Pa]

References

- [Stettler *et al.*, 2013]
- [Cumpsty and Heyes, 2015]

pycontrails.physics.jet.**compressor_inlet_temperature**(*T*, *mach_num*)

Calculate compressor inlet temperature for Jet engine, T_2.

Parameters

- **T** (*ArrayScalarLike*) – Ambient temperature, [K]
- **mach_num** (*ArrayScalarLike*) – Mach number

Returns

ArrayScalarLike – Compressor inlet temperature, [K]

References

- [Stettler *et al.*, 2013]
- [Cumpsty and Heyes, 2015]

pycontrails.physics.jet.**density_ratio**(*rho*)

Calculate the ratio of air density relative to the air density at mean-sea-level.

Parameters

rho (`npt.NDArray[np.float64]`) – Air density, [$kg\ m^3$]

Returns

`npt.NDArray[np.float64]` – Ratio of the density to the air density at mean sea-level (MSL).

pycontrails.physics.jet.**equivalent_fuel_flow_rate_at_cruise**(*fuel_flow_sls*, *theta_amb*, *delta_amb*, *mach_num*)

Convert fuel mass flow rate at sea level to equivalent fuel flow rate at cruise conditions.

Refer to Eq. (40) in [DuBois and Paynter, 2006].

Parameters

- **fuel_flow_sls** (`npt.NDArray[np.float64] | float`) – Fuel mass flow rate, [kgs^{-1}]

- **theta_amb** (ArrayOrFloat) – Ratio of the ambient temperature to the temperature at mean sea-level.
- **delta_amb** (ArrayOrFloat) – Ratio of the pressure altitude to the surface pressure.
- **mach_num** (ArrayOrFloat) – Mach number

Returns
npt.NDArray[np.float64] – Estimate of fuel mass flow rate at sea level, [$kg\ s^{-1}$]

References

- [DuBois and Paynter, 2006]

pycontrails.physics.jet.**equivalent_fuel_flow_rate_at_sea_level**(*fuel_flow_cruise*, *theta_amb*, *delta_amb*, *mach_num*)

Convert fuel mass flow rate at cruise conditions to equivalent flow rate at sea level.

Refer to Eq. (40) in [DuBois and Paynter, 2006].

Parameters
- **fuel_flow_cruise** (npt.NDArray[np.float64]) – Fuel mass flow rate per engine, [kgs^{-1}]
- **theta_amb** (npt.NDArray[np.float64]) – Ratio of the ambient temperature to the temperature at mean sea-level.
- **delta_amb** (npt.NDArray[np.float64]) – Ratio of the pressure altitude to the surface pressure.
- **mach_num** (npt.NDArray[np.float64]) – Mach number, [:math: *Ma*]

Returns
npt.NDArray[np.float64] – Estimate of fuel flow per engine at sea level, [$kg\ s^{-1}$].

References

- [DuBois and Paynter, 2006]

pycontrails.physics.jet.**fuel_burn**(*fuel_flow*, *segment_duration*)

Calculate the fuel consumption at each waypoint.

Parameters
- **fuel_flow** (npt.NDArray[np.float64]) – Fuel mass flow rate, [kgs^{-1}]
- **segment_duration** (npt.NDArray[np.float64]) – Time difference between waypoints, [s]

Returns
npt.NDArray[np.float64] – Fuel consumption at each waypoint, [kg]

pycontrails.physics.jet.**initial_aircraft_mass**(*, *operating_empty_weight*, *max_takeoff_weight*, *max_payload*, *total_fuel_burn*, *total_reserve_fuel*, *load_factor*)

Estimate initial aircraft mass as a function of load factor and fuel requirements.

This function uses the following equation:

```
TOM = OEM + PM + FM_nc + TFM
    = OEM + LF * MPM + FM_nc + TFM
```

where: - TOM is the aircraft take-off mass - OEM is the aircraft operating empty weight - PM is the payload mass - FM_nc is the mass of the fuel not consumed - TFM is the trip fuel mass - LF is the load factor - MPM is the maximum payload mass

Parameters

- **operating_empty_weight** (`float`) – Aircraft operating empty weight, i.e. the basic weight of an aircraft including the crew and necessary equipment, but excluding usable fuel and payload, [kg]
- **max_takeoff_weight** (`float`) – Aircraft maximum take-off weight, [kg]
- **max_payload** (`float`) – Aircraft maximum payload, [kg]
- **total_fuel_burn** (`float`) – Total fuel consumption for the flight, obtained from prior iterations, [kg]
- **total_reserve_fuel** (`float`) – Total reserve fuel requirements, [kg]
- **load_factor** (`float`) – Aircraft load factor assumption (between 0 and 1)

Returns

`float` – Aircraft mass at the initial waypoint, [kg]

References

- [Wasiuk *et al.*, 2015]

See also:

reserve_fuel_requirements()

pycontrails.physics.jet.**overall_propulsion_efficiency**(*true_airspeed*, *F_thrust*, *fuel_flow*, *q_fuel*, *is_descent*, *threshold=0.5*)

Calculate the overall propulsion efficiency (OPE).

Negative OPE values can occur during the descent phase and is clipped to a lower bound of 0, while an upper bound of `threshold` is also applied. The most efficient engines today do not exceed this value.

Parameters

- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed for each waypoint, [ms^{-1}].
- **F_thrust** (`npt.NDArray[np.float64]`) – Thrust force provided by the engine, [N].
- **fuel_flow** (`npt.NDArray[np.float64]`) – Fuel mass flow rate, [kgs^{-1}].
- **q_fuel** (`float`) – Lower calorific value (LCV) of fuel, [$J\ kg_{fuel}^{-1}$].
- **is_descent** (`npt.NDArray[np.float64] | None`) – Boolean array that indicates if a waypoint is in a descent phase.
- **threshold** (`float`) – Upper bound for realistic engine efficiency.

Returns

`npt.NDArray[np.float64]` – Overall propulsion efficiency (OPE)

References

- [Schumann, 1996]
- [Cumpsty and Heyes, 2015]

pycontrails.physics.jet.**pressure_ratio**(*p*)

Calculate the ratio of ambient pressure relative to the surface pressure.

Parameters
p (npt.NDArray[np.float64]) – Air pressure, [Pa]

Returns
npt.NDArray[np.float64] – Ratio of the pressure altitude to the surface pressure.

pycontrails.physics.jet.**reserve_fuel_requirements**(*rocd*, *altitude_ft*, *fuel_flow*, *fuel_burn*)

Estimate reserve fuel requirements.

Parameters

- **rocd** (npt.NDArray[np.float64]) – Rate of climb and descent, [$ft\ min^{-1}$]
- **altitude_ft** (npt.NDArray[np.float64]) – Altitude, [ft]
- **fuel_flow** (npt.NDArray[np.float64]) – Fuel mass flow rate, [$kg\ s^{-1}$].
- **fuel_burn** (npt.NDArray[np.float64]) – Fuel consumption for each waypoint, [kg]

Returns
npt.NDArray[np.float64] – Reserve fuel requirements, [kg]

References

- [Wasiuk *et al.*, 2015]

Notes

The real-world calculation of the reserve fuel requirements is highly complex (refer to Section 2.3.3 of [Wasiuk *et al.*, 2015]). This implementation is simplified by taking the maximum between the following two conditions:

1. Fuel required to fly +90 minutes at the main cruise altitude at the end of the cruise aircraft weight.
2. Uplift the total fuel consumption for the flight by +15%

See also:

flight.segment_phase(), *fuel_burn()*

pycontrails.physics.jet.**temperature_ratio**(*T*)

Calculate the ratio of ambient temperature relative to the temperature at mean sea level.

Parameters
T (npt.NDArray[np.float64]) – Air temperature, [K]

Returns
npt.NDArray[np.float64] – Ratio of the temperature to the temperature at mean sea-level (MSL).

pycontrails.physics.jet.**thrust_force**(*altitude*, *true_airspeed*, *segment_duration*, *aircraft_mass*, *F_drag*)

Calculate the thrust force at each waypoint.

Parameters

- **altitude** (`npt.NDArray[np.float64]`) – Waypoint altitude, $[m]$
- **true_airspeed** (`npt.NDArray[np.float64]`) – True airspeed, $[m\ s^{-1}]$
- **segment_duration** (`npt.NDArray[np.float64]`) – Time difference between waypoints, $[s]$
- **aircraft_mass** (`npt.NDArray[np.float64]`) – Aircraft mass, $[kg]$
- **F_drag** (`npt.NDArray[np.float64]`) – Draft force, $[N]$

Returns

`npt.NDArray[np.float64]` – Thrust force, $[N]$

References

- [Eurocontrol, 2010]

Notes

The model balances the rate of forces acting on the aircraft with the rate in increase in energy.

This estimate of thrust force is used in the BADA Total-Energy Model (Eq. 3.2-1).

Negative thrust must be corrected.

pycontrails.physics.jet.**thrust_setting_nd**(*true_airspeed*, *thrust_setting*, *T*, *p*, *pressure_ratio*, *q_fuel*, ***, *comp_efficiency=0.9*, *cruise=False*)

Calculate the non-dimensionalized thrust setting of a Jet engine.

Result is in terms of the ratio of turbine inlet to the compressor inlet temperature (t4_t2)

Parameters

- **true_airspeed** (*ArrayScalarLike*) – True airspeed, $[m\ s^{-1}]$
- **thrust_setting** (*ArrayScalarLike*) – Engine thrust setting, unitless
- **T** (*ArrayScalarLike*) – Ambient temperature, $[K]$
- **p** (*ArrayScalarLike*) – Ambient pressure, $[Pa]$
- **pressure_ratio** (`float`) – Engine pressure ratio, unitless
- **q_fuel** (`float`) – Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$
- **comp_efficiency** (`float`, *optional*) – Engine compressor efficiency, $[0 - 1]$. Defaults to 0.9
- **cruise** (`bool`, *optional*) – Defaults to False

Returns

ArrayScalarLike – Ratio of turbine inlet to the compressor inlet temperature, unitless

References

- [Cumpsty and Heyes, 2015]
- [Teoh *et al.*, 2022]

pycontrails.physics.jet.**turbine_inlet_temperature**(*afr*, *T_comb_inlet*, *q_fuel*)

Calculate turbine inlet temperature, T_4.

Parameters

- **afr** (*ArrayScalarLike*) – Air-to-fuel ratio, unitless
- **T_comb_inlet** (*ArrayScalarLike*) – Combustor inlet temperature, $[K]$
- **q_fuel** (float) – Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

Returns

ArrayScalarLike – Tubrine inlet temperature, $[K]$

References

- [Cumpsty and Heyes, 2015]

pycontrails.physics.jet.**update_aircraft_mass**(*, *operating_empty_weight*, *max_takeoff_weight*, *max_payload*, *fuel_burn*, *total_reserve_fuel*, *load_factor*, *takeoff_mass*)

Update aircraft mass based on the simulated total fuel consumption.

Used internally for finding aircraft mass iteratively.

Parameters

- **operating_empty_weight** (float) – Aircraft operating empty weight, i.e. the basic weight of an aircraft including the crew and necessary equipment, but excluding usable fuel and payload, $[kg]$.
- **ref_mass** (float) – Aircraft reference mass, $[kg]$.
- **max_takeoff_weight** (float) – Aircraft maximum take-off weight, $[kg]$.
- **max_payload** (float) – Aircraft maximum payload, $[kg]$
- **fuel_burn** (npt.NDArray[np.float64]) – Fuel consumption for each waypoint, $[kg]$
- **total_reserve_fuel** (float) – Total reserve fuel requirements, $[kg]$
- **load_factor** (float) – Aircraft load factor assumption (between 0 and 1). This is the ratio of the actual payload weight to the maximum payload weight.
- **takeoff_mass** (float | None) – Initial aircraft mass, $[kg]$. If None, the initial mass is calculated using *initial_aircraft_mass()*. If supplied, all other parameters except fuel_burn are ignored.

Returns

npt.NDArray[np.float64] – Updated aircraft mass, $[kg]$

See also:

fuel_burn(), *reserve_fuel_requirements()*, *initial_aircraft_mass()*

11.4.4 pycontrails.physics.geo

Tools for spherical geometry, solar radiation, and wind advection.

Functions

advect_latitude(latitude, v_wind, dt)	Calculate the latitude of a particle after time `dt` caused by advection due to wind.
advect_level(level, vertical_velocity, ...)	Calculate the pressure level of a particle after time `dt`.
advect_longitude(longitude, latitude, u_wind, dt)	Calculate the longitude of a particle after time *dt* caused by advection due to wind.
azimuth(lons0, lats0, lons1, lats1)	Calculate angle relative to true north for set of coordinates.
azimuth_to_direction(azimuth_, latitude)	Calculate rectangular direction from spherical azimuth.
cosine_solar_zenith_angle(longitude, ...)	Calculate the cosine of the solar zenith angle.
days_since_reference_year(time[, ref_year])	Calculate the days elapsed since the start of the reference year.
domain_surface_area([spatial_bbox, ...])	Calculate surface area in the provided spatial bounding box.
forward_azimuth(lons, lats, az, dist)	Calculate coordinates along forward azimuth.
grid_surface_area(longitude, latitude)	Calculate surface area that is covered by each pixel in a longitude-latitude grid.
haversine(lons0, lats0, lons1, lats1)	Calculate haversine distance between points in (lons0, lats0) and (lons1, lats1).
hours_since_start_of_day(time)	Calculate the hours elapsed since the start of day (00:00:00 UTC).
longitudinal_angle(lons0, lats0, lons1, lats1)	Calculate angle with longitudinal axis for sequence of segments.
orbital_correction_for_solar_hour_angle(...)	Calculate correction to the solar hour angle due to Earth's orbital location.
orbital_position(time)	Calculate the orbital position of Earth to a reference point set at the start of year.
segment_angle(longitude, latitude)	Calculate the angle between coordinate segments and the longitudinal axis.
segment_azimuth(longitude, latitude)	Calculate the angle between coordinate segments and true north.
segment_haversine(longitude, latitude)	Calculate haversine distance between consecutive points along path.
segment_length(longitude, latitude, altitude)	Calculate the segment length between coordinates by assuming a great circle distance.
solar_constant(theta_rad)	Calculate the solar electromagnetic radiation per unit area from orbital position.
solar_declination_angle(theta_rad)	Calculate the solar declination angle from the orbital position in radians (theta_rad).
solar_direct_radiation(longitude, latitude, time)	Calculate the instantaneous theoretical solar direct radiation (SDR).
solar_hour_angle(longitude, time, theta_rad)	Calculate the sun's East to West angular displacement around the polar axis.
spatial_bounding_box(longitude, latitude[, ...])	Construct rectangular spatial bounding box from a set of waypoints.

pycontrails.physics.geo.**advect_latitude**(*latitude*, *v_wind*, *dt*)

Calculate the latitude of a particle after time `dt` caused by advection due to wind.

Note: It is possible for advected latitude values to lie outside of the WGS84 domain `[-90, 90]`. In `Cocip` models, latitude values close to the poles create an end of life condition, thereby avoiding this issue. In practice, such situations are very rare.

These polar divergence issues could also be addressed by reflecting the longitude values 180 degrees via a spherical equivalence such as `(lon, lat) ~ (lon + 180, 180 - lat)`. This approach is not currently taken.

Parameters

- **latitude** (*ArrayLike*) – Original latitude, [deg]
- **v_wind** (*ArrayLike*) – Wind speed in the latitudinal direction, $[ms^{-1}]$
- **dt** (numpy.ndarray) – Advection time delta

Returns

ArrayLike – New latitude value, [deg]

pycontrails.physics.geo.**advect_level**(*level*, *vertical_velocity*, *rho_air*, *terminal_fall_speed*, *dt*)

Calculate the pressure level of a particle after time `dt`.

This function calculates the new pressure level of a particle as a result of vertical advection caused by the vertical velocity and terminal fall speed.

Parameters

- **level** (*ArrayLike*) – Pressure level, $[hPa]$
- **vertical_velocity** (*ArrayLike*) – Vertical velocity, $[Pas^{-1}]$
- **rho_air** (`ArrayLike | float`) – Air density, $[kgm^{-3}]$
- **terminal_fall_speed** (`ArrayLike | float`) – Terminal fall speed of the particle, $[ms^{-1}]$
- **dt** (`npt.NDArray[np.timedelta64] | np.timedelta64`) – Time delta for each waypoint

Returns

ArrayLike – New pressure level, $[hPa]$

pycontrails.physics.geo.**advect_longitude**(*longitude*, *latitude*, *u_wind*, *dt*)

Calculate the longitude of a particle after time *dt* caused by advection due to wind.

Automatically wrap over the antimeridian if necessary.

Parameters

- **longitude** (*ArrayLike*) – Original longitude, [deg]
- **latitude** (*ArrayLike*) – Original latitude, [deg]
- **u_wind** (*ArrayLike*) – Wind speed in the longitudinal direction, $[ms^{-1}]$
- **dt** (numpy.ndarray) – Advection timestep

Returns

ArrayLike – New longitude value, [deg]

pycontrails.physics.geo.**azimuth**(*lons0*, *lats0*, *lons1*, *lats1*)

Calculate angle relative to true north for set of coordinates.

Parameters

- **lons0** (npt.NDArray[np.float64]) – Longitude values of initial endpoints, [deg].
- **lats0** (npt.NDArray[np.float64]) – Latitude values of initial endpoints, [deg].
- **lons1** (npt.NDArray[np.float64]) – Longitude values of terminal endpoints, [deg].
- **lats1** (npt.NDArray[np.float64]) – Latitude values of terminal endpoints, [deg].

References

- [Wikipedia contributors, 2023]

Returns

npt.NDArray[np.float64] – Azimuth relative to true north (0 deg), [deg]

See also:

longitudinal_angle()

pycontrails.physics.geo.**azimuth_to_direction**(*azimuth_*, *latitude*)

Calculate rectangular direction from spherical azimuth.

This implementation uses the equation

```
cos(latitude) / tan(azimuth) = sin_a / cos_a
```

to solve for *sin_a* and *cos_a*.

Parameters

- **azimuth_** (npt.NDArray[np.float64]) – Angle measured clockwise from true north, [deg]
- **latitude** (npt.NDArray[np.float64]) – Latitude value of the point, [deg]

Returns

tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]] – A tuple of sine and cosine values.

pycontrails.physics.geo.**cosine_solar_zenith_angle**(*longitude*, *latitude*, *time*, *theta_rad*)

Calculate the cosine of the solar zenith angle.

Return $(\cos(\theta))$, where θ is the angle between the sun and the vertical direction.

Parameters

- **longitude** (*ArrayLike*) – Longitude, [deg]
- **latitude** (*ArrayLike*) – Latitude, [deg]
- **time** (*ArrayLike*) – Time, formatted as np.datetime64
- **theta_rad** (*ArrayLike*) – Orbital position, [rad]. Output of *orbital_position()*.

Returns

ArrayLike – Cosine of the solar zenith angle

References

- [Wikipedia contributors, 2023]

See also:

orbital_position(), *solar_declination_angle()*, *solar_hour_angle()*

pycontrails.physics.geo.**days_since_reference_year**(*time*, *ref_year=2000*)

Calculate the days elapsed since the start of the reference year.

Parameters

- **time** (*ArrayLike*) – ArrayLike of `np.datetime64` times
- **ref_year** (`int`, *optional*) – Year of reference

Returns

ArrayLike – Days elapsed since the reference year. Output `dtype` is `np.float64`.

Raises

`RuntimeError` – Raises when reference year is greater than the time of *time* element

pycontrails.physics.geo.**domain_surface_area**(*spatial_bbox=(-180.0, -90.0, 180.0, 90.0)*, *spatial_grid_res=0.5*)

Calculate surface area in the provided spatial bounding box.

Parameters

- **spatial_bbox** (`tuple[float, float, float, float]`) – Spatial bounding box, `(lon_min, lat_min, lon_max, lat_max)`, [deg]
- **spatial_grid_res** (`float`) – Spatial grid resolution, [deg]

Returns

`float` – Domain surface area, [m^2]

pycontrails.physics.geo.**forward_azimuth**(*lons*, *lats*, *az*, *dist*)

Calculate coordinates along forward azimuth.

This function is identical to the *pyproj.Geod.fwd* method when working on a spherical earth. Both signatures are also identical. This implementation is generally more performant.

Parameters

- **lons** (`npt.NDArray[np.float64]`) – Array of longitude values.
- **lats** (`npt.NDArray[np.float64]`) – Array of latitude values.
- **az** (`npt.NDArray[np.float64] | float`) – Azimuth, measured in [deg].
- **dist** (`npt.NDArray[np.float64] | float`) – Distance [m] between initial longitude latitude values and point to be computed.

Returns

`tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]]` – Tuple of longitude latitude arrays.

See also:

:meth:pyproj.Geod.fwd

pycontrails.physics.geo.**grid_surface_area**(*longitude*, *latitude*)

Calculate surface area that is covered by each pixel in a longitude-latitude grid.

Parameters

- **longitude** (`npt.NDArray[np.float64]`) – Longitude coordinates in a longitude-latitude grid, [deg]. Must be in ascending order.
- **latitude** (`npt.NDArray[np.float64]`) – Latitude coordinates in a longitude-latitude grid, [deg]. Must be in ascending order.

Returns

`xarray.DataArray` – Surface area of each pixel in a longitude-latitude grid, $[m^2]$

References

- https://www.pmel.noaa.gov/maillists/tmap/ferret_users/fu_2004/msg00023.html

pycontrails.physics.geo.**haversine**(*lons0*, *lats0*, *lons1*, *lats1*)

Calculate haversine distance between points in (lons0, lats0) and (lons1, lats1).

Handles coordinates crossing the antimeridian line (-180, 180).

Parameters

- **lons0, lats0** (*ArrayLike*) – Coordinates of initial points, [deg]
- **lons1, lats1** (*ArrayLike*) – Coordinates of terminal points, [deg]

Returns

ArrayLike – Distances between corresponding points. $[m]$

Notes

This formula does not take into account the non-spheroidal (ellipsoidal) shape of the Earth. Originally referenced from https://andrew.hedges.name/experiments/haversine/.

References

- [*Calculate Distance and Bearing between Two Latitude/Longitude Points Using Haversine Formula in JavaScript*, n.d.]

See also:

`sklearn.metrics.pairwise.haversine_distances()`
Compute the Haversine distance

`pyproj.Geod`
Performs forward and inverse geodetic, or Great Circle, computations

pycontrails.physics.geo.**hours_since_start_of_day**(*time*)

Calculate the hours elapsed since the start of day (00:00:00 UTC).

Parameters

time (*ArrayLike*) – ArrayLike of `np.datetime64` times

Returns
ArrayLike – Hours elapsed since the start of today day. Output `dtype` is `np.float64`.

`pycontrails.physics.geo.`**`longitudinal_angle`**(*lons0*, *lats0*, *lons1*, *lats1*)

Calculate angle with longitudinal axis for sequence of segments.

Parameters

- **lons0** (`npt.NDArray[np.float64]`) – Longitude values of initial endpoints, [deg].
- **lats0** (`npt.NDArray[np.float64]`) – Latitude values of initial endpoints, [deg].
- **lons1** (`npt.NDArray[np.float64]`) – Longitude values of terminal endpoints, [deg].
- **lats1** (`npt.NDArray[np.float64]`) – Latitude values of terminal endpoints, [deg].

References

- [Wikipedia contributors, 2023]

Returns

- **sin_a** (`npt.NDArray[np.float64]`) – Sine values.
- **cos_a** (`npt.NDArray[np.float64]`) – Cosine values.

`pycontrails.physics.geo.`**`orbital_correction_for_solar_hour_angle`**(*theta_rad*)

Calculate correction to the solar hour angle due to Earth's orbital location.

Parameters
theta_rad (*ArrayLike*) – Orbital position, [rad]

Returns
ArrayLike – Correction to the solar hour angle as a result of Earth's orbital location, [deg]

References

- [Paltridge *et al.*, 1976]

Notes

Tested against [NOAA, n.d.]

`pycontrails.physics.geo.`**`orbital_position`**(*time*)

Calculate the orbital position of Earth to a reference point set at the start of year.

Parameters
time (*ArrayLike*) – ArrayLike of `np.datetime64` times

Returns
ArrayLike – Orbital position of Earth, [rad]

`pycontrails.physics.geo.`**`segment_angle`**(*longitude*, *latitude*)

Calculate the angle between coordinate segments and the longitudinal axis.

np.nan is added to the final value so the length of the output is the same as the inputs.

Parameters

- **longitude** (npt.NDArray[np.float64]) – Longitude values, [deg]
- **latitude** (npt.NDArray[np.float64]) – Latitude values, [deg]

Returns

tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]] – sin(a), cos(a), where a is the angle between the segment and the longitudinal axis. Final entry of each array is set to *np.nan*.

References

- [Wikipedia contributors, 2023]

Notes

```
        (lon_2, lat_2)  X
                       /|
                      / |
                     /  |
                    /   |
                   /    |
                  /     |
                 /      |
                /       |
(lon_1, lat_1)  X -------> longitude (x-axis)
```

See also:

longitudinal_angle()

pycontrails.physics.geo.**segment_azimuth**(*longitude*, *latitude*)

Calculate the angle between coordinate segments and true north.

np.nan is added to the final value so the length of the output is the same as the inputs.

Parameters

- **longitude** (npt.NDArray[np.float64]) – Longitude values, [deg]
- **latitude** (npt.NDArray[np.float64]) – Latitude values, [deg]

Returns

npt.NDArray[np.float64] – Azimuth relative to true north (0 deg), [deg] Final entry of each array is set to *np.nan*.

References

- [Wikipedia contributors, 2023]

See also:

azimuth()

pycontrails.physics.geo.**segment_haversine**(*longitude*, *latitude*)

Calculate haversine distance between consecutive points along path.

Parameters

- **longitude** (npt.NDArray[np.float64]) – 1D Longitude values with index corresponding to latitude inputs, [deg]
- **latitude** (npt.NDArray[np.float64]) – 1D Latitude values with index corresponding to longitude inputs, [deg]

Returns

npt.NDArray[np.float64] – Haversine distance between (lat_i, lon_i) and (lat_i+1, lon_i+1), [m] The final entry of the output is set to nan.

See also:

pyproj.Geod.line_lengths()

pycontrails.physics.geo.**segment_length**(*longitude*, *latitude*, *altitude*)

Calculate the segment length between coordinates by assuming a great circle distance.

Requires coordinates to be in EPSG:4326. Lengths are calculated using both horizontal and vertical displacement of segments.

np.nan is added to the final value so the length of the output is the same as the inputs.

Parameters

- **longitude** (npt.NDArray[np.float64]) – Longitude values, [deg]
- **latitude** (npt.NDArray[np.float64]) – Latitude values, [deg]
- **altitude** (npt.NDArray[np.float64]) – Altitude values, [m]

Returns

npt.NDArray[np.float64] – Array of distances in [m] between coordinates. Final entry of each array is set to *np.nan*.

See also:

haversine(), *segment_haversine()*

pycontrails.physics.geo.**solar_constant**(*theta_rad*)

Calculate the solar electromagnetic radiation per unit area from orbital position.

On average, the extraterrestrial irradiance is 1367 W/m**2 and varies by +- 3% as the Earth orbits the sun.

Parameters

theta_rad (*ArrayLike*) – Orbital position, [rad]. Use *orbital_position()* to calculate the orbital position from time input.

Returns

ArrayLike – Solar constant, [Wm^{-2}]

References

- [Laboratory, 2022]
- [Paltridge *et al.*, 1976]
- [Duffie and Beckman, 1991]

Notes

$orbital_effect = (R_{av}/R)^2$ where R is the separation of Earth from the sun and R_{av} is the mean separation.

pycontrails.physics.geo.**solar_declination_angle**(*theta_rad*)

Calculate the solar declination angle from the orbital position in radians (theta_rad).

The solar declination angle is the angle between the rays of the Sun and the plane of the Earth's equator.

It has a range of between -23.5 (winter solstice) and +23.5 (summer solstice) degrees.

Parameters
theta_rad (*ArrayLike*) – Orbital position, [rad]. Output of *orbital_position()*.

Returns
ArrayLike – Solar declination angle, [deg]

References

- [Paltridge *et al.*, 1976]

Notes

Tested against [NOAA, n.d.]

See also:

orbital_position(), *cosine_solar_zenith_angle()*

pycontrails.physics.geo.**solar_direct_radiation**(*longitude*, *latitude*, *time*, *threshold_cos_sza=0.0*)

Calculate the instantaneous theoretical solar direct radiation (SDR).

Parameters

- **longitude** (*ArrayLike*) – Longitude, [deg]
- **latitude** (*ArrayLike*) – Latitude, [deg]
- **time** (*ArrayLike*) – Time, formatted as `np.datetime64`
- **threshold_cos_sza** (`float`, *optional*) – Set the SDR to 0 when the *cosine_solar_zenith_angle()* is below a certain value. By default, set to 0.

Returns
ArrayLike – Solar direct radiation of incoming radiation, [Wm^{-2}]

References

- [Laboratory, 2022]

pycontrails.physics.geo.**solar_hour_angle**(*longitude*, *time*, *theta_rad*)

Calculate the sun's East to West angular displacement around the polar axis.

The solar hour angle is an expression of time in angular measurements: the value of the hour angle is zero at noon, negative in the morning, and positive in the afternoon, increasing by 15 degrees per hour.

Parameters

- **longitude** (*ArrayLike*) – Longitude, [deg]

- **time** (*ArrayLike*) – ArrayLike of `np.datetime64` times
- **theta_rad** (*ArrayLike*) – Orbital position, [rad]. Output of *orbital_position()*.

Returns

ArrayLike – Solar hour angle, [deg]

See also:

orbital_position(), *cosine_solar_zenith_angle()*, *orbital_correction_for_solar_hour_angle()*

pycontrails.physics.geo.**spatial_bounding_box**(*longitude*, *latitude*, *buffer=1.0*)

Construct rectangular spatial bounding box from a set of waypoints.

Parameters

- **longitude** (`numpy.ndarray`) – 1D Longitude values with index corresponding to longitude inputs, [deg]
- **latitude** (`numpy.ndarray`) – 1D Latitude values with index corresponding to latitude inputs, [deg]
- **buffer** (`float`) – Add buffer to rectangular spatial bounding box, [deg]

Returns

`tuple[float, float, float, float]` – Spatial bounding box, `(lon_min, lat_min, lon_max, lat_max)`, [deg]

Examples

```
>>> rng = np.random.default_rng(654321)
>>> lon = rng.uniform(-180, 180, size=30)
>>> lat = rng.uniform(-90, 90, size=30)
>>> spatial_bounding_box(lon, lat)
(-168.0, -77.0, 155.0, 82.0)
```

11.4.5 pycontrails.physics.units

Unit conversion support.

Functions

degrees_to_radians(degrees)	Convert from degrees to radians.
dt_to_seconds(dt[, dtype])	Convert a time delta to seconds as a float with specified `dtype` precision.
ft_to_m(ft)	Convert length from feet to meter.
ft_to_pl(h)	Convert from altitude (ft) to pressure level (hPa).
kelvin_to_celsius(kelvin)	Convert temperature from Kelvin to Celsius.
knots_to_m_per_s(knots)	Convert speed from knots to meters per second (m/s).
latitude_distance_to_m(distance_degrees)	Convert latitude degrees distance between two points to cartesian distances in meters.
lbs_to_kg(lbs)	Convert mass from pounds (lbs) to kilograms (kg).
longitude_distance_to_m(distance_degrees, ...)	Convert longitude degrees distance between two points to cartesian distances in meters.
m_per_s_to_knots(m_per_s)	Convert speed from meters per second (m/s) to knots.
m_to_T_isa(h)	Calculate the ambient temperature (K) for a given altitude (m).
m_to_ft(m)	Convert length from meters to feet.
m_to_latitude_distance(distance_m)	Convert cartesian distance (meters) to differences in latitude degrees.
m_to_longitude_distance(distance_m, ...)	Convert cartesian distance (meters) to differences in longitude degrees.
m_to_pl(h)	Convert from altitude (m) to pressure level (hPa).
mach_number_to_tas(mach_number, T)	Calculate true airspeed from the Mach number at a specified ambient temperature.
pl_to_ft(pl)	Convert from pressure level (hPa) to altitude (ft).
pl_to_m(pl)	Convert from pressure level (hPa) to altitude (m).
radians_to_degrees(radians)	Convert from radians to degrees.
tas_to_mach_number(true_airspeed, T)	Calculate Mach number from true airspeed at a specified ambient temperature.

pycontrails.physics.units.**degrees_to_radians**(*degrees*)

Convert from degrees to radians.

Parameters
degrees (*ArrayScalarLike*) – Degrees values, [deg]

Returns
ArrayScalarLike – Radians values

pycontrails.physics.units.**dt_to_seconds**(*dt*, *dtype=<class 'numpy.float64'>*)

Convert a time delta to seconds as a float with specified `dtype` precision.

Parameters

- **dt** (numpy.ndarray) – Time delta for each waypoint
- **dtype** (np.dtype) – Data type of the output array

Returns
numpy.ndarray – Time delta in seconds as a float

pycontrails.physics.units.**ft_to_m**(*ft*)

Convert length from feet to meter.

Parameters
ft (*ArrayScalarLike*) – length, [ft]

Returns
ArrayScalarLike – length, [m]

pycontrails.physics.units.**ft_to_pl**(*h*)

Convert from altitude (ft) to pressure level (hPa).

Assumes the ICAO standard atmosphere.

Parameters
h (*ArrayScalarLike*) – altitude, [ft]

Returns
ArrayScalarLike – pressure level, [hPa], [$mbar$]

See also:

m_to_pl, *pl_to_ft*, *m_to_T_isa*

pycontrails.physics.units.**kelvin_to_celsius**(*kelvin*)

Convert temperature from Kelvin to Celsius.

Parameters
kelvin (*ArrayScalarLike*) – temperature [K]

Returns
ArrayScalarLike – temperature [C]

pycontrails.physics.units.**knots_to_m_per_s**(*knots*)

Convert speed from knots to meters per second (m/s).

Parameters
knots (*ArrayScalarLike*) – Speed, [$knots$]

Returns
ArrayScalarLike – Speed, [$m\ s^{-1}$]

pycontrails.physics.units.**latitude_distance_to_m**(*distance_degrees*)

Convert latitude degrees distance between two points to cartesian distances in meters.

Parameters
distance_degrees (*ArrayScalarLike*) – latitude distance, [deg]

Returns
ArrayScalarLike – Cartesian distance along the latitude axis, [m]

pycontrails.physics.units.**lbs_to_kg**(*lbs*)

Convert mass from pounds (lbs) to kilograms (kg).

Parameters
lbs (*ArrayScalarLike*) – mass, pounds [lbs]

Returns
ArrayScalarLike – mass, kilograms [kg]

pycontrails.physics.units.**longitude_distance_to_m**(*distance_degrees*, *latitude_mean*)

Convert longitude degrees distance between two points to cartesian distances in meters.

Parameters

- **distance_degrees** (*ArrayScalarLike*) – longitude distance, [deg]

- **latitude_mean** (*ArrayScalarLike*, *optional*) – mean latitude between `longitude_1` and `longitude_2`, [deg]

Returns
ArrayScalarLike – cartesian distance along the longitude axis, [m]

`pycontrails.physics.units.`**`m_per_s_to_knots`**(*m_per_s*)

Convert speed from meters per second (m/s) to knots.

Parameters
m_per_s (*ArrayScalarLike*) – Speed, [$m\ s^{-1}$]

Returns
ArrayScalarLike – Speed, [$knots$]

`pycontrails.physics.units.`**`m_to_T_isa`**(*h*)

Calculate the ambient temperature (K) for a given altitude (m).

Assumes the ICAO standard atmosphere.

Parameters
h (*ArrayScalarLike*) – altitude, [m]

Returns
ArrayScalarLike – ICAO standard atmosphere ambient temperature, [K]

References

- [Wikipedia contributors, 2023]

Notes

See https://en.wikipedia.org/wiki/International_Standard_Atmosphere

See also:

m_to_pl, *ft_to_pl*

`pycontrails.physics.units.`**`m_to_ft`**(*m*)

Convert length from meters to feet.

Parameters
m (*ArrayScalarLike*) – length, [m]

Returns
ArrayScalarLike – length, [ft]

`pycontrails.physics.units.`**`m_to_latitude_distance`**(*distance_m*)

Convert cartesian distance (meters) to differences in latitude degrees.

Small angle approximation for `distance_m` << `constants.radius_earth`

Parameters
distance_m (*ArrayScalarLike*) – cartesian distance along latitude axis, [m]

Returns
ArrayScalarLike – latitude distance, [deg]

pycontrails.physics.units.**m_to_longitude_distance**(*distance_m*, *latitude_mean*)

Convert cartesian distance (meters) to differences in longitude degrees.

Small angle approximation for `distance_m << constants.radius_earth`

Parameters

- **distance_m** (*ArrayScalarLike*) – cartesian distance along longitude axis, [m]
- **latitude_mean** (*ArrayScalarLike*) – mean latitude between `longitude_1` and `longitude_2`, [deg]

Returns

ArrayScalarLike – longitude distance, [deg]

pycontrails.physics.units.**m_to_pl**(*h*)

Convert from altitude (m) to pressure level (hPa).

Parameters

h (`npt.NDArray[np.float64]`) – altitude, [m]

Returns

`npt.NDArray[np.float64]` – pressure level, [hPa], [$mbar$]

References

- [Wikipedia contributors, 2023]

Notes

See https://en.wikipedia.org/wiki/Barometric_formula

See also:

m_to_T_isa, *ft_to_pl*

pycontrails.physics.units.**mach_number_to_tas**(*mach_number*, *T*)

Calculate true airspeed from the Mach number at a specified ambient temperature.

Parameters

- **mach_number** (`float | npt.NDArray[np.float64]`) – Mach number, [:math: Ma]
- **T** (`npt.NDArray[np.float64]`) – Ambient temperature, [K]

Returns

`npt.NDArray[np.float64]` – True airspeed, [$m\ s^{-1}$]

References

- [Cumpsty and Heyes, 2015]

pycontrails.physics.units.**pl_to_ft**(*pl*)

Convert from pressure level (hPa) to altitude (ft).

Assumes the ICAO standard atmosphere.

Parameters

pl (*ArrayScalarLike*) – pressure level, [hPa], [$mbar$]

Returns
ArrayScalarLike – altitude, [ft]

See also:

pl_to_m, *ft_to_pl*, *m_to_T_isa*

pycontrails.physics.units.**pl_to_m**(*pl*)

Convert from pressure level (hPa) to altitude (m).

Function is slightly different from the classical formula: `constants.T_msl / 0.0065) * (1 - (pl_pa / constants.p_surface) ** (1 / 5.255)` in order to provide a mathematical inverse to *m_to_pl()*.

For low altitudes (below the tropopause), this implementation closely agrees to classical formula.

Parameters
pl (*ArrayLike*) – pressure level, [hPa], [$mbar$]

Returns
ArrayLike – altitude, [m]

References

- [Wikipedia contributors, 2023]

Notes

See https://en.wikipedia.org/wiki/Barometric_formula

See also:

pl_to_ft, *m_to_pl*, *m_to_T_isa*

pycontrails.physics.units.**radians_to_degrees**(*radians*)

Convert from radians to degrees.

Parameters
radians (*ArrayScalarLike*) – degrees values, []

Returns
ArrayScalarLike – Radian values

pycontrails.physics.units.**tas_to_mach_number**(*true_airspeed*, *T*)

Calculate Mach number from true airspeed at a specified ambient temperature.

Parameters

- **true_airspeed** (*ArrayScalarLike*) – True airspeed, [$m\ s^{-1}$]
- **T** (*ArrayScalarLike*) – Ambient temperature, [K]

Returns
ArrayScalarLike – Mach number, [:math: Ma]

References

- [Cumpsty and Heyes, 2015]

11.5 Cache

DiskCacheStore([cache_dir, allow_clear])	Cache that uses a folder on the local filesystem.
GCPCacheStore([cache_dir, project, bucket, ...])	Google Cloud Platform (Storage) Cache.

11.5.1 pycontrails.DiskCacheStore

class pycontrails.**DiskCacheStore**(*cache_dir=None*, *allow_clear=False*)

Bases: *CacheStore*

Cache that uses a folder on the local filesystem.

Parameters

- **allow_clear** (bool, *optional*) – Allow this cache to be cleared using *clear()*. Defaults to False.
- **cache_dir** (str | pathlib.Path, *optional*) – Root cache directory. By default, looks first for PYCONTRAILS_CACHE_DIR environment variable, then uses the OS specific platformdirs.user_cache_dir() function.

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> disk_cache.cache_dir
'cache'
```

```
>>> disk_cache.clear()  # cleanup
```

__init__(*cache_dir=None*, *allow_clear=False*)

Methods

__init__([cache_dir, allow_clear])	
clear([cache_path])	Delete all files and folders within cache_path.
exists(cache_path)	Check if a path in cache exists.
get(cache_path)	Get data path from the local cache store.
listdir([path])	List the contents of a directory in the cache.
path(cache_path)	Return a full filepath in cache.
put(data_path[, cache_path])	Save data to the local cache store.
put_multiple(data_path, cache_path)	Put multiple files into the cache at once.

Attributes

cache_dir	
allow_clear	
size	Return the disk size (in MBytes) of the local cache.

allow_clear

cache_dir

clear(*cache_path=''*)

Delete all files and folders within `cache_path`.

If no `cache_path` is provided, this will clear the entire cache.

If *allow_clear* is set to `False`, this method will do nothing.

Parameters
cache_path (`str`, *optional*) – Path to subdirectory or file in cache

Raises
`RuntimeError` – Raises a RuntimeError when *allow_clear* is set to `False`

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
```

```
>>> # Write some data to the cache
>>> disk_cache.put("README.md", "test/example.txt")
'test/example.txt'
```

```
>>> disk_cache.exists("test/example.txt")
True
```

```
>>> # clear a specific path
>>> disk_cache.clear("test/example.txt")
```

```
>>> # clear the whole cache
>>> disk_cache.clear()
```

exists(*cache_path*)

Check if a path in cache exists.

Parameters
cache_path (`str`) – Path to directory or file in cache

Returns
`bool` – True if directory or file exists

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.exists("file.nc")
False
```

get(*cache_path*)

Get data path from the local cache store.

Alias for *path()*

Parameters
cache_path (str) – Cache path to retrieve

Returns
str – Returns the relative path in the cache to the stored file

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>>
>>> # returns a path
>>> disk_cache.get("test/file.md")
'cache/test/file.md'
```

listdir(*path=''*)

List the contents of a directory in the cache.

Parameters
path (str) – Path to the directory to list

Returns
list[str] – List of files in the directory

path(*cache_path*)

Return a full filepath in cache.

Parameters
cache_path (str) – string path or filepath to create in cache If parent directories do not exist, they will be created.

Returns
str – Full path string to subdirectory directory or object in cache directory

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.path("file.nc")
'cache/file.nc'
```

```
>>> cache.clear()  # cleanup
```

put(*data_path*, *cache_path=None*)

Save data to the local cache store.

Parameters

- **data_path** (`str | pathlib.Path`) – Path to data to cache.
- **cache_path** (`str | None`, *optional*) – Path in cache store to save data Defaults to the same filename as `data_path`

Returns

`str` – Returns the relative path in the cache to the stored file

Raises

`FileNotFoundError` – Raises if *data* is a string and a file is not found at the string

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>>
>>> # put a file directly
>>> disk_cache.put("README.md", "test/file.md")
'test/file.md'
```

property size

Return the disk size (in MBytes) of the local cache.

Returns

`float` – Size of the disk cache store in MB

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.size
0.0...
```

```
>>> cache.clear()  # cleanup
```

11.5.2 pycontrails.GCPCacheStore

class pycontrails.**GCPCacheStore**(*cache_dir=''*, *project=None*, *bucket=None*, *disk_cache=None*, *read_only=True*, *allow_clear=False*, *timeout=300*, *show_progress=False*, *chunk_size=16777216*)

Bases: *CacheStore*

Google Cloud Platform (Storage) Cache.

This class downloads files from Google Cloud Storage locally to a *DiskCacheStore* initialized with `cache_dir=".gcp"` to avoid re-downloading files. If the source files on GCP changes, the local mirror of the GCP DiskCacheStore must be cleared by initializing this class and running *clear_disk()*.

Note by default, GCP Cache Store is *read only*. When a *put()* is called and *read_only* is set to *True*, the cache will throw an `RuntimeError` error. Set `read_only` to *False* to enable writing to cache store.

Parameters

- **cache_dir** (`str`, *optional*) – Root object prefix within *bucket* Defaults to `PYCONTRAILS_CACHE_DIR` environment variable, or the root of the bucket. The full GCP URI (ie, *"gs://<MY_BUCKET>/<PREFIX>"*) can be used here.
- **project** (*str , *optional**) – GCP Project. Defaults to the current active project set in the *google-cloud-sdk* environment
- **bucket** (`str`, *optional*) – GCP Bucket to use for cache. Defaults to `PYCONTRAILS_CACHE_BUCKET` environment variable.
- **read_only** (`bool`, *optional*) – Only enable reading from cache. Defaults to `True`.
- **allow_clear** (`bool`, *optional*) – Allow this cache to be cleared using `clear()`. Defaults to `False`.
- **disk_cache** (*DiskCacheStore*, *optional*) – Specify a custom local disk cache store to mirror files. Defaults to `DiskCacheStore(cache_dir="{user_cache_dir}/.gcp/{bucket}")`
- **show_progress** (`bool`, *optional*) – Show progress bar on cache *put()*. Defaults to False
- **chunk_size** (`int`, *optional*) – Chunk size for uploads and downloads with progress. Set a larger size to see more granular progress, and set a smaller size for more optimal download speed. Chunk size must be a multiple of 262144 (ie, 10 * 262144). Default value is 8 * 262144, which will throttle fast download speeds.

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
... )
>>> cache.cache_dir
'cache/'
>>> cache.bucket
'contrails-301217-unit-test'
```

__init__(*cache_dir=''*, *project=None*, *bucket=None*, *disk_cache=None*, *read_only=True*, *allow_clear=False*, *timeout=300*, *show_progress=False*, *chunk_size=16777216*)

Methods

__init__([cache_dir, project, bucket, ...])	
clear_disk([cache_path])	Clear the local disk cache mirror of the GCP Cache Store.
exists(cache_path)	Check if a path in cache exists.
get(cache_path)	Get data from the local cache store.
gs_path(cache_path)	Return a full Google Storage (gs://) URI to object.
listdir([path])	List the contents of a directory in the cache.
path(cache_path)	Return a full filepath in cache.
put(data_path[, cache_path])	Save data to the GCP cache store.
put_multiple(data_path, cache_path)	Put multiple files into the cache at once.

Attributes

project	
bucket	
read_only	
timeout	
show_progress	
chunk_size	
allow_clear	
cache_dir	
client	Handle to Google Cloud Storage client.
size	Return the disk size (in MBytes) of the local cache.

bucket

chunk_size

clear_disk(*cache_path=''*)

Clear the local disk cache mirror of the GCP Cache Store.

Parameters

cache_path (str, *optional*) – Path in mirrored cache store. Passed into `_disk_clear.clear()`. By default, this method will clear the entire mirrored cache store.

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
... )
>>> cache.clear_disk()
```

property client

Handle to Google Cloud Storage client.

Returns
`google.cloud.storage.Client` – Handle to Google Cloud Storage client

exists(*cache_path*)

Check if a path in cache exists.

Parameters
cache_path (`str`) – Path to directory or file in cache

Returns
`bool` – True if directory or file exists

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.exists("file.nc")
False
```

get(*cache_path*)

Get data from the local cache store.

Parameters
cache_path (`str`) – Path in cache store to get data

Returns
`str` – Returns path to downloaded local file

Raises
`ValueError` – Raises value error is `cache_path` refers to a directory

Examples

```
>>> import pathlib
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
...     read_only=False,
... )
```

```
>>> cache.put("README.md", "example/file.md")
'example/file.md'
```

```
>>> # returns a full path to local copy of the file
>>> path = cache.get("example/file.md")
>>> pathlib.Path(path).is_file()
True
```

```
>>> pathlib.Path(path).read_text()[17:69]
'Python library for modeling aviation climate impacts'
```

gs_path(*cache_path*)

Return a full Google Storage (gs://) URI to object.

Parameters
cache_path (str) – string path to object in cache

Returns
str – Google Storage URI (gs://) to object in cache

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
... )
>>> cache.path("file.nc")
'cache/file.nc'
```

listdir(*path=''*)

List the contents of a directory in the cache.

Parameters
path (str) – Path to the directory to list

Returns
list[str] – List of files in the directory

path(*cache_path*)

Return a full filepath in cache.

Parameters
cache_path (str) – string path or filepath to create in cache If parent directories do not exist, they will be created.

Returns
str – Full path string to subdirectory directory or object in cache directory

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.path("file.nc")
'cache/file.nc'
```

```
>>> cache.clear()  # cleanup
```

project

put(*data_path*, *cache_path=None*)

Save data to the GCP cache store.

If *read_only* is *True*, this method will return the path to the local disk cache store path.

Parameters

- **data_path** (`str | pathlib.Path`) – Data to save to GCP cache store.
- **cache_path** (`str`, *optional*) – Path in cache store to save data. Defaults to the same filename as `data_path`.

Returns

`str` – Returns the path in the cache to the stored file

Raises

- `RuntimeError` – Raises if *read_only* is True
- `FileNotFoundError` – Raises if `data` is a string and a file is not found at the string

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
...     read_only=False,
... )
```

```
>>> # put a file directly
>>> cache.put("README.md", "test/file.md")
'test/file.md'
```

read_only

show_progress

property size

Return the disk size (in MBytes) of the local cache.

Returns

`float` – Size of the disk cache store in MB

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.size
0.0...
```

```
>>> cache.clear()  # cleanup
```

timeout

11.6 Core

core.airports	Airport data support.
core.cache	Pycontrails Caching Support.
core.coordinates	Coordinates utilities.
core.datalib	Datalib utilities.
core.fleet	A single data structure encompassing a sequence of *Flight* instances.
core.flight	Flight Data Handling.
core.fuel	Fuel data support.
core.interpolation	Interpolation utilities.
core.met	Meteorology data models.
core.met_var	Module containing core met variables.
core.models	Physical model data structures.
core.polygon	Algorithm support for grid to polygon conversion.
core.vector	Lightweight data structures for vector paths.

11.6.1 pycontrails.core.airports

Airport data support.

Module Attributes

OURAIRPORTS_DATABASE_URL	URL for Our Airports database.

Functions

distance_to_airports(airports, longitude, ...)	Calculate the 3D distance from the waypoint to the provided airports.
find_nearest_airport(airports, longitude, ...)	Find airport nearest to the waypoints.
global_airport_database([cachestore, ...])	Load and process global airport database from Our Airports.

pycontrails.core.airports.**OURAIRPORTS_DATABASE_URL = 'https://github.com/contrailcirrus/ourairports-data/raw/main/airports.csv'**

URL for Our Airports database. Fork of the ourairports-data repository.

pycontrails.core.airports.**distance_to_airports**(*airports*, *longitude*, *latitude*, *altitude*)

Calculate the 3D distance from the waypoint to the provided airports.

Parameters

- **airports** (pandas.DataFrame) – Airport database in the format returned from *global_airport_database()*.
- **longitude** (float) – Waypoint longitude, [deg]
- **latitude** (float) – Waypoint latitude, [deg]
- **altitude** (float) – Waypoint altitude, [m]

Returns

numpy.ndarray – 3D distance from waypoint to airports, [m]

See also:

geo.haversine()

pycontrails.core.airports.**find_nearest_airport**(*airports*, *longitude*, *latitude*, *altitude*, ***, *bbox=2.0*)

Find airport nearest to the waypoints.

Parameters

- **airports** (pandas.DataFrame) – Airport database in the format returned from *global_airport_database()*.
- **longitude** (float) – Waypoint longitude, [deg]
- **latitude** (float) – Waypoint latitude, [deg]
- **altitude** (float) – Waypoint altitude, [m]
- **bbox** (float) – Search airports within spatial bounding box of $\pm$ *bbox* from the waypoint, [deg] Defaults to 2 deg

Returns

str – ICAO code of nearest airport. Returns None if no airport is found within bbox.

Notes

Function will first search for large airports around the waypoint vicinity. If none is found, it will search for medium and small airports around the waypoint vicinity.

The waypoint must be below 10,000 feet to increase the probability of identifying the correct airport.

pycontrails.core.airports.**global_airport_database**(*cachestore=None*, *update_cache=False*)

Load and process global airport database from Our Airports.

The database includes coordinates and metadata for 74867 unique airports.

Parameters

- **cachestore** (cache.CacheStore | None, *optional*) – Cache store for airport database. Defaults to cache.DiskCacheStore.
- **update_cache** (bool, *optional*) – Force update to cached airports database.

Returns

`pandas.DataFrame` – Processed global airport database.

Global airport database.

Notes

As of 2023 March 30, the global airport database contains:

Airport Type	Number
small_airport	39327
heliport	19039
closed	10107
medium_airport	4753
seaplane_base	1133
large_airport	463
balloonport	45

References

- [Megginson, 2023]

11.6.2 pycontrails.core.cache

Pycontrails Caching Support.

Classes

CacheStore()	Abstract cache storage class for storing staged and intermediate data.
DiskCacheStore([cache_dir, allow_clear])	Cache that uses a folder on the local filesystem.
GCPCacheStore([cache_dir, project, bucket, ...])	Google Cloud Platform (Storage) Cache.

class pycontrails.core.cache.**CacheStore**

Bases: `ABC`

Abstract cache storage class for storing staged and intermediate data.

allow_clear

cache_dir

abstract exists(*cache_path*)

Check if a path in cache exists.

Parameters
cache_path (`str`) – Path to directory or file in cache

Returns
`bool` – True if directory or file exists

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.exists("file.nc")
False
```

abstract get(*cache_path*)

Get data from cache.

abstract listdir(*path=''*)

List the contents of a directory in the cache.

Parameters
path (str) – Path to the directory to list

Returns
list[str] – List of files in the directory

abstract path(*cache_path*)

Return a full filepath in cache.

Parameters
cache_path (str) – string path or filepath to create in cache If parent directories do not exist, they will be created.

Returns
str – Full path string to subdirectory directory or object in cache directory

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.path("file.nc")
'cache/file.nc'
```

```
>>> cache.clear()  # cleanup
```

abstract put(*data*, *cache_path=None*)

Save data to cache.

put_multiple(*data_path*, *cache_path*)

Put multiple files into the cache at once.

Parameters

- **data_path** (Sequence[str | pathlib.Path]) – List of data files to cache. Each member is passed directly on to *put()*.
- **cache_path** (list[str]) – List of cache paths corresponding to each element in the data_path list. Each member is passed directly on to *put()*.

Returns
list[str] – Returns a list of relative paths to the stored files in the cache

abstract property size

Return the disk size (in MBytes) of the local cache.

Returns

`float` – Size of the disk cache store in MB

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.size
0.0...
```

```
>>> cache.clear()  # cleanup
```

class pycontrails.core.cache.**DiskCacheStore**(*cache_dir=None*, *allow_clear=False*)

Bases: *CacheStore*

Cache that uses a folder on the local filesystem.

Parameters

- **allow_clear** (`bool`, *optional*) – Allow this cache to be cleared using *clear()*. Defaults to False.
- **cache_dir** (`str | pathlib.Path`, *optional*) – Root cache directory. By default, looks first for `PYCONTRAILS_CACHE_DIR` environment variable, then uses the OS specific `platformdirs.user_cache_dir()` function.

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> disk_cache.cache_dir
'cache'
```

```
>>> disk_cache.clear()  # cleanup
```

allow_clear

cache_dir

clear(*cache_path=''*)

Delete all files and folders within `cache_path`.

If no `cache_path` is provided, this will clear the entire cache.

If *allow_clear* is set to `False`, this method will do nothing.

Parameters

cache_path (`str`, *optional*) – Path to subdirectory or file in cache

Raises

`RuntimeError` – Raises a RuntimeError when *allow_clear* is set to `False`

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
```

```
>>> # Write some data to the cache
>>> disk_cache.put("README.md", "test/example.txt")
'test/example.txt'
```

```
>>> disk_cache.exists("test/example.txt")
True
```

```
>>> # clear a specific path
>>> disk_cache.clear("test/example.txt")
```

```
>>> # clear the whole cache
>>> disk_cache.clear()
```

exists(*cache_path*)

Check if a path in cache exists.

Parameters
cache_path (str) – Path to directory or file in cache

Returns
bool – True if directory or file exists

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.exists("file.nc")
False
```

get(*cache_path*)

Get data path from the local cache store.

Alias for *path()*

Parameters
cache_path (str) – Cache path to retrieve

Returns
str – Returns the relative path in the cache to the stored file

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>>
>>> # returns a path
>>> disk_cache.get("test/file.md")
'cache/test/file.md'
```

listdir(*path=''*)

List the contents of a directory in the cache.

Parameters
path (`str`) – Path to the directory to list

Returns
`list[str]` – List of files in the directory

path(*cache_path*)

Return a full filepath in cache.

Parameters
cache_path (`str`) – string path or filepath to create in cache If parent directories do not exist, they will be created.

Returns
`str` – Full path string to subdirectory directory or object in cache directory

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.path("file.nc")
'cache/file.nc'
```

```
>>> cache.clear()  # cleanup
```

put(*data_path*, *cache_path=None*)

Save data to the local cache store.

Parameters

- **data_path** (`str | pathlib.Path`) – Path to data to cache.
- **cache_path** (`str | None`, *optional*) – Path in cache store to save data Defaults to the same filename as `data_path`

Returns
`str` – Returns the relative path in the cache to the stored file

Raises
`FileNotFoundError` – Raises if *data* is a string and a file is not found at the string

Examples

```
>>> from pycontrails import DiskCacheStore
>>> disk_cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>>
>>> # put a file directly
>>> disk_cache.put("README.md", "test/file.md")
'test/file.md'
```

property size

Return the disk size (in MBytes) of the local cache.

Returns

`float` – Size of the disk cache store in MB

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.size
0.0...
```

```
>>> cache.clear()  # cleanup
```

class pycontrails.core.cache.**GCPCacheStore**(*cache_dir=''*, *project=None*, *bucket=None*, *disk_cache=None*, *read_only=True*, *allow_clear=False*, *timeout=300*, *show_progress=False*, *chunk_size=16777216*)

Bases: *CacheStore*

Google Cloud Platform (Storage) Cache.

This class downloads files from Google Cloud Storage locally to a *DiskCacheStore* initialized with `cache_dir=".gcp"` to avoid re-downloading files. If the source files on GCP changes, the local mirror of the GCP DiskCacheStore must be cleared by initializing this class and running *clear_disk()*.

Note by default, GCP Cache Store is *read only*. When a *put()* is called and *read_only* is set to *True*, the cache will throw an `RuntimeError` error. Set `read_only` to *False* to enable writing to cache store.

Parameters

- **cache_dir** (`str`, *optional*) – Root object prefix within *bucket* Defaults to `PYCONTRAILS_CACHE_DIR` environment variable, or the root of the bucket. The full GCP URI (ie, *"gs://<MY_BUCKET>/<PREFIX>"*) can be used here.
- **project** (*str , *optional**) – GCP Project. Defaults to the current active project set in the *google-cloud-sdk* environment
- **bucket** (`str`, *optional*) – GCP Bucket to use for cache. Defaults to `PYCONTRAILS_CACHE_BUCKET` environment variable.
- **read_only** (`bool`, *optional*) – Only enable reading from cache. Defaults to `True`.
- **allow_clear** (`bool`, *optional*) – Allow this cache to be cleared using `clear()`. Defaults to `False`.

- **disk_cache** (*DiskCacheStore*, *optional*) – Specify a custom local disk cache store to mirror files. Defaults to DiskCacheStore(cache_dir="{user_cache_dir}/.gcp/{bucket}")
- **show_progress** (bool, *optional*) – Show progress bar on cache *put()*. Defaults to False
- **chunk_size** (int, *optional*) – Chunk size for uploads and downloads with progress. Set a larger size to see more granular progress, and set a smaller size for more optimal download speed. Chunk size must be a multiple of 262144 (ie, 10 * 262144). Default value is 8 * 262144, which will throttle fast download speeds.

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
... )
>>> cache.cache_dir
'cache/'
>>> cache.bucket
'contrails-301217-unit-test'
```

bucket

chunk_size

clear_disk(*cache_path=''*)

Clear the local disk cache mirror of the GCP Cache Store.

Parameters

cache_path (str, *optional*) – Path in mirrored cache store. Passed into _disk_clear.clear(). By default, this method will clear the entire mirrored cache store.

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
... )
>>> cache.clear_disk()
```

property client

Handle to Google Cloud Storage client.

Returns

google.cloud.storage.Client – Handle to Google Cloud Storage client

exists(*cache_path*)

Check if a path in cache exists.

Parameters

cache_path (str) – Path to directory or file in cache

Returns
bool – True if directory or file exists

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.exists("file.nc")
False
```

get(*cache_path*)

Get data from the local cache store.

Parameters
cache_path (str) – Path in cache store to get data

Returns
str – Returns path to downloaded local file

Raises
ValueError – Raises value error is cache_path refers to a directory

Examples

```
>>> import pathlib
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
...     read_only=False,
... )
```

```
>>> cache.put("README.md", "example/file.md")
'example/file.md'
```

```
>>> # returns a full path to local copy of the file
>>> path = cache.get("example/file.md")
>>> pathlib.Path(path).is_file()
True
```

```
>>> pathlib.Path(path).read_text()[17:69]
'Python library for modeling aviation climate impacts'
```

gs_path(*cache_path*)

Return a full Google Storage (gs://) URI to object.

Parameters
cache_path (str) – string path to object in cache

Returns
str – Google Storage URI (gs://) to object in cache

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
... )
>>> cache.path("file.nc")
'cache/file.nc'
```

listdir(*path=''*)

List the contents of a directory in the cache.

Parameters
path (str) – Path to the directory to list

Returns
list[str] – List of files in the directory

path(*cache_path*)

Return a full filepath in cache.

Parameters
cache_path (str) – string path or filepath to create in cache If parent directories do not exist, they will be created.

Returns
str – Full path string to subdirectory directory or object in cache directory

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.path("file.nc")
'cache/file.nc'
```

```
>>> cache.clear()  # cleanup
```

project

put(*data_path*, *cache_path=None*)

Save data to the GCP cache store.

If *read_only* is *True*, this method will return the path to the local disk cache store path.

Parameters

- **data_path** (str | pathlib.Path) – Data to save to GCP cache store.
- **cache_path** (str, *optional*) – Path in cache store to save data. Defaults to the same filename as data_path.

Returns
str – Returns the path in the cache to the stored file

Raises

- RuntimeError – Raises if *read_only* is True

- **FileNotFoundError** – Raises if `data` is a string and a file is not found at the string

Examples

```
>>> from pycontrails import GCPCacheStore
>>> cache = GCPCacheStore(
...     bucket="contrails-301217-unit-test",
...     cache_dir="cache",
...     read_only=False,
... )
```

```
>>> # put a file directly
>>> cache.put("README.md", "test/file.md")
'test/file.md'
```

read_only

show_progress

property size

Return the disk size (in MBytes) of the local cache.

Returns

`float` – Size of the disk cache store in MB

Examples

```
>>> from pycontrails import DiskCacheStore
>>> cache = DiskCacheStore(cache_dir="cache", allow_clear=True)
>>> cache.size
0.0...
```

```
>>> cache.clear()  # cleanup
```

timeout

11.6.3 pycontrails.core.coordinates

Coordinates utilities.

Functions

intersect_domain(domain, request)	Return boolean mask of `request` that are within the bounds of `domain`.
slice_domain(domain, request[, buffer])	Return slice of `domain` containing coordinates overlapping `request`.

pycontrails.core.coordinates.**intersect_domain**(*domain*, *request*)

Return boolean mask of `request` that are within the bounds of `domain`.

Parameters

- **domain** (`numpy.ndarray`) – Full set of domain values
- **request** (`numpy.ndarray`) – Full set of requested values

Returns

`numpy.ndarray` – Boolean array of `request` values within the bounds of `domain`

Raises

`ValueError` – Raises a ValueError when `domain` has all `nan` values.

Examples

```
>>> domain = np.array([3.0, 4.0, 2.0])
>>> request = np.arange(1.0, 6.0)
>>> intersect_domain(domain, request)
array([False,  True,  True,  True, False])
```

```
>>> domain = np.array([3.0, np.nan, np.nan])
>>> request = np.arange(1.0, 6.0)
>>> intersect_domain(domain, request)
array([False, False,  True, False, False])
```

pycontrails.core.coordinates.**slice_domain**(*domain*, *request*, *buffer=(0.0, 0.0)*)

Return slice of `domain` containing coordinates overlapping `request`.

Computes index-based slice (to be used with `xarray.Dataset.isel()` method) as opposed to value-based slice.

Returns `slice(None, None)` when `domain` has a length <= 2 or the `request` has all `nan` values.

Changed in version 0.24.1: The returned slice is the minimum index-based slice that contains the requested coordinates. In other words, it's now possible for `domain[sl][0]` to equal `min(request)` where `sl` is the output of this function. Previously, we were guaranteed that `domain[sl][0]` would be less than `min(request)`.

Parameters

- **domain** (`numpy.ndarray`) – Full set of domain values
- **request** (`npt.ArrayLike`) – Requested values. Only the nanmin and nanmax values are considered.
- **buffer** (`tuple[float | np.timedelta64, float | np.timedelta64]`, *optional*) – Extend the domain past the requested coordinates by `buffer[0]` on the low side and `buffer[1]` on the high side. Units of `buffer` must be the same as `domain`.

Returns

`slice` – Slice object for slicing out encompassing, or nearest, domain values

Raises

`ValueError` – Raises a ValueError when `domain` has all `nan` values.

Examples

```
>>> domain = np.arange(-180, 180, 0.25)
```

```
>>> # Call with request as np.array
>>> request = np.linspace(-20, 20, 100)
>>> slice_domain(domain, request)
slice(640, 801, None)
```

```
>>> # Call with request as tuple
>>> request = -20, 20
>>> slice_domain(domain, request)
slice(640, 801, None)
```

```
>>> # Call with a buffer
>>> request = -16, 13
>>> buffer = 4, 7
>>> slice_domain(domain, request, buffer)
slice(640, 801, None)
```

```
>>> # Call with request as a single number
>>> request = -20
>>> slice_domain(domain, request)
slice(640, 641, None)
```

```
>>> request = -19.9
>>> slice_domain(domain, request)
slice(640, 642, None)
```

11.6.4 pycontrails.core.datalib

Datalib utilities.

Module Attributes

NETCDF_ENGINE	NetCDF engine to use for parsing netcdf files
DEFAULT_CHUNKS	Default chunking strategy when opening datasets with xarray
OPEN_IN_PARALLEL	Whether to open multi-file datasets in parallel
OPEN_WITH_LOCK	Whether to use file locking when opening multi-file datasets

Functions

parse_grid(grid, supported)	Parse input grid spacing.
parse_pressure_levels(pressure_levels[, ...])	Check input pressure levels are consistent type and ensure levels exist in ECMWF data source.
parse_timesteps(time[, freq])	Parse time input into set of time steps.
parse_variables(variables, supported)	Parse input variables.
round_hour(time, hour)	Round time to the nearest whole hour before input time.
validate_timestep_freq(freq, datasource_freq)	Check that input timestep frequency is compatible with the data source timestep frequency.

Classes

MetDataSource(time, variables[, ...])	Abstract class for wrapping meteorology data sources.

pycontrails.core.datalib.**DEFAULT_CHUNKS = {'time': 1}**

Default chunking strategy when opening datasets with xarray

class pycontrails.core.datalib.**MetDataSource**(*time*, *variables*, *pressure_levels=-1*, *paths=None*, *grid=None*, ***kwargs*)

Bases: ABC

Abstract class for wrapping meteorology data sources.

abstract cache_dataset(*dataset*)

Cache data from data source.

Parameters
dataset (xarray.Dataset) – Dataset loaded from remote API or local files. The dataset must have the same format as the original data source API or files.

cachestore

Cache store for intermediates while processing data source If None, cache is turned off.

abstract create_cachepath(*t*)

Return cachepath to local data file based on datetime.

Parameters
t (datetime) – Datetime of datafile

Returns
str – Path to cached data file

download(***xr_kwargs*)

Confirm all data files are downloaded and available locally in the *cachestore*.

Parameters
****xr_kwargs** – Passed into xarray.open_dataset() via *is_datafile_cached()*.

abstract download_dataset(*times*)

Download data from data source for input times.

Parameters
times (list[datetime]) – List of datetimes to download a store in cache

grid

Lat / Lon grid spacing

property hash

Generate a unique hash for this datasource.

Returns
str – Unique hash for met instance (sha1)

is_datafile_cached(*t*, ***xr_kwargs*)

Check datafile defined by datetime for variables and pressure levels in class.

If using a cloud cache store (i.e. cache.GCPCacheStore), this is where the datafile will be mirrored to a local file for access.

Parameters

- **t** (datetime) – Datetime of datafile
- ****xr_kwargs** (Any) – Additional kwargs passed directly to xarray.open_mfdataset() when opening files. By default, the following values are used if not specified:
  - chunks: {"time": 1}
  - engine: "netcdf4"
  - parallel: True

Returns
bool – True if data file exists for datetime with all variables and pressure levels, False otherwise

property is_single_level

Return True if the datasource is single level data.

New in version 0.50.0.

list_timesteps_cached(***xr_kwargs*)

Get a list of data files available locally in the *cachestore*.

Parameters
****xr_kwargs** – Passed into xarray.open_dataset() via *is_datafile_cached()*.

list_timesteps_not_cached(***xr_kwargs*)

Get a list of data files not available locally in the *cachestore*.

Parameters
****xr_kwargs** – Passed into xarray.open_dataset() via *is_datafile_cached()*.

open_dataset(*disk_paths*, ***xr_kwargs*)

Open multi-file dataset in xarray.

Parameters

- **disk_paths** (str | list[str] | pathlib.Path | list[pathlib.Path]) – list of string paths to local files to open
- ****xr_kwargs** (Any) – Additional kwargs passed directly to xarray.open_mfdataset() when opening files. By default, the following values are used if not specified:
  - chunks: {"time": 1}
  - engine: "netcdf4"

- parallel: False
- lock: False

Returns
xarray.Dataset – Open xarray dataset

abstract open_metdataset(*dataset=None*, *xr_kwargs=None*, ***kwargs*)

Open MetDataset from data source.

This method should download / load any required datafiles and returns a MetDataset of the multi-file dataset opened by xarray.

Parameters

- **dataset** (xr.Dataset | None, *optional*) – Input xr.Dataset loaded manually. The dataset must have the same format as the original data source API or files.
- **xr_kwargs** (dict[str, Any] | None, *optional*) – Dictionary of keyword arguments passed into xarray.open_mfdataset() when opening files. Examples include "chunks", "engine", "parallel", etc. Ignored if dataset is input.
- ****kwargs** (Any) – Keyword arguments passed through directly into MetDataset constructor.

Returns
MetDataset – Meteorology dataset

See also:

xarray.open_mfdataset()

paths

Path to local source files to load. Set to the paths of files cached in *cachestore* if no paths input is provided on init.

property pressure_level_variables

Parameters available from data source.

Returns
list[MetVariable] | None – List of MetVariable available in datasource

pressure_levels

List of pressure levels. Set to [-1] for data without level coordinate. Use *parse_pressure_levels()* to handle PressureLevelInput.

abstract set_metadata(*ds*)

Set met source metadata on ds.attrs.

This is called within the *open_metdataset()* method to set metadata on the returned MetDataset instance.

Parameters
ds (xr.Dataset | MetDataset) – Dataset to set metadata on. Mutated in place.

property single_level_variables

Parameters available from data source.

Returns
list[MetVariable] | None – List of MetVariable available in datasource

property supported_pressure_levels

Pressure levels available from datasource.

Returns
`list[int] | None` – List of integer pressure levels for class. If None, no pressure level information available for class.

property supported_variables

Parameters available from data source.

Returns
`list[MetVariable] | None` – List of MetVariable available in datasource

timesteps

List of individual timesteps from data source derived from `time` Use `parse_time()` to handle `TimeInput`.

property variable_shortnames

Return a list of variable short names.

Returns
`list[str]` – Lst of variable short names.

property variable_standardnames

Return a list of variable standard names.

Returns
`list[str]` – Lst of variable standard names.

variables

Variables requested from data source Use *parse_variables()* to handle `VariableInput`.

pycontrails.core.datalib.**NETCDF_ENGINE = 'netcdf4'**

NetCDF engine to use for parsing netcdf files

pycontrails.core.datalib.**OPEN_IN_PARALLEL = False**

Whether to open multi-file datasets in parallel

pycontrails.core.datalib.**OPEN_WITH_LOCK = False**

Whether to use file locking when opening multi-file datasets

pycontrails.core.datalib.**parse_grid**(*grid*, *supported*)

Parse input grid spacing.

Parameters

- **grid** (`float`) – Input grid float
- **supported** (`Sequence[float]`) – Sequence of support grid values

Returns
`float` – Parsed grid spacing

Raises
`ValueError` – Raises ValueError when `grid` is not in supported

pycontrails.core.datalib.**parse_pressure_levels**(*pressure_levels*, *supported=None*)

Check input pressure levels are consistent type and ensure levels exist in ECMWF data source.

Changed in version 0.50.0: The returned pressure levels are now sorted. Pressure levels must be unique. Raises ValueError if pressure levels have mixed signs.

Parameters

- **pressure_levels** (`PressureLevelInput`) – Input pressure levels for data, in hPa (mbar) Set to [-1] to represent surface level.
- **supported** (`list[int]`, *optional*) – List of supported pressures levels in data source

Returns
`list[int]` – List of integer pressure levels supported by ECMWF data source

Raises
`ValueError` – Raises ValueError if pressure level is not supported by ECMWF data source

pycontrails.core.datalib.**parse_timesteps**(*time*, *freq='1h'*)

Parse time input into set of time steps.

If input time is length 2, this creates a range of equally spaced time points between `[start, end]` with interval `freq`.

Parameters

- **time** (`TimeInput | None`) – Input datetime(s) specifying the time or time range of the data [start, end]. Either a single datetime-like or tuple of datetime-like with the first value the start of the date range and second value the end of the time range. Input values can be any type compatible with `pandas.to_datetime()`.
- **freq** (`str | None`, *optional*) – Timestep interval in range. See https://pandas.pydata.org/pandas-docs/stable/user_guide/timeseries.html#timeseries-offset-aliases for a list of frequency aliases. If None, returns input *time* as a list. Defaults to "1h".

Returns
`list[datetime]` – List of unique datetimes. If input `time` is None, returns an empty list

Raises
`ValueError` – Raises when the time has len > 2 or when time elements fail to be parsed with pd.to_datetime

pycontrails.core.datalib.**parse_variables**(*variables*, *supported*)

Parse input variables.

Changed in version 0.50.0: The output is no longer copied. Each `MetVariable` is a frozen dataclass, so copying is unnecessary.

Parameters

- **variables** (`VariableInput`) – Variable name, or sequence of variable names. i.e. `"air_temperature"`, `["air_temperature, relative_humidity"]`, `[130]`, `[AirTemperature]`, `[[EastwardWind, NorthwardWind]]` If an element is a list of MetVariable, the first MetVariable that is supported will be chosen.
- **supported** (`list[MetVariable]`) – Supported MetVariable.

Returns
`list[MetVariable]` – List of MetVariable

Raises
`ValueError` – Raises ValueError if variable is not supported

pycontrails.core.datalib.**round_hour**(*time*, *hour*)

Round time to the nearest whole hour before input time.

Parameters

- **time** (`datetime`) – Input time
- **hour** (`int`) – Hour to round down time

Returns
datetime – Rounded time

Raises
ValueError – Description

pycontrails.core.datalib.**validate_timestep_freq**(*freq*, *datasource_freq*)

Check that input timestep frequency is compatible with the data source timestep frequency.

A data source timestep frequency of 1 hour allows input timestep frequencies of 1 hour, 2 hours, 3 hours, etc., but not 1.5 hours or 30 minutes.

Parameters

- **freq** (str) – Input timestep frequency
- **datasource_freq** (str) – Datasource timestep frequency

Returns
bool – True if the input timestep frequency is an even multiple of the data source timestep frequency.

11.6.5 pycontrails.core.fleet

A single data structure encompassing a sequence of Flight instances.

Classes

Fleet([data, longitude, latitude, altitude, ...])	Data structure for holding a sequence of Flight instances.

class pycontrails.core.fleet.**Fleet**(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, *fuel=None*, *fl_attrs=None*, ***attrs_kwargs*)

Bases: *Flight*

Data structure for holding a sequence of Flight instances.

Flight waypoints are merged into a single Flight-like object.

clean_and_resample(*freq='1min'*, *fill_method='geodesic'*, *geodesic_threshold=100000.0*, *nominal_rocd=0.0*, *kernel_size=17*, *cruise_threshold=120*, *force_filter=False*, *drop=True*, *keep_original_index=False*, *climb_descend_at_end=False*)

Resample and (possibly) filter a flight trajectory.

Waypoints are resampled according to the frequency freq. If the original flight data has a short sampling period, *filter_altitude* will also be called to clean the data. Large gaps in trajectories may be interpolated as step climbs through *_altitude_interpolation*.

Parameters

- **freq** (str, *optional*) – Resampling frequency, by default "1min"
- **fill_method** ({"geodesic", "linear"}, *optional*) – Choose between "geodesic" and "linear", by default "geodesic". In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.

- **geodesic_threshold** (`float`, *optional*) – Threshold for geodesic interpolation, [m]. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (`float`, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to `constants.nominal_rocd`.
- **kernel_size** (`int`, *optional*) – Passed directly to `scipy.signal.medfilt()`, by default 11. Passed also to `scipy.signal.medfilt()`
- **cruise_theshold** (`float`, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter
- **force_filter** (`bool`, *optional*) – If set to true, meth:*filter_altitude* will always be called. otherwise, it will only be called if the flight has a median sample period under 10 seconds
- **drop** (`bool`, *optional*) – Drop any columns that are not resampled and filled. Defaults to `True`, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with `nan` or `None` values, depending on the data type.
- **keep_original_index** (`bool`, *optional*) – Keep the original index of the `Flight` in addition to the new resampled index. Defaults to `False`. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (`bool`) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns
Flight – Filled Flight

copy(***kwargs*)

Return a copy of this VectorDatasetType class.

Parameters
****kwargs** (`Any`) – Additional keyword arguments passed into the constructor of the returned class.

Returns
`VectorDatasetType` – Copy of class

filter(*mask*, *copy=True*, ***kwargs*)

Filter `data` according to a boolean array `mask`.

Entries corresponding to `mask == True` are kept.

Parameters

- **mask** (`npt.NDArray[np.bool_]`) – Boolean array with compatible shape.
- **copy** (`bool`, *optional*) – Copy data on filter. Defaults to True. See numpy best practices for insight into whether copy is appropriate.
- ****kwargs** (`Any`) – Additional keyword arguments passed into the constructor of the returned class.

Returns
`VectorDatasetType` – Containing filtered data

Raises
`TypeError` – If `mask` is not a boolean array.

final_waypoints

`fl_attrs`

classmethod from_seq(*seq*, *broadcast_numeric=True*, *copy=True*, *attrs=None*)

Instantiate a *Fleet* instance from an iterable of `Flight`.

Changed in version 0.49.3: Empty flights are now filtered out before concatenation.

Parameters

- **seq** (`Iterable[Flight]`) – An iterable of `Flight` instances.
- **broadcast_numeric** (`bool`, *optional*) – If True, broadcast numeric attributes to data variables.
- **copy** (`bool`, *optional*) – If True, make copy of each flight instance in `seq`.
- **attrs** (`dict[str, Any] | None`, *optional*) – Global attribute to attach to instance.

Returns

Fleet – A *Fleet* instance made from concatenating the `Flight` instances in `seq`. The fuel type is taken from the first `Flight` in `seq`.

property max_distance_gap

Return maximum distance gap between waypoints along flight trajectory.

Distance is calculated based on WGS84 geodesic.

Returns

`float` – Maximum distance between waypoints, $[m]$

Raises

NotImplementedError – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.max_distance_gap
7391.27...
```

property n_flights

Return number of distinct flights.

Returns

`int` – Number of flights

resample_and_fill(**args*, ***kwargs*)

Resample and fill flight trajectory with geodesics and linear interpolation.

Waypoints are resampled according to the frequency `freq`. Values for `data` columns `longitude`, `latitude`, and `altitude` are interpolated.

Resampled waypoints will include all multiples of `freq` between the flight start and end time. For example, when resampling to a frequency of 1 minute, a flight that starts at 2020/1/1 00:00:59 and ends at 2020/1/1

00:01:01 will return a single waypoint at 2020/1/1 00:01:00, whereas a flight that starts at 2020/1/1 00:01:01 and ends at 2020/1/1 00:01:59 will return an empty flight.

Parameters

- **freq** (`str`, *optional*) – Resampling frequency, by default "1min"
- **fill_method** (`{"geodesic", "linear"}`, *optional*) – Choose between `"geodesic"` and `"linear"`, by default `"geodesic"`. In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.
- **geodesic_threshold** (`float`, *optional*) – Threshold for geodesic interpolation, $[m]$. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (`float | None`, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to `constants.nominal_rocd`.
- **drop** (`bool`, *optional*) – Drop any columns that are not resampled and filled. Defaults to `True`, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with `nan` or `None` values, depending on the data type.
- **keep_original_index** (`bool`, *optional*) – Keep the original index of the `Flight` in addition to the new resampled index. Defaults to `False`. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (`bool`) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns

Flight – Filled Flight

Raises

ValueError – Unknown `fill_method`

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
```

```
>>> df = pd.DataFrame()
>>> df['longitude'] = [0, 0, 50]
>>> df['latitude'] = 0
>>> df['altitude'] = 0
>>> df['time'] = [datetime(2020, 1, 1, h) for h in range(3)]
```

```
>>> fl = Flight(df)
>>> fl.dataframe
   longitude  latitude  altitude                time
           0       0.0       0.0       0.0 2020-01-01 00:00:00
           1       0.0       0.0       0.0 2020-01-01 01:00:00
           2      50.0       0.0       0.0 2020-01-01 02:00:00
```

```
>>> fl.resample_and_fill('10min').dataframe  # resample with 10 minute frequency
    longitude  latitude  altitude                time
```

(continues on next page)

(continued from previous page)

```
0    0.000000       0.0       0.0 2020-01-01 00:00:00
1    0.000000       0.0       0.0 2020-01-01 00:10:00
2    0.000000       0.0       0.0 2020-01-01 00:20:00
3    0.000000       0.0       0.0 2020-01-01 00:30:00
4    0.000000       0.0       0.0 2020-01-01 00:40:00
5    0.000000       0.0       0.0 2020-01-01 00:50:00
6    0.000000       0.0       0.0 2020-01-01 01:00:00
7    8.333333       0.0       0.0 2020-01-01 01:10:00
8   16.666667       0.0       0.0 2020-01-01 01:20:00
9   25.000000       0.0       0.0 2020-01-01 01:30:00
10  33.333333       0.0       0.0 2020-01-01 01:40:00
11  41.666667       0.0       0.0 2020-01-01 01:50:00
12  50.000000       0.0       0.0 2020-01-01 02:00:00
```

segment_angle()

Calculate sine and cosine for the angle between each segment and the longitudinal axis.

This is different from the usual navigational angle between two points known as *bearing*.

Bearing in 3D spherical coordinates is referred to as *azimuth*.

```
        (lon_2, lat_2)  X
                       /|
                      / |
                     /  |
                    /   |
                   /    |
                  /     |
                 /      |
(lon_1, lat_1)  X -------> longitude (x-axis)
```

Returns

npt.NDArray[np.float64], npt.NDArray[np.float64] – Returns sin(a), cos(a), where a is the angle between the segment and the longitudinal axis. The final values are of both arrays are np.nan.

See also:

geo.segment_angle(), units.heading_to_longitudinal_angle(), *segment_azimuth()*, geo.forward_azimuth()

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> sin, cos = fl.segment_angle()
```

(continues on next page)

(continued from previous page)

```
>>> sin
array([0.70716063, 0.70737598, 0.44819424, 0.31820671,        nan])
```

```
>>> cos
array([0.70705293, 0.70683748, 0.8939362 , 0.94802136,        nan])
```

segment_azimuth()

Calculate (forward) azimuth at each waypoint.

Method calls *pyproj.Geod.inv*, which is slow. See *geo.forward_azimuth* for an outline of a faster implementation.

Changed in version 0.33.7: The dtype of the output now matches the dtype of `self["longitude"]`.

Returns
`npt.NDArray[np.float64]` – Array of azimuths.

See also:

segment_angle(), `geo.forward_azimuth()`

segment_groundspeed(**args*, ***kwargs*)

Return groundspeed across segments.

Calculate by dividing the horizontal segment length by the difference in waypoint times.

Parameters

- **smooth** (`bool`, *optional*) – Smooth airspeed with Savitzky-Golay filter. Defaults to False.
- **window_length** (`int`, *optional*) – Passed directly to `scipy.signal.savgol_filter()`, by default 7.
- **polyorder** (`int`, *optional*) – Passed directly to `scipy.signal.savgol_filter()`, by default 1.

Returns
`npt.NDArray[np.float64]` – Groundspeed of the segment, $[ms^{-1}]$

segment_length()

Compute spherical distance between flight waypoints.

Helper function used in `length()` and `length_met()`. *np.nan* appended so the length of the output is the same as number of waypoints.

Returns
`npt.NDArray[np.float64]` – Array of distances in $[m]$ between waypoints

Raises
`NotImplementedError` – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> fl.segment_length()
array([157255.03346286, 157231.08336815, 248456.48781503, 351047.44358851,
                   nan])
```

See also:

segment_length()

segment_true_airspeed(*u_wind=0.0*, *v_wind=0.0*, *smooth=True*, *window_length=7*, *polyorder=1*)

Calculate the true airspeed $[m/s]$ from the ground speed and horizontal winds.

Because Flight.segment_true_airspeed uses a smoothing pattern, waypoints in `data` are not independent. Moreover, we expect the final waypoint of each flight to have a nan value associated to any segment property. Consequently, we need to define a custom method here to deal with these issues when applying this method on a fleet of flights.

See docstring for `Flight.segment_true_airspeed()`.

Raises

RuntimeError – Unexpected key *__u_wind* or *__v_wind* found in `data`.

sort(*by*)

Sort data by key(s).

This method always creates a copy of the data by calling `pandas.DataFrame.sort_values()`.

Parameters

by (`str | list[str]`) – Key or list of keys to sort by.

Returns

`VectorDatasetType` – Instance with sorted data.

to_flight_list(*copy=True*)

De-concatenate merged waypoints into a list of Flight instances.

Any global `attrs` are lost.

Parameters

copy (`bool`, *optional*) – If True, make copy of each flight instance in *seq*.

Returns

`list[Flight]` – List of Flights in the same order as was passed into the *Fleet* instance.

11.6.6 pycontrails.core.flight

Flight Data Handling.

Module Attributes

MAX_AIRPORT_ELEVATION	Max airport elevation, $[ft]$ See Daocheng_Yading_Airport
MIN_CRUISE_ALTITUDE	Min estimated cruise altitude, $[ft]$
SHORT_HAUL_DURATION	Short haul duration cutoff, $[s]$
MAX_ON_GROUND_SPEED	Set maximum speed compatible with "on_ground" indicator, $[mph]$ Thresholds assessed based on scatter plot (150 knots = 278 km/h)

Functions

filter_altitude(time, altitude_ft[, ...])	Filter noisy altitude on a single flight.
segment_duration(time[, dtype])	Calculate the time difference between waypoints.
segment_phase(rocd, altitude_ft, *[, ...])	Identify the phase of flight (climb, cruise, descent) for each segment.
segment_rocd(segment_duration, altitude_ft)	Calculate the rate of climb and descent (ROCD).

Classes

Flight([data, longitude, latitude, ...])	A single flight trajectory.
FlightPhase(value[, names, module, ...])	Flight phase enumeration.

class pycontrails.core.flight.**Flight**(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, *fuel=None*, *drop_duplicated_times=False*, ***attrs_kwargs*)

Bases: *GeoVectorDataset*

A single flight trajectory.

Expect latitude-longitude coordinates in WGS 84. Expect altitude in $[m]$. Expect pressure level (*level*) in $[hPa]$.

Use the attribute `attrs["crs"]` to specify coordinate reference system using PROJ or EPSG syntax.

Parameters

- **data** (`dict[str, np.ndarray] | pd.DataFrame | VectorDataDict | VectorDataset | None`) – Flight trajectory waypoints as data dictionary or `pandas.DataFrame`. Must include columns `time`, `latitude`, `longitude`, `altitude` or `level`. Keyword arguments for `time`, `latitude`, `longitude`, `altitude` or `level` will override `data` inputs. Expects `altitude` in meters and `time` as a DatetimeLike (or array that can processed with `pd.to_datetime()`). Additional waypoint-specific data can be included as additional keys/columns.
- **longitude** (`npt.ArrayLike`, *optional*) – Flight trajectory waypoint longitude. Defaults to None.

- **latitude** (`npt.ArrayLike`, *optional*) – Flight trajectory waypoint latitude. Defaults to None.
- **altitude** (`npt.ArrayLike`, *optional*) – Flight trajectory waypoint altitude, $[m]$. Defaults to None.
- **altitude_ft** (`npt.ArrayLike`, *optional*) – Flight trajectory waypoint altitude, $[ft]$.
- **level** (`npt.ArrayLike`, *optional*) – Flight trajectory waypoint pressure level, $[hPa]$. Defaults to None.
- **time** (`npt.ArrayLike`, *optional*) – Flight trajectory waypoint time. Defaults to None.
- **attrs** (`dict[str, Any]`, *optional*) – Additional flight properties as a dictionary. While different models may utilize Flight attributes differently, pycontrails applies the following conventions:
  - `flight_id`: An internal flight identifier. Used internally for `Fleet` interoperability.
  - `aircraft_type`: Aircraft type ICAO, e.g. `"A320"`.
  - `wingspan`: Aircraft wingspan, $[m]$.
  - `n_engine`: Number of aircraft engines.
  - `engine_uid`: Aircraft engine unique identifier. Used for emissions calculations with the ICAO Aircraft Emissions Databank (EDB).
  - `max_mach_number`: Maximum Mach number at cruise altitude. Used by some aircraft performance models to clip true airspeed.

  Numeric quantities that are constant over the entire flight trajectory should be included as attributes.
- **copy** (`bool`, *optional*) – Copy data on Flight creation. Defaults to True.
- **fuel** (*Fuel*, *optional*) – Fuel used in flight trajectory. Defaults to `JetA`.
- **drop_duplicated_times** (`bool`, *optional*) – Drop duplicate times in flight trajectory. Defaults to False.
- ****attrs_kwargs** (`Any`) – Additional flight properties passed as keyword arguments.

Raises

`KeyError` – Raises if `data` input does not contain at least `time`, `latitude`, `longitude`, (`altitude` or `level`).

Notes

The Traffic library has many helpful flight processing utilities.

See `traffic.core.Flight` for more information.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from pycontrails import Flight
```

```
>>> # Create `Flight` from a DataFrame.
>>> df = pd.DataFrame({
...     "longitude": np.linspace(20, 30, 500),
...     "latitude": np.linspace(40, 10, 500),
...     "altitude": 10500,
...     "time": pd.date_range('2021-01-01T10', '2021-01-01T15', periods=500),
... })
>>> fl = Flight(data=df, flight_id=123)  # specify a flight_id by keyword
>>> fl
Flight [4 keys x 500 length, 2 attributes]
Keys: longitude, latitude, altitude, time
Attributes:
time                [2021-01-01 10:00:00, 2021-01-01 15:00:00]
longitude           [20.0, 30.0]
latitude            [10.0, 40.0]
altitude            [10500.0, 10500.0]
flight_id           123
crs                 EPSG:4326
```

```
>>> # Create `Flight` from keywords
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl
Flight [4 keys x 200 length, 1 attributes]
    Keys: longitude, latitude, time, altitude
    Attributes:
    time                [2021-01-01 12:00:00, 2021-01-01 14:00:00]
    longitude           [20.0, 30.0]
    latitude            [30.0, 40.0]
    altitude            [11000.0, 11000.0]
    crs                 EPSG:4326
```

```
>>> # Access the underlying data as DataFrame
>>> fl.dataframe.head()
   longitude   latitude                          time  altitude
0  20.000000  40.000000 2021-01-01 12:00:00.000000000   11000.0
1  20.050251  39.949749 2021-01-01 12:00:36.180904522   11000.0
2  20.100503  39.899497 2021-01-01 12:01:12.361809045   11000.0
3  20.150754  39.849246 2021-01-01 12:01:48.542713567   11000.0
4  20.201005  39.798995 2021-01-01 12:02:24.723618090   11000.0
```

clean_and_resample(*freq='1min'*, *fill_method='geodesic'*, *geodesic_threshold=100000.0*, *nominal_rocd=12.7*, *kernel_size=17*, *cruise_threshold=120.0*, *force_filter=False*, *drop=True*, *keep_original_index=False*, *climb_descend_at_end=False*)

Resample and (possibly) filter a flight trajectory.

Waypoints are resampled according to the frequency `freq`. If the original flight data has a short sampling period, *filter_altitude* will also be called to clean the data. Large gaps in trajectories may be interpolated as step climbs through *_altitude_interpolation*.

Parameters

- **freq** (`str`, *optional*) – Resampling frequency, by default "1min"
- **fill_method** (`{"geodesic", "linear"}`, *optional*) – Choose between `"geodesic"` and `"linear"`, by default `"geodesic"`. In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.
- **geodesic_threshold** (`float`, *optional*) – Threshold for geodesic interpolation, $[m]$. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.
- **nominal_rocd** (`float`, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to `constants.nominal_rocd`.
- **kernel_size** (`int`, *optional*) – Passed directly to `scipy.signal.medfilt()`, by default 11. Passed also to `scipy.signal.medfilt()`
- **cruise_theshold** (`float`, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter
- **force_filter** (`bool`, *optional*) – If set to true, meth:*filter_altitude* will always be called. otherwise, it will only be called if the flight has a median sample period under 10 seconds
- **drop** (`bool`, *optional*) – Drop any columns that are not resampled and filled. Defaults to `True`, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with `nan` or `None` values, depending on the data type.
- **keep_original_index** (`bool`, *optional*) – Keep the original index of the *`Flight`* in addition to the new resampled index. Defaults to `False`. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (`bool`) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns

`Flight` – Filled Flight

copy(***kwargs*)

Return a copy of this VectorDatasetType class.

Parameters

****kwargs** (`Any`) – Additional keyword arguments passed into the constructor of the returned class.

Returns

`VectorDatasetType` – Copy of class

distance_to_coords(*distance*)

Convert distance along flight path to geodesic coordinates.

Will return a tuple containing *(lat, lon, index)*, where index indicates which flight segment contains the returned coordinate.

Parameters
distance (ArrayOrFloat) – Distance along flight path, [m]

Returns
(ArrayOrFloat, ArrayOrFloat, int | npt.NDArray[int]) – latitude, longitude, and segment index cooresponding to distance.

property duration

Determine flight duration.

Returns
pd.Timedelta – Difference between terminal and initial time

filter(*mask*, *copy=True*, ***kwargs*)

Filter data according to a boolean array mask.

Entries corresponding to mask == True are kept.

Parameters

- **mask** (npt.NDArray[np.bool_]) – Boolean array with compatible shape.
- **copy** (bool, *optional*) – Copy data on filter. Defaults to True. See numpy best practices for insight into whether copy is appropriate.
- ****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Containing filtered data

Raises
TypeError – If mask is not a boolean array.

filter_altitude(*kernel_size=17*, *cruise_threshold=120.0*)

Filter noisy altitude on a single flight.

Currently runs altitude through a median filter using scipy.signal.medfilt() with kernel_size, then a Savitzky-Golay filter to filter noise. The median filter is only applied during cruise segments that are longer than cruise_threshold.

Parameters

- **kernel_size** (int, *optional*) – Passed directly to scipy.signal.medfilt(), by default 11. Passed also to scipy.signal.medfilt()
- **cruise_theshold** (float, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter

Returns
Flight – Filtered Flight

Notes

Algorithm is derived from `traffic.core.flight.Flight.filter()`.

The traffic algorithm also computes thresholds on sliding windows and replaces unacceptable values with NaNs.

Errors may raised if the `kernel_size` is too large.

See also:

`traffic.core.flight.Flight.filter()`, `scipy.signal.medfilt()`

filter_by_first()

Keep first row of group of waypoints with identical coordinates.

Chaining this method with *resample_and_fill* often gives a cleaner trajectory when using noisy flight waypoints.

Returns

Flight – Filtered Flight instance

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
```

```
>>> df = pd.DataFrame()
>>> df['longitude'] = [0, 0, 50]
>>> df['latitude'] = 0
>>> df['altitude'] = 0
>>> df['time'] = [datetime(2020, 1, 1, h) for h in range(3)]
```

```
>>> fl = Flight(df)
```

```
>>> fl.filter_by_first().dataframe
   longitude  latitude  altitude                time
0        0.0       0.0       0.0 2020-01-01 00:00:00
1       50.0       0.0       0.0 2020-01-01 02:00:00
```

fuel

Fuel used in flight trajectory

property length

Return flight length based on WGS84 geodesic.

Returns

`float` – Total flight length, $[m]$

Raises

`NotImplementedError` – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.length
1436924.67...
```

length_met(*key*, *threshold=1.0*)

Calculate total horizontal distance where column `key` exceeds `threshold`.

Parameters

- **key** (`str`) – Column key in `data`
- **threshold** (`float`) – Consider trajectory waypoints whose associated `key` value exceeds `threshold`, by default 1.0

Returns

`float` – Total distance, $[m]$

Raises

- `KeyError` – `data` does not contain column `key`
- `NotImplementedError` – Raised when `attrs["crs"]` is not EPSG:4326

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
>>> import numpy as np
>>> from pycontrails.datalib.ecmwf import ERA5
>>> from pycontrails import Flight
```

```
>>> # Get met data
>>> times = (datetime(2022, 3, 1, 0),  datetime(2022, 3, 1, 3))
>>> variables = ["air_temperature", "specific_humidity"]
>>> levels = [300, 250, 200]
>>> era5 = ERA5(time=times, variables=variables, pressure_levels=levels)
>>> met = era5.open_metdataset()
```

```
>>> # Build flight
>>> df = pd.DataFrame()
>>> df['time'] = pd.date_range('2022-03-01T00', '2022-03-01T03', periods=11)
>>> df['longitude'] = np.linspace(-20, 20, 11)
>>> df['latitude'] = np.linspace(-20, 20, 11)
>>> df['altitude'] = np.linspace(9500, 10000, 11)
>>> fl = Flight(df).resample_and_fill('10s')
```

```
>>> # Intersect and attach
>>> fl["air_temperature"] = fl.intersect_met(met['air_temperature'])
>>> fl["air_temperature"]
array([235.94657007, 235.95766965, 235.96873412, ..., 234.59917962,
       234.60387402, 234.60845312])
```

```
>>> # Length (in meters) of waypoints whose temperature exceeds 236K
>>> fl.length_met("air_temperature", threshold=236)
4132178.159...
```

```
>>> # Proportion (with respect to distance) of waypoints whose temperature␣
↪exceeds 236K
>>> fl.proportion_met("air_temperature", threshold=236)
0.663552...
```

property max_distance_gap

Return maximum distance gap between waypoints along flight trajectory.

Distance is calculated based on WGS84 geodesic.

Returns
`float` – Maximum distance between waypoints, $[m]$

Raises
`NotImplementedError` – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.max_distance_gap
7391.27...
```

property max_time_gap

Return maximum time gap between waypoints along flight trajectory.

Returns
`pd.Timedelta` – Gap size

Examples

```
>>> import numpy as np
>>> fl = Flight(
...     longitude=np.linspace(20, 30, 200),
...     latitude=np.linspace(40, 30, 200),
...     altitude=11000 * np.ones(200),
...     time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=200),
... )
>>> fl.max_time_gap
Timedelta('0 days 00:00:36.180...')
```

plot(***kwargs*)

Plot flight trajectory longitude-latitude values.

Parameters

****kwargs** (Any) – Additional plot properties to passed to *pd.DataFrame.plot*

Returns

matplotlib.axes.Axes – Plot

proportion_met(*key*, *threshold=1.0*)

Calculate proportion of flight with certain meteorological constraint.

Parameters

- **key** (str) – Column key in data
- **threshold** (float) – Consider trajectory waypoints whose associated key value exceeds threshold, Defaults to 1.0

Returns

float – Ratio

resample_and_fill(*freq='1min'*, *fill_method='geodesic'*, *geodesic_threshold=100000.0*, *nominal_rocd=12.7*, *drop=True*, *keep_original_index=False*, *climb_descend_at_end=False*)

Resample and fill flight trajectory with geodesics and linear interpolation.

Waypoints are resampled according to the frequency freq. Values for data columns longitude, latitude, and altitude are interpolated.

Resampled waypoints will include all multiples of freq between the flight start and end time. For example, when resampling to a frequency of 1 minute, a flight that starts at 2020/1/1 00:00:59 and ends at 2020/1/1 00:01:01 will return a single waypoint at 2020/1/1 00:01:00, whereas a flight that starts at 2020/1/1 00:01:01 and ends at 2020/1/1 00:01:59 will return an empty flight.

Parameters

- **freq** (str, *optional*) – Resampling frequency, by default "1min"
- **fill_method** ({"geodesic", "linear"}, *optional*) – Choose between "geodesic" and "linear", by default "geodesic". In geodesic mode, large gaps between waypoints are filled with geodesic interpolation and small gaps are filled with linear interpolation. In linear mode, all gaps are filled with linear interpolation.
- **geodesic_threshold** (float, *optional*) – Threshold for geodesic interpolation, $[m]$. If the distance between consecutive waypoints is under this threshold, values are interpolated linearly.

- **nominal_rocd** (`float | None`, *optional*) – Nominal rate of climb / descent for aircraft type. Defaults to `constants.nominal_rocd`.
- **drop** (`bool`, *optional*) – Drop any columns that are not resampled and filled. Defaults to `True`, dropping all keys outside of "time", "latitude", "longitude" and "altitude". If set to False, the extra keys will be kept but filled with `nan` or `None` values, depending on the data type.
- **keep_original_index** (`bool`, *optional*) – Keep the original index of the *Flight* in addition to the new resampled index. Defaults to `False`. .. versionadded:: 0.45.2
- **climb_or_descend_at_end** (`bool`) – If true, the climb or descent will be placed at the end of each segment rather than the start. Default is false (climb or descent immediately).

Returns

Flight – Filled Flight

Raises

ValueError – Unknown `fill_method`

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
```

```
>>> df = pd.DataFrame()
>>> df['longitude'] = [0, 0, 50]
>>> df['latitude'] = 0
>>> df['altitude'] = 0
>>> df['time'] = [datetime(2020, 1, 1, h) for h in range(3)]
```

```
>>> fl = Flight(df)
>>> fl.dataframe
   longitude  latitude  altitude                time
           0       0.0       0.0       0.0 2020-01-01 00:00:00
           1       0.0       0.0       0.0 2020-01-01 01:00:00
           2      50.0       0.0       0.0 2020-01-01 02:00:00
```

```
>>> fl.resample_and_fill('10min').dataframe  # resample with 10 minute frequency
    longitude  latitude  altitude                time
0    0.000000       0.0       0.0 2020-01-01 00:00:00
1    0.000000       0.0       0.0 2020-01-01 00:10:00
2    0.000000       0.0       0.0 2020-01-01 00:20:00
3    0.000000       0.0       0.0 2020-01-01 00:30:00
4    0.000000       0.0       0.0 2020-01-01 00:40:00
5    0.000000       0.0       0.0 2020-01-01 00:50:00
6    0.000000       0.0       0.0 2020-01-01 01:00:00
7    8.333333       0.0       0.0 2020-01-01 01:10:00
8   16.666667       0.0       0.0 2020-01-01 01:20:00
9   25.000000       0.0       0.0 2020-01-01 01:30:00
10  33.333333       0.0       0.0 2020-01-01 01:40:00
11  41.666667       0.0       0.0 2020-01-01 01:50:00
12  50.000000       0.0       0.0 2020-01-01 02:00:00
```

segment_angle()

Calculate sine and cosine for the angle between each segment and the longitudinal axis.

This is different from the usual navigational angle between two points known as *bearing*.

Bearing in 3D spherical coordinates is referred to as *azimuth*.

```
        (lon_2, lat_2)  X
                       /|
                      / |
                     /  |
                    /   |
                   /    |
                  /     |
                 /      |
                /       |
(lon_1, lat_1)  X -------> longitude (x-axis)
```

Returns

npt.NDArray[np.float64], npt.NDArray[np.float64] – Returns sin(a), cos(a), where a is the angle between the segment and the longitudinal axis. The final values are of both arrays are np.nan.

See also:

geo.segment_angle(), units.heading_to_longitudinal_angle(), *segment_azimuth()*, geo.forward_azimuth()

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> sin, cos = fl.segment_angle()
>>> sin
array([0.70716063, 0.70737598, 0.44819424, 0.31820671,        nan])
```

```
>>> cos
array([0.70705293, 0.70683748, 0.8939362 , 0.94802136,        nan])
```

segment_azimuth()

Calculate (forward) azimuth at each waypoint.

Method calls *pyproj.Geod.inv*, which is slow. See *geo.forward_azimuth* for an outline of a faster implementation.

Changed in version 0.33.7: The dtype of the output now matches the dtype of self["longitude"].

Returns

npt.NDArray[np.float64] – Array of azimuths.

See also:

segment_angle(), geo.forward_azimuth()

segment_duration(*dtype=<class 'numpy.float32'>*)

Compute time elapsed between waypoints in seconds.

np.nan appended so the length of the output is the same as number of waypoints.

Parameters
dtype (np.dtype) – Numpy dtype for time difference. Defaults to np.float64

Returns
npt.NDArray[np.float64] – Time difference between waypoints, $[s]$. Returns an array with dtype specified by``dtype``

segment_groundspeed(*smooth=False*, *window_length=7*, *polyorder=1*)

Return groundspeed across segments.

Calculate by dividing the horizontal segment length by the difference in waypoint times.

Parameters

- **smooth** (bool, *optional*) – Smooth airspeed with Savitzky-Golay filter. Defaults to False.
- **window_length** (int, *optional*) – Passed directly to scipy.signal.savgol_filter(), by default 7.
- **polyorder** (int, *optional*) – Passed directly to scipy.signal.savgol_filter(), by default 1.

Returns
npt.NDArray[np.float64] – Groundspeed of the segment, $[ms^{-1}]$

segment_haversine()

Compute Haversine (great circle) distance between flight waypoints.

Helper function used in *resample_and_fill()*. *np.nan* appended so the length of the output is the same as number of waypoints.

To account for vertical displacements when computing segment lengths, use *segment_length()*.

Returns
npt.NDArray[np.float64] – Array of great circle distances in $[m]$ between waypoints

Raises
NotImplementedError – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> fl.segment_haversine()
array([157255.03346286, 157231.08336815, 248456.48781503, 351047.44358851,
                   nan])
```

See also:

segment_haversine(), *segment_length()*

segment_length()

Compute spherical distance between flight waypoints.

Helper function used in *length()* and *length_met()*. *np.nan* appended so the length of the output is the same as number of waypoints.

Returns

npt.NDArray[np.float64] – Array of distances in $[m]$ between waypoints

Raises

NotImplementedError – Raises when attr:*attrs["crs"]* is not EPSG:4326

Examples

```
>>> from pycontrails import Flight
>>> fl = Flight(
... longitude=np.array([1, 2, 3, 5, 8]),
... latitude=np.arange(5),
... altitude=np.full(shape=(5,), fill_value=11000),
... time=pd.date_range('2021-01-01T12', '2021-01-01T14', periods=5),
... )
>>> fl.segment_length()
array([157255.03346286, 157231.08336815, 248456.48781503, 351047.44358851,
                   nan])
```

See also:

segment_length()

segment_mach_number(*true_airspeed*, *air_temperature*)

Calculate the mach number of each segment.

Parameters

- **true_airspeed** (npt.NDArray[np.float64]) – True airspeed of the segment, $[m\ s^{-1}]$. See *segment_true_airspeed()*.
- **air_temperature** (npt.NDArray[np.float64]) – Average air temperature of each segment, $[K]$

Returns

npt.NDArray[np.float64] – Mach number of each segment

segment_phase(*threshold_rocd=250.0*, *min_cruise_altitude_ft=20000.0*)

Identify the phase of flight (climb, cruise, descent) for each segment.

Parameters

- **threshold_rocd** (float, *optional*) – ROCD threshold to identify climb and descent, $[ftmin^{-1}]$. Currently set to 250 ft/min.
- **min_cruise_altitude_ft** (float, *optional*) – Minimum altitude for cruise, $[ft]$ This is specific for each aircraft type, and can be approximated as 50% of the altitude ceiling. Defaults to 20000 ft.

Returns
npt.NDArray[np.uint8] – Array of values enumerating the flight phase. See flight.FlightPhase for enumeration.

See also:

FlightPhase, *segment_phase()*, *segment_rocd()*

segment_rocd()

Calculate the rate of climb and descent (ROCD).

Returns
npt.NDArray[np.float64] – Rate of climb and descent over segment, $[ftmin^{-1}]$

See also:

segment_rocd()

segment_true_airspeed(*u_wind=0.0*, *v_wind=0.0*, *smooth=True*, *window_length=7*, *polyorder=1*)

Calculate the true airspeed $[m/s]$ from the ground speed and horizontal winds.

The calculated ground speed will first be smoothed with a Savitzky-Golay filter if enabled.

Parameters

- **u_wind** (npt.NDArray[np.float64] | float) – U wind speed, $[m\ s^{-1}]$. Defaults to 0 for all waypoints.
- **v_wind** (npt.NDArray[np.float64] | float) – V wind speed, $[m\ s^{-1}]$. Defaults to 0 for all waypoints.
- **smooth** (bool, *optional*) – Smooth airspeed with Savitzky-Golay filter. Defaults to True.
- **window_length** (int, *optional*) – Passed directly to scipy.signal.savgol_filter(), by default 7.
- **polyorder** (int, *optional*) – Passed directly to scipy.signal.savgol_filter(), by default 1.

Returns
npt.NDArray[np.float64] – True wind speed of each segment, $[m\ s^{-1}]$

sort(*by*)

Sort data by key(s).

This method always creates a copy of the data by calling pandas.DataFrame.sort_values().

Parameters
by (str | list[str]) – Key or list of keys to sort by.

Returns
VectorDatasetType – Instance with sorted data.

property time_end

Last waypoint time.

Returns
pandas.Timestamp – Last waypoint time

property time_start

First waypoint time.

Returns
pandas.Timestamp – First waypoint time

`to_geojson_linestring()`

Return trajectory as geojson FeatureCollection containing single LineString.

Returns
`dict[str, Any]` – Python representation of geojson FeatureCollection

`to_geojson_multilinestring(key, split_antimeridian=True)`

Return trajectory as GeoJSON FeatureCollection of MultiLineStrings.

Flight `data` is grouped according to values of `key`. Each group gives rise to a Feature containing a MultiLineString geometry. LineStrings can be split over the antimeridian.

Parameters

- **key** (`str`) – Name of `data` column to group by
- **split_antimeridian** (`bool`, *optional*) – Split linestrings that cross the antimeridian. Defaults to True

Returns
`dict[str, Any]` – Python representation of GeoJSON FeatureCollection of MultiLinestring Features

Raises
`KeyError` – `data` does not contain column `key`

`to_traffic()`

Convert Flight instance to `traffic.core.Flight` instance.

See https://traffic-viz.github.io/traffic.core.flight.html#traffic.core.Flight

Returns
`traffic.core.Flight` – *traffic.core.Flight* instance

Raises
`ModuleNotFoundError` – *traffic* package not installed

`class pycontrails.core.flight.FlightPhase(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)`

Bases: `IntEnum`

Flight phase enumeration.

Use *`segment_phase()`* or *`Flight.segment_phase()`* to determine flight phase.

`CLIMB = 1`

Waypoints at which the flight is in a climb phase

`CRUISE = 2`

Waypoints at which the flight is in a cruise phase

`DESCENT = 3`

Waypoints at which the flight is in a descent phase

`LEVEL_FLIGHT = 4`

Waypoints at which the flight is not in a climb, cruise, or descent phase. In practice, this category is used for waypoints at which the ROCD resembles that of a cruise phase, but the altitude is below the minimum cruise altitude.

`NAN = 5`

Waypoints at which the ROCD is not defined.

pycontrails.core.flight.**MAX_AIRPORT_ELEVATION = 15000.0**

Max airport elevation, [ft] See Daocheng_Yading_Airport

pycontrails.core.flight.**MAX_ON_GROUND_SPEED = 150.0**

Set maximum speed compatible with "on_ground" indicator, [mph] Thresholds assessed based on scatter plot (150 knots = 278 km/h)

pycontrails.core.flight.**MIN_CRUISE_ALTITUDE = 20000.0**

Min estimated cruise altitude, [ft]

pycontrails.core.flight.**SHORT_HAUL_DURATION = 3600.0**

Short haul duration cutoff, [s]

pycontrails.core.flight.**filter_altitude**(*time*, *altitude_ft*, *kernel_size=17*, *cruise_threshold=120*)

Filter noisy altitude on a single flight.

Currently runs altitude through a median filter using `scipy.signal.medfilt()` with `kernel_size`, then a Savitzky-Golay filter to filter noise. The median filter is only applied during cruise segments that are longer than `cruise_threshold`.

Parameters

- **time** (`npt.NDArray[np.datetime64]`) – Waypoint time in `np.datetime64` format.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude signal in feet
- **kernel_size** (`int`, *optional*) – Passed directly to `scipy.signal.medfilt()`, by default 11. Passed also to `scipy.signal.medfilt()`
- **cruise_theshold** (`int`, *optional*) – Minimal length of time, in seconds, for a flight to be in cruise to apply median filter

Returns

`npt.NDArray[np.float64]` – Filtered altitude

Notes

Algorithm is derived from `traffic.core.flight.Flight.filter()`.

The traffic algorithm also computes thresholds on sliding windows and replaces unacceptable values with NaNs.

Errors may raised if the `kernel_size` is too large.

See also:

`traffic.core.flight.Flight.filter()`, `scipy.signal.medfilt()`

pycontrails.core.flight.**segment_duration**(*time*, *dtype=<class 'numpy.float32'>*)

Calculate the time difference between waypoints.

`np.nan` appended so the length of the output is the same as number of waypoints.

Parameters

- **time** (`npt.NDArray[np.datetime64]`) – Waypoint time in `np.datetime64` format.
- **dtype** (`np.dtype`) – Numpy dtype for time difference. Defaults to `np.float64`

Returns

`npt.NDArray[np.float64]` – Time difference between waypoints, [s]. This returns an array with dtype specified by``dtype``.

pycontrails.core.flight.**segment_phase**(*rocd*, *altitude_ft*, *, *threshold_rocd=250.0*, *min_cruise_altitude_ft=20000.0*)

Identify the phase of flight (climb, cruise, descent) for each segment.

Parameters

- **rocd** (pt.NDArray[np.float64]) – Rate of climb and descent across segment, $[ftmin^{-1}]$. See output from *segment_rocd()*.
- **altitude_ft** (npt.NDArray[np.float64]) – Altitude, $[ft]$
- **threshold_rocd** (float, *optional*) – ROCD threshold to identify climb and descent, $[ftmin^{-1}]$. Defaults to 250 ft/min.
- **min_cruise_altitude_ft** (float, *optional*) – Minimum threshold altitude for cruise, $[ft]$ This is specific for each aircraft type, and can be approximated as 50% of the altitude ceiling. Defaults to *MIN_CRUISE_ALTITUDE*.

Returns

npt.NDArray[np.uint8] – Array of values enumerating the flight phase. See flight.FlightPhase for enumeration.

Notes

Flight data derived from ADS-B and radar sources could contain noise leading to small changes in altitude and ROCD. Hence, an arbitrary threshold_rocd is specified to identify the different phases of flight.

The flight phase "level-flight" is when an aircraft is holding at lower altitudes. The cruise phase of flight only occurs above a certain threshold altitude.

See also:

FlightPhase, *segment_rocd()*

pycontrails.core.flight.**segment_rocd**(*segment_duration*, *altitude_ft*)

Calculate the rate of climb and descent (ROCD).

Parameters

- **segment_duration** (npt.NDArray[np.float64]) – Time difference between waypoints, $[s]$. Expected to have numeric *dtype*, not *"timedelta64"*. See output from *segment_duration()*.
- **altitude_ft** (npt.NDArray[np.float64]) – Altitude of each waypoint, $[ft]$

Returns

npt.NDArray[np.float64] – Rate of climb and descent over segment, $[ftmin^{-1}]$

See also:

segment_duration()

11.6.7 pycontrails.core.fuel

Fuel data support.

Classes

Fuel(fuel_name, q_fuel, hydrogen_content, ...)	Base class for the physical parameters of the fuel.
HydrogenFuel([fuel_name, q_fuel, ...])	Hydrogen Fuel.
JetA([fuel_name, q_fuel, hydrogen_content, ...])	Jet A-1 Fuel.
SAFBlend(pct_blend)	Jet A-1 / Sustainable Aviation Fuel Blend.

class pycontrails.core.fuel.**Fuel**(*fuel_name*, *q_fuel*, *hydrogen_content*, *ei_co2*, *ei_h2o*, *ei_so2*, *ei_sulphates*, *ei_oc*)

Bases: object

Base class for the physical parameters of the fuel.

ei_co2

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

ei_so2

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$

ei_sulphates

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$

fuel_name

Fuel Name

hydrogen_content

Percentage of hydrogen mass content in the fuel

q_fuel

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

class pycontrails.core.fuel.**HydrogenFuel**(*fuel_name='Hydrogen'*, *q_fuel=122800000.0*, *hydrogen_content=nan*, *ei_co2=0.0*, *ei_h2o=9.21*, *ei_so2=0.0*, *ei_sulphates=0.0*, *ei_oc=0.0*)

Bases: *Fuel*

Hydrogen Fuel.

References

- [Khan *et al.*, 2022]

ei_co2 = 0.0

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o = 9.21

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc = 0.0

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

ei_so2 = 0.0

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$

ei_sulphates = 0.0

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$

fuel_name = 'Hydrogen'

Fuel Name

hydrogen_content = nan

Percentage of hydrogen mass content in the fuel

q_fuel = 122800000.0

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

class pycontrails.core.fuel.**JetA**(*fuel_name='Jet A-1'*, *q_fuel=43130000.0*, *hydrogen_content=13.8*, *ei_co2=3.159*, *ei_h2o=1.23*, *ei_so2=0.0012*, *ei_sulphates=2.4489795918367345e-05*, *ei_oc=1.9999999999999998e-05*)

Bases: *Fuel*

Jet A-1 Fuel.

References

- [Celikel and Jelinek, 2001]
- [Lee *et al.*, 2021]
- [Stettler *et al.*, 2011]
- [Wilkerson *et al.*, 2010]

ei_co2 = 3.159

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o = 1.23

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc = 1.9999999999999998e-05

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$ - High uncertainty - Wilkerson et al. (2010): EI_OC = 15 mg/kg-fuel - Stettler et al. (2011): EI_OC = 20 [1, 40] mg/kg-fuel

ei_so2 = 0.0012

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$ - The EI SO2 is proportional to the fuel sulphur content - Celikel (2001): EI_SO2 = 0.84 g/kg-fuel for 450 ppm fuel - Lee et al. (2021): EI_SO2 = 1.2 g/kg-fuel for 600 ppm fuel

ei_sulphates = 2.4489795918367345e-05

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$ - The SOx-S is partitioned into 98% SO2-S gas and 2% S(VI)-S particle - References: Wilkerson et al. (2010) & Stettler et al. (2011)

fuel_name = 'Jet A-1'

Fuel Name

hydrogen_content = 13.8

Percentage of hydrogen mass content in the fuel

q_fuel = 43130000.0

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

class pycontrails.core.fuel.**SAFBlend**(*pct_blend*)

Bases: *Fuel*

Jet A-1 / Sustainable Aviation Fuel Blend.

SAF only changes the CO2 lifecycle emissions, not the CO2 emissions emitted at the aircraft exhaust. We assume that the EI OC stays the same as Jet A-1 fuel due to lack of data.

Parameters

pct_blend (float) – Sustainable aviation fuel percentage blend ratio by volume, %. Expected to be in the interval [0, 100].

References

- [Teoh *et al.*, 2022]
- [Schripp *et al.*, 2022]

ei_co2

CO2 emissions index for fuel, $[kg_{CO_2}\ kg_{fuel}^{-1}]$

ei_h2o

Water vapour emissions index for fuel, $[kg_{H_2O}\ kg_{fuel}^{-1}]$

ei_oc

Organic carbon emissions index for fuel, $[kg_{OC}\ kg_{fuel}^{-1}]$

ei_so2

Sulphur oxide, SO2-S gas, emissions index for fuel, $[kg_{SO_2}\ kg_{fuel}^{-1}]$

ei_sulphates

Sulphates, S(VI)-S particle, emissions index for fuel, $[kg_S\ kg_{fuel}^{-1}]$

fuel_name

Fuel Name

hydrogen_content

Percentage of hydrogen mass content in the fuel

q_fuel

Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$

11.6.8 pycontrails.core.interpolation

Interpolation utilities.

Functions

interp(longitude, latitude, level, time, da, ...)	Interpolate over a grid with `localize` option.

Classes

EmissionsProfileInterpolator(xp, fp[, ...])	Support for interpolating a profile on a linear or logarithmic scale.
PycontrailsRegularGridInterpolator(points, ...)	Support for performant interpolation over a regular grid.
RGIArtifacts(xi_indices, norm_distances, ...)	An interface to intermediate RGI interpolation artifacts.

class pycontrails.core.interpolation.**EmissionsProfileInterpolator**(*xp*, *fp*, *drop_duplicates=True*)

Bases: `object`

Support for interpolating a profile on a linear or logarithmic scale.

This class simply wraps `numpy.interp()` with fixed values for the `xp` and `fp` arguments. Unlike `xarray.DataArray` interpolation, the *numpy.interp* automatically clips values outside the range of the `xp` array.

Parameters

- **xp** (`npt.NDarray[np.float64]`) – Array of x-values. These must be strictly increasing and free from any nan values. Passed to `numpy.interp()`.
- **fp** (`npt.NDarray[np.float64]`) – Array of y-values. Passed to `numpy.interp()`.
- **drop_duplicates** (`bool`, *optional*) – Whether to drop duplicate values in `xp`. Defaults to `True`.

Examples

```
>>> xp = np.array([3, 7, 10, 30], dtype=float)
>>> fp = np.array([0.1, 0.2, 0.3, 0.4], dtype=float)
>>> epi = EmissionsProfileInterpolator(xp, fp)
>>> # Interpolate a single value
>>> epi.interp(5)
0.150000...
```

```
>>> # Interpolate a single value on a logarithmic scale
>>> epi.log_interp(5)
1.105171...
```

```
>>> # Demonstrate speed up compared with xarray.DataArray interpolation
>>> import time, xarray as xr
>>> da = xr.DataArray(fp, dims=["x"], coords={"x": xp})
```

```
>>> inputs = [np.random.uniform(0, 31, size=200) for _ in range(1000)]
>>> t0 = time.perf_counter()
>>> xr_out = [da.interp(x=x.clip(3, 30)).values for x in inputs]
>>> t1 = time.perf_counter()
>>> np_out = [epi.interp(x) for x in inputs]
>>> t2 = time.perf_counter()
>>> assert np.allclose(xr_out, np_out)
```

```
>>> # We see a 100 fold speed up (more like 500x faster, but we don't
>>> # want the test to fail!)
>>> assert t2 - t1 < (t1 - t0) / 100
```

interp(*x*)

Interpolate x against xp and fp.

Parameters
x (`npt.NDArray[np.float64]`) – Array of x-values to interpolate.

Returns
`npt.NDArray[np.float64]` – Array of interpolated y-values arising from the x-values. The `dtype` of the output array is the same as the `dtype` of `x`.

log_interp(*x*)

Interpolate x against xp and fp on a logarithmic scale.

This method composes the following three functions.

1. `numpy.log()`
2. *interp()*
3. `numpy.exp()`

Parameters
x (`npt.NDArray[np.float64]`) – Array of x-values to interpolate.

Returns
`npt.NDArray[np.float64]` – Array of interpolated y-values arising from the x-values.

class pycontrails.core.interpolation.**PycontrailsRegularGridInterpolator**(*points*, *values*, *method*, *bounds_error*, *fill_value*)

Bases: `RegularGridInterpolator`

Support for performant interpolation over a regular grid.

This class is a thin wrapper around the `scipy.interpolate.RegularGridInterpolator` in order to make typical `pycontrails` use-cases more efficient.

#. Avoid `RegularGridInterpolator` constructor validation. In *interp()*, parameters are carefully crafted to fit into the intended form, thereby making validation unnecessary. #. Override the `_evaluate_linear()` method with a faster implementation. See the `_evaluate_linear()` docstring for more information.

This class should not be used directly. Instead, use the *interp()* function.

Changed in version 0.40.0: The `_evaluate_linear()` method now uses a Cython implementation. The dtype of the output is now consistent with the dtype of the underlying `values`

Parameters

- **points** (`tuple[npt.NDArray[np.float64], ]`) – Coordinates of the grid points.
- **values** (`npt.NDArray[np.float64]`) – Grid values. The shape of this array must be compatible with the coordinates. An error is raised if the dtype is not `np.float32` or `np.float64`.
- **method** (`str`) – Passed into `scipy.interpolate.RegularGridInterpolator`
- **bounds_error** (`bool`) – Passed into `scipy.interpolate.RegularGridInterpolator`
- **fill_value** (`float | np.float64 | None`) – Passed into `scipy.interpolate.RegularGridInterpolator`

class pycontrails.core.interpolation.**RGIArtifacts**(*xi_indices*, *norm_distances*, *out_of_bounds*)

Bases: `object`

An interface to intermediate RGI interpolation artifacts.

norm_distances

out_of_bounds

xi_indices

pycontrails.core.interpolation.**interp**(*longitude*, *latitude*, *level*, *time*, *da*, *method*, *bounds_error*, *fill_value*, *localize*, ***, *indices=None*, *return_indices=False*)

Interpolate over a grid with `localize` option.

Changed in version 0.25.6: Utilize scipy 1.9 upgrades to remove singleton dimensions.

Changed in version 0.26.0: Include `indices` and `return_indices` experimental parameters. Currently, nan values in `longitude`, `latitude`, `level`, or `time` are always propagated through to the output, regardless of `bounds_error`. In other words, a ValueError for an out of bounds coordinate is only raised if a non-nan value is out of bounds.

Changed in version 0.40.0: When `return_indices` is True, an instance of *RGIArtifacts* is used to store the indices artifacts.

Parameters

- **longitude, latitude, level, time** (`numpy.ndarray`) – Coordinates of points to be interpolated. These parameters have the same meaning as `x` in analogy with `numpy.interp()`. All four of these arrays must be 1 dimensional of the same size.
- **da** (`xarray.DataArray`) – Gridded data interpolated over. Must adhere to `MetDataArray` conventions. In particular, the dimensions of `da` must be `longitude`, `latitude`, `level`, and `time`. The three spatial dimensions must be monotonically increasing with `float64` dtype. The `time` dimension must be monotonically increasing with `datetime64` dtype. Assumed to be cheap to load into memory (`xr.DataArray.values` is used without hesitation).
- **method** (`str`) – Passed into `scipy.interpolate.RegularGridInterpolator`.
- **bounds_error** (`bool`) – Passed into `scipy.interpolate.RegularGridInterpolator`.
- **fill_value** (`float | np.float64 | None`) – Passed into `scipy.interpolate.RegularGridInterpolator`.
- **localize** (`bool`) – If True, clip `da` to the smallest bounding box that contains all of `coords`.

- **indices** (`tuple | None`, *optional*) – Experimental. Provide intermediate artifacts computed by :meth:`scipy.interpolate.RegularGridInterpolator._find_indices` to avoid redundant computation. If known and provided, this can speed up interpolation by avoiding an unnecessary call to ``_find_indices. By default, None. Must be used precisely.
- **return_indices** (`bool`, *optional*) – If True, return output of `scipy.interpolate.RegularGridInterpolator._find_indices()` in addition to interpolated values.

Returns

`npt.NDArray[np.float64] | tuple[npt.NDArray[np.float64], RGIArtifacts]` – Interpolated values with same size as `longitude`. If `return_indices` is True, return intermediate indices artifacts as well.

See also:

- meth:*MetDataArray.interpolate*
- func:*scipy.interpolate.interpn*
- class:*scipy.interpolate.RegularGridInterpolator*

11.6.9 pycontrails.core.met

Meteorology data models.

Functions

downselect(data, bbox)	Downselect `xr.Dataset` or `xr.DataArray` with spatial bounding box.
originates_from_ecmwf(met)	Check if data appears to have originated from an ECMWF source.
shift_longitude(data[, bound])	Shift longitude values from any input domain to [bound, 360 + bound) domain.
standardize_variables(ds, variables)	Rename all variables in dataset from short name to standard name.

Classes

MetBase()	Abstract class for building Meteorology Data handling classes.
MetDataArray(data[, cachestore, ...])	Meteorological DataArray of single variable.
MetDataset(data[, cachestore, ...])	Meteorological dataset with multiple variables.

class pycontrails.core.met.**MetBase**

Bases: `ABC`, `Generic`[`XArrayType`]

Abstract class for building Meteorology Data handling classes.

All support here should be generic to work on xr.DataArray and xr.Dataset.

property attrs

Pass through to `self.data.attrs`.

abstract broadcast_coords(*name*)

Broadcast coordinates along other dimensions.

Parameters
name (`str`) – Coordinate/dimension name to broadcast. Can be a dimension or non-dimension coordinates.

Returns
`xarray.DataArray` – DataArray of the coordinate broadcasted along all other dimensions. The DataArray will have the same shape as the gridded data.

cachestore

Cache datastore to use for `save()` or `load()`

property coords

Get coordinates of underlying *data* coordinates.

Only return non-dimension coordinates.

See: http://xarray.pydata.org/en/stable/user-guide/data-structures.html#coordinates

Returns
`dict[str, np.ndarray]` – Dictionary of coordinates

data

DataArray or Dataset

dim_order = ['longitude', 'latitude', 'level', 'time']

Default dimension order for DataArray or Dataset (x, y, z, t)

abstract downselect(*bbox*)

Downselect met data within spatial bounding box.

Parameters
bbox (`list[float]`) – List of coordinates defining a spatial bounding box in WGS84 coordinates. For 2D queries, list is [west, south, east, north]. For 3D queries, list is [west, south, min-level, east, north, max-level] with level defined in $[hPa]$.

Returns
MetBase – Return downselected data

downselect_met(*met*, ***, *longitude_buffer=(0.0, 0.0)*, *latitude_buffer=(0.0, 0.0)*, *level_buffer=(0.0, 0.0)*, *time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *copy=True*)

Downselect `met` to encompass a spatiotemporal region of the data.

> **Warning:** This method is analogous to `GeoVectorDataset.downselect_met()`. It does not change the instance data, but instead operates on the `met` input. This method is different from *downselect()* which operates on the instance data.

Parameters

- **met** (`MetDataset | MetDataArray`) – MetDataset or MetDataArray to downselect.

- **longitude_buffer** (`tuple[float, float]`, *optional*) – Extend longitude domain past by `longitude_buffer[0]` on the low side and `longitude_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` degrees.
- **latitude_buffer** (`tuple[float, float]`, *optional*) – Extend latitude domain past by `latitude_buffer[0]` on the low side and `latitude_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` degrees.
- **level_buffer** (`tuple[float, float]`, *optional*) – Extend level domain past by `level_buffer[0]` on the low side and `level_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` $[hPa]$.
- **time_buffer** (`tuple[np.timedelta64, np.timedelta64]`, *optional*) – Extend time domain past by `time_buffer[0]` on the low side and `time_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(np.timedelta64(0, "h"), np.timedelta64(0, "h"))`.
- **copy** (`bool`) – If returned object is a copy or view of the original. True by default.

Returns

`MetDataset | MetDataArray` – Copy of downselected MetDataset or MetDataArray.

property hash

Generate a unique hash for this met instance.

Note this is not as robust as it could be since *repr* cuts off.

Returns

`str` – Unique hash for met instance (sha1)

property indexes

Low level access to underlying *data* indexes.

This method is typically is faster for accessing coordinate indexes.

New in version 0.25.2.

Returns

`dict[Hashable, pd.Index]` – Dictionary of indexes.

Examples

```
>>> from pycontrails.datalib.ecmwf import ERA5
>>> times = (datetime(2022, 3, 1, 12),  datetime(2022, 3, 1, 13))
>>> variables = "air_temperature", "specific_humidity"
>>> levels = [200, 300]
>>> era5 = ERA5(times, variables, levels)
>>> mds = era5.open_metdataset()
>>> mds.indexes["level"].to_numpy()
array([200., 300.])
```

```
>>> mda = mds["air_temperature"]
>>> mda.indexes["level"].to_numpy()
array([200., 300.])
```

property is_single_level

Check if instance contains "single level" or "surface level" data.

This method checks if `level` dimension contains a single value equal to -1, the pycontrails convention for surface only data.

Returns
`bool` – If instance contains single level data.

property is_wrapped

Check if the longitude dimension covers the closed interval `[-180, 180]`.

Assumes the longitude dimension is sorted (this is established by the *MetDataset* or *MetDataArray* constructor).

Returns
`bool` – True if longitude coordinates cover `[-180, 180]`

See also:

pycontrails.physics.geo.advect_longitude()

property is_zarr

Check if underlying *data* is sourced from a Zarr group.

Implementation is very brittle, and may break as external libraries change.

Some `dask` intermediate artifact is cached when this is called. Typically, subsequent calls to this method are much faster than the initial call.

New in version 0.26.0.

Returns
`bool` – If `data` is based on a Zarr group.

abstract property shape

Return the shape of the dimensions.

Returns
`tuple[int, int, int, int]` – Shape of underlying data

abstract property size

Return the size of (each) array in underlying *data*.

Returns
`int` – Total number of grid points in underlying data

property variables

See *indexes*.

class pycontrails.core.met.**MetDataArray**(*data*, *cachestore=None*, *wrap_longitude=False*, *copy=True*, *validate=True*, *name=None*, ***kwargs*)

Bases: *MetBase*

Meteorological DataArray of single variable.

Wrapper around xr.DataArray to enforce certain variables and dimensions for internal usage.

Parameters

- **data** (*ArrayLike*) – xr.DataArray or other array-like data source. When array-like input is provided, input `**kwargs` passed directly to xr.DataArray constructor.

- **cachestore** (CacheStore, *optional*) – Cache datastore for staging intermediates with *save()*. Defaults to DiskCacheStore.
- **wrap_longitude** (bool, *optional*) – Wrap data along the longitude dimension. If True, duplicate and shift longitude values (ie, -180 -> 180) to ensure that the longitude dimension covers the entire interval `[-180, 180]`. Defaults to False.
- **copy** (bool, *optional*) – Copy *data* parameter on construction, by default *True*. If *data* is lazy-loaded via *dask*, this parameter has no effect. If *data* is already loaded into memory, a copy of the data (rather than a view) may be created if *True*.
- **validate** (bool, *optional*) – Confirm that the parameter *data* has correct specification. This automatically handled in the case that *copy=True*. Validation only introduces a very small overhead. This parameter should only be set to *False* if working with data derived from an existing MetDataset or :class`MetDataArray`. By default *True*.
- **name** (Hashable, *optional*) – Name of the data variable. If not specified, the name will be set to "met".
- ****kwargs** – To be removed in future versions. Passed directly to xr.DataArray constructor.

Examples

```
>>> import numpy as np
>>> import xarray as xr
>>> rng = np.random.default_rng(seed=456)
```

```
>>> # Cook up random xarray object
>>> coords = {
...     "longitude": np.arange(-20, 20),
...     "latitude": np.arange(-30, 30),
...     "level": [220, 240, 260, 280],
...     "time": [np.datetime64("2021-08-01T12", "ns"), np.datetime64("2021-08-01T16
↪", "ns")]
...     }
>>> da = xr.DataArray(rng.random((40, 60, 4, 2)), dims=coords.keys(), coords=coords)
```

```
>>> # Cast to `MetDataArray` in order to interpolate
>>> from pycontrails import MetDataArray
>>> mda = MetDataArray(da)
>>> mda.interpolate(-11.4, 5.7, 234, np.datetime64("2021-08-01T13"))
array([0.52358215])
```

```
>>> mda.interpolate(-11.4, 5.7, 234, np.datetime64("2021-08-01T13"), method='nearest
↪')
array([0.4188465])
```

```
>>> da.sel(longitude=-11, latitude=6, level=240, time=np.datetime64("2021-08-01T12
↪")).item()
0.41884649899766946
```

property binary

Determine if all data is a binary value (0, 1).

Returns
bool – True if all data values are binary value (0, 1)

broadcast_coords(*name*)

Broadcast coordinates along other dimensions.

Parameters
name (str) – Coordinate/dimension name to broadcast. Can be a dimension or non-dimension coordinates.

Returns
xarray.DataArray – DataArray of the coordinate broadcasted along all other dimensions. The DataArray will have the same shape as the gridded data.

cachestore

Cache datastore to use for *save()* or *load()*

copy()

Create a copy of the current class.

Returns
MetDataArray – MetDataArray copy

data

DataArray or Dataset

downselect(*bbox*)

Downselect met data within spatial bounding box.

Parameters
bbox (list[float]) – List of coordinates defining a spatial bounding box in WGS84 coordinates. For 2D queries, list is [west, south, east, north]. For 3D queries, list is [west, south, min-level, east, north, max-level] with level defined in [hPa].

Returns
MetBase – Return downselected data

find_edges()

Find edges of regions.

Returns
MetDataArray – MetDataArray with a binary field, 1 on the edge of the regions, 0 outside and inside the regions.

Raises
NotImplementedError – If the instance is not binary.

property in_memory

Check if underlying *data* is loaded into memory.

This method uses protected attributes of underlying *xarray* objects, and may be subject to deprecation.

Changed in version 0.26.0: Rename from `is_loaded` to `in_memory`.

Returns
bool – If underlying data exists as an *np.ndarray* in memory.

interpolate(*longitude*, *latitude*, *level*, *time*, *, *method='linear'*, *bounds_error=False*, *fill_value=nan*, *localize=False*, *indices=None*, *return_indices=False*)

Interpolate values over underlying DataArray.

Zero dimensional coordinates are reshaped to 1D arrays.

Method automatically loads underlying *data* into memory.

If `method == "nearest"`, the out array will have the same `dtype` as the underlying *data*.

If `method == "linear"`, the out array will be promoted to the most precise `dtype` of:

- underlying *data*
- `data.longitude`
- `data.latitude`
- `data.level`
- `longitude`
- `latitude`

New in version 0.24: This method can now handle singleton dimensions with `method == "linear"`. Previously these degenerate dimensions caused nan values to be returned.

Parameters

- **longitude** (`float | npt.NDArray[np.float64]`) – Longitude values to interpolate. Assumed to be 0 or 1 dimensional.
- **latitude** (`float | npt.NDArray[np.float64]`) – Latitude values to interpolate. Assumed to be 0 or 1 dimensional.
- **level** (`float | npt.NDArray[np.float64]`) – Level values to interpolate. Assumed to be 0 or 1 dimensional.
- **time** (`np.datetime64 | npt.NDArray[np.datetime64]`) – Time values to interpolate. Assumed to be 0 or 1 dimensional.
- **method** (`str`, *optional*) – Additional keyword arguments to pass to `scipy.interpolate.RegularGridInterpolator`. Defaults to "linear".
- **bounds_error** (`bool`, *optional*) – Additional keyword arguments to pass to `scipy.interpolate.RegularGridInterpolator`. Defaults to `False`.
- **fill_value** (`float | np.float64`, *optional*) – Additional keyword arguments to pass to `scipy.interpolate.RegularGridInterpolator`. Set to None to extrapolate outside the boundary when `method` is `nearest`. Defaults to `np.nan`.
- **localize** (`bool`, *optional*) – Experimental. If True, downselect gridded data to smallest bounding box containing all points. By default False.
- **indices** (`tuple | None`, *optional*) – Experimental. See `interpolation.interp()`. None by default.
- **return_indices** (`bool`, *optional*) – Experimental. See `interpolation.interp()`. False by default.

Returns

`numpy.ndarray` – Interpolated values

See also:

`GeoVectorDataset.intersect_met()`

Examples

```
>>> from datetime import datetime
>>> import numpy as np
>>> import pandas as pd
>>> from pycontrails.datalib.ecmwf import ERA5
```

```
>>> times = (datetime(2022, 3, 1, 12),  datetime(2022, 3, 1, 15))
>>> variables = "air_temperature"
>>> levels = [200, 250, 300]
>>> era5 = ERA5(times, variables, levels)
>>> met = era5.open_metdataset()
>>> mda = met["air_temperature"]
```

```
>>> # Interpolation at a grid point agrees with value
>>> mda.interpolate(1, 2, 300, np.datetime64('2022-03-01T14:00'))
array([241.91972984])
```

```
>>> da = mda.data
>>> da.sel(longitude=1, latitude=2, level=300, time=np.datetime64('2022-03-01T14
↪')).item()
241.9197298421629
```

```
>>> # Interpolation off grid
>>> mda.interpolate(1.1, 2.1, 290, np.datetime64('2022-03-01 13:10'))
array([239.83793798])
```

```
>>> # Interpolate along path
>>> longitude = np.linspace(1, 2, 10)
>>> latitude = np.linspace(2, 3, 10)
>>> level = np.linspace(200, 300, 10)
>>> time = pd.date_range("2022-03-01T14", periods=10, freq="5min")
>>> mda.interpolate(longitude, latitude, level, time)
array([220.44347694, 223.08900738, 225.74338924, 228.41642088,
       231.10858599, 233.54857391, 235.71504913, 237.86478872,
       239.99274623, 242.10792167])
```

classmethod load(*hash*, *cachestore=None*, *chunks=None*)

Load saved intermediate from *cachestore*.

Parameters

- **hash** (str) – Saved hash to load.
- **cachestore** (CacheStore, *optional*) – Cache datastore to use for sourcing files. Defaults to DiskCacheStore.
- **chunks** (dict[str, int], *optional*) – Chunks kwarg passed to xarray.open_mfdataset() when opening files.

Returns

MetDataArray – New MetDataArray with loaded data.

property name

Return the DataArray name.

Returns
Hashable – DataArray name

property proportion

Compute proportion of points with value 1.

Returns
float – Proportion of points with value 1

Raises
NotImplementedError – If instance does not contain binary data.

save(***kwargs*)

Save intermediate to *cachestore* as netcdf.

Load and restore using *load()*.

Parameters
****kwargs** (Any) – Keyword arguments passed directly to xarray.save_mfdataset()

Returns
list[str] – Returns filenames of saved files

property shape

Return the shape of the dimensions.

Returns
tuple[int, int, int, int] – Shape of underlying data

property size

Return the size of (each) array in underlying *data*.

Returns
int – Total number of grid points in underlying data

to_polygon_feature(*level=None*, *time=None*, *fill_value=nan*, *iso_value=None*, *min_area=0.0*, *epsilon=0.0*, *lower_bound=True*, *precision=None*, *interiors=True*, *convex_hull=False*, *include_altitude=False*, *properties=None*)

Create GeoJSON Feature artifact from spatial array on a single level and time slice.

Computed polygons always contain an exterior linear ring as defined by the *GeoJSON Polygon specification <https://www.rfc-editor.org/rfc/rfc7946.html#section-3.1.6>*. Polygons may also contain interior linear rings (holes). This method does not support nesting beyond the GeoJSON specification. See the *pycontrails.core.polygon* for additional polygon support.

Changed in version 0.25.12: Previous implementation include several additional parameters which have been removed:

- The approximate parameter
- An path parameter to save output as JSON
- Passing arbitrary kwargs to skimage.measure.find_contours().

New implementation includes new parameters previously lacking:

- fill_value
- min_area
- include_altitude

Changed in version 0.38.0: Change default value of `epsilon` from 0.15 to 0.

Changed in version 0.41.0: Convert continuous fields to binary fields before computing polygons. The parameters `max_area` and `epsilon` are now expressed in terms of longitude/latitude units instead of pixels.

Parameters

- **level** (`float`, *optional*) – Level slice to create polygons. If the "level" coordinate is length 1, then the single level slice will be selected automatically.
- **time** (`datetime`, *optional*) – Time slice to create polygons. If the "time" coordinate is length 1, then the single time slice will be selected automatically.
- **fill_value** (`float`, *optional*) – Value used for filling missing data and for padding the underlying data array. Set to `np.nan` by default, which ensures that regions with missing data are never included in polygons.
- **iso_value** (`float`, *optional*) – Value in field to create iso-surface. Defaults to the average of the min and max value of the array. (This is the same convention as used by `skimage`.)
- **min_area** (`float`, *optional*) – Minimum area of each polygon. Polygons with area less than `min_area` are not included in the output. The unit of this parameter is in longitude/latitude degrees squared. Set to 0 to omit any polygon filtering based on a minimal area conditional. By default, 0.0.
- **epsilon** (`float`, *optional*) – Control the extent to which the polygon is simplified. A value of 0 does not alter the geometry of the polygon. The unit of this parameter is in longitude/latitude degrees. By default, 0.0.
- **lower_bound** (`bool`, *optional*) – Whether to use `iso_value` as a lower or upper bound on values in polygon interiors. By default, True.
- **precision** (`int`, *optional*) – Number of decimal places to round coordinates to. If None, no rounding is performed.
- **interiors** (`bool`, *optional*) – If True, include interior linear rings (holes) in the output. True by default.
- **convex_hull** (`bool`, *optional*) – EXPERIMENTAL. If True, compute the convex hull of each polygon. Only implemented for depth=1. False by default. A warning is issued if the underlying algorithm fails to make valid polygons after computing the convex hull.
- **include_altitude** (`bool`, *optional*) – If True, include the array altitude [m] as a z-coordinate in the *GeoJSON output <https://www.rfc-editor.org/rfc/rfc7946#section-3.1.1>*. False by default.
- **properties** (`dict`, *optional*) – Additional properties to include in the GeoJSON output. By default, None.

Returns

`dict[str, Any]` – Python representation of GeoJSON Feature with MultiPolygon geometry.

Notes

`Cocip` and `CocipGrid` set some quantities to 0 and other quantities to `np.nan` in regions where no contrails form. When computing polygons from `Cocip` or `CocipGrid` output, take care that the choice of `fill_value` correctly includes or excludes contrail-free regions. See the `Cocip` documentation for details about `np.nan` in model output.

See also:

to_polyhedra(), `pycontrails.core.polygons.find_multipolygons()`

Examples

```
>>> from pprint import pprint
>>> from pycontrails.datalib.ecmwf import ERA5
>>> era5 = ERA5("2022-03-01", variables="air_temperature", pressure_levels=250)
>>> mda = era5.open_metdataset()["air_temperature"]
>>> mda.shape
(1440, 721, 1, 1)
```

```
>>> pprint(mda.to_polygon_feature(iso_value=239.5, precision=2, epsilon=0.1))
{'geometry': {'coordinates': [[[[167.88, -22.5],
                                [167.75, -22.38],
                                [167.62, -22.5],
                                [167.75, -22.62],
                                [167.88, -22.5]]],
                              [[[43.38, -33.5],
                                [43.5, -34.12],
                                [43.62, -33.5],
                                [43.5, -33.38],
                                [43.38, -33.5]]]],
              'type': 'MultiPolygon'},
 'properties': {},
 'type': 'Feature'}
```

to_polygon_feature_collection(*time=None*, *fill_value=nan*, *iso_value=None*, *min_area=0.0*, *epsilon=0.0*, *lower_bound=True*, *precision=None*, *interiors=True*, *convex_hull=False*, *include_altitude=False*, *properties=None*)

Create GeoJSON FeatureCollection artifact from spatial array at time slice.

See the *to_polygon_feature()* method for a description of the parameters.

Returns

`dict[str, Any]` – Python representation of GeoJSON FeatureCollection. This dictionary is comprised of individual GeoJON Features, one per `self.data["level"]`.

to_polyhedra(***, *time=None*, *iso_value=0.0*, *simplify_fraction=1.0*, *lower_bound=True*, *return_type='geojson'*, *path=None*, *altitude_scale=1.0*, *output_vertex_normals=False*, *closed=True*)

Create a collection of polyhedra from spatial array corresponding to a single time slice.

Parameters

- **time** (`datetime`, *optional*) – Time slice to create mesh. If the "time" coordinate is length 1, then the single time slice will be selected automatically.

- **iso_value** (`float`, *optional*) – Value in field to create iso-surface. Defaults to 0.
- **simplify_fraction** (`float`, *optional*) – Apply *open3d simplify_quadric_decimation* method to simplify the polyhedra geometry. This parameter must be in the half-open interval (0.0, 1.0]. Defaults to 1.0, corresponding to no reduction.
- **lower_bound** (`bool`, *optional*) – Whether to use `iso_value` as a lower or upper bound on values in polyhedra interiors. By default, True.
- **return_type** (`str`, *optional*) – Must be one of "geojson" or "mesh". Defaults to "geojson". If "geojson", this method returns a dictionary representation of a geojson MultiPolygon object whose polygons are polyhedra faces. If "mesh", this method returns an *open3d TriangleMesh* instance.
- **path** (`str`, *optional*) – Output geojson or mesh to file. If *return_type* is "mesh", see Open3D File I/O for Mesh for file type options.
- **altitude_scale** (`float`, *optional*) – Rescale the altitude dimension of the mesh, $[m]$
- **output_vertex_normals** (`bool`, *optional*) – If `path` is defined, write out vertex normals. Defaults to False.
- **closed** (`bool`, *optional*) – If True, pad spatial array along all axes to ensure polyhedra are "closed". This flag often gives rise to cleaner visualizations. Defaults to True.

Returns

`dict | :class:`o3d.geometry.TriangleMesh`` – Python representation of geojson object or Open3D Triangle Mesh depending on the *return_type* parameter.

Raises

- `ModuleNotFoundError` – Method requires the *vis* optional dependencies
- `ValueError` – If input parameters are invalid.

See also:

`to_polygons()` skimage.measure.marching_cubes

Notes

Uses the scikit-image Marching Cubes algorithm to reconstruct a surface from the point-cloud like arrays.

property values

Return underlying numpy array.

This methods loads *data* if it is not already in memory.

Returns

`numpy.ndarray` – Underlying numpy array

See also:

-

meth:*xr.Dataset.load*

-

meth:*xr.DataArray.load*

wrap_longitude()

Wrap longitude coordinates.

Returns

MetDataArray – Copy of MetDataArray with wrapped longitude values. Returns copy of current MetDataArray when longitude values are already wrapped

class pycontrails.core.met.**MetDataset**(*data*, *cachestore=None*, *wrap_longitude=False*, *copy=True*, *attrs=None*, ***attrs_kwargs*)

Bases: *MetBase*

Meteorological dataset with multiple variables.

Composition around xr.Dataset to enforce certain variables and dimensions for internal usage

Parameters

- **data** (xarray.Dataset) – xarray.Dataset containing meteorological variables and coordinates
- **cachestore** (CacheStore, *optional*) – Cache datastore for staging intermediates with *save()*. Defaults to None.
- **wrap_longitude** (bool, *optional*) – Wrap data along the longitude dimension. If True, duplicate and shift longitude values (ie, -180 -> 180) to ensure that the longitude dimension covers the entire interval [-180, 180]. Defaults to False.
- **copy** (bool, *optional*) – Copy data on construction. Defaults to True.
- **attrs** (dict[str, Any], *optional*) – Attributes to add to data.attrs. Defaults to None. Generally, pycontrails Models may use the following attributes:
  - provider: Name of the data provider (e.g. "ECMWF").
  - dataset: Name of the dataset (e.g. "ERA5").
  - product: Name of the product type (e.g. "reanalysis").
- ****attrs_kwargs** (Any) – Keyword arguments to add to data.attrs. Defaults to None.

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> import xarray as xr
>>> from pycontrails.datalib.ecmwf import ERA5
```

```
>>> time = ("2022-03-01T00", "2022-03-01T02")
>>> variables = ["air_temperature", "specific_humidity"]
>>> pressure_levels = [200, 250, 300]
>>> era5 = ERA5(time, variables, pressure_levels)
```

```
>>> # Open directly as `MetDataset`
>>> met = era5.open_metdataset()
>>> # Use `data` attribute to access `xarray` object
>>> assert isinstance(met.data, xr.Dataset)
```

```
>>> # Alternatively, open with `xarray` and cast to `MetDataset`
>>> ds = xr.open_mfdataset(era5._cachepaths)
>>> met = MetDataset(ds)
```

```
>>> # Access sub-`DataArrays`
>>> mda = met["t"]  # `MetDataArray` instance, needed for interpolation operations
>>> da = mda.data  # Underlying `xarray` object
```

```
>>> # Check out a few values
>>> da[5:8, 5:8, 1, 1].values
array([[224.08959005, 224.41374427, 224.75945349],
       [224.09456429, 224.42037658, 224.76525676],
       [224.10036756, 224.42617985, 224.77106004]])
```

```
>>> # Mean temperature over entire array
>>> da.mean().load().item()
223.5083
```

broadcast_coords(*name*)

Broadcast coordinates along other dimensions.

Parameters

name (str) – Coordinate/dimension name to broadcast. Can be a dimension or non-dimension coordinates.

Returns

xarray.DataArray – DataArray of the coordinate broadcasted along all other dimensions. The DataArray will have the same shape as the gridded data.

cachestore

Cache datastore to use for *save()* or *load()*

copy()

Create a copy of the current class.

Returns

MetDataset – MetDataset copy

data

DataArray or Dataset

property dataset_attr

Look up the 'dataset' attribute with a custom error message.

Returns

str – Dataset of the data. If not one of 'ERA5', 'HRES', 'IFS', or 'GFS', a warning is issued.

downselect(*bbox*)

Downselect met data within spatial bounding box.

Parameters

bbox (list[float]) – List of coordinates defining a spatial bounding box in WGS84 coordinates. For 2D queries, list is [west, south, east, north]. For 3D queries, list is [west, south, min-level, east, north, max-level] with level defined in $[hPa]$.

Returns

MetBase – Return downselected data

ensure_vars(*vars*, *raise_error=True*)

Ensure variables exist in xr.Dataset.

Parameters

- **vars** (`MetVariable | str | Sequence[MetVariable | str | list[MetVariable]]`) – List of MetVariable (or string key), or individual MetVariable (or string key). If `vars` contains an element with a list[MetVariable], then only one variable in the list must be present in dataset.
- **raise_error** (`bool`, *optional*) – Raise KeyError if data does not contain variables. Defaults to True.

Returns

`list[str]` – List of met keys verified in MetDataset. Returns an empty list if any MetVariable is missing.

Raises

KeyError – Raises when dataset does not contain variable in `vars`

classmethod from_coords(*longitude*, *latitude*, *level*, *time*)

Create a *MetDataset* containing a coordinate skeleton from coordinate arrays.

Parameters

- **longitude, latitude** (`npt.ArrayLike | float`) – Horizontal coordinates, in [deg]
- **level** (`npt.ArrayLike | float`) – Vertical coordinate, in $[hPa]$
- **time** (`npt.ArrayLike | np.datetime64,`) – Temporal coordinates, in $[UTC]$. Will be sorted.

Returns

MetDataset – MetDataset with no variables.

Examples

```
>>> # Create skeleton MetDataset
>>> longitude = np.arange(0, 10, 0.5)
>>> latitude = np.arange(0, 10, 0.5)
>>> level = [250, 300]
>>> time = np.datetime64("2019-01-01")
>>> met = MetDataset.from_coords(longitude, latitude, level, time)
>>> met
MetDataset with data:
<xarray.Dataset> Size: 360B
Dimensions:       (longitude: 20, latitude: 20, level: 2, time: 1)
Coordinates:
  * longitude     (longitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * latitude      (latitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * level         (level) float64 16B 250.0 300.0
  * time          (time) datetime64[ns] 8B 2019-01-01
    air_pressure  (level) float32 8B 2.5e+04 3e+04
    altitude      (level) float32 8B 1.036e+04 9.164e+03
Data variables:
    *empty*
```

```
>>> met.shape
(20, 20, 2, 1)
```

```
>>> met.size
800
```

```
>>> # Fill it up with some constant data
>>> met["temperature"] = xr.DataArray(np.full(met.shape, 234.5), coords=met.
↪coords)
>>> met["humidity"] = xr.DataArray(np.full(met.shape, 0.5), coords=met.coords)
>>> met
MetDataset with data:
<xarray.Dataset> Size: 13kB
Dimensions:       (longitude: 20, latitude: 20, level: 2, time: 1)
Coordinates:
  * longitude     (longitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * latitude      (latitude) float64 160B 0.0 0.5 1.0 1.5 ... 8.0 8.5 9.0 9.5
  * level         (level) float64 16B 250.0 300.0
  * time          (time) datetime64[ns] 8B 2019-01-01
    air_pressure  (level) float32 8B 2.5e+04 3e+04
    altitude      (level) float32 8B 1.036e+04 9.164e+03
Data variables:
    temperature   (longitude, latitude, level, time) float64 6kB 234.5 ... 234.5
    humidity      (longitude, latitude, level, time) float64 6kB 0.5 0.5 ... 0.5
```

```
>>> # Convert to a GeoVectorDataset
>>> vector = met.to_vector()
>>> vector.dataframe.head()
longitude  latitude  level       time  temperature  humidity
0        0.0       0.0  250.0 2019-01-01        234.5       0.5
1        0.0       0.0  300.0 2019-01-01        234.5       0.5
2        0.0       0.5  250.0 2019-01-01        234.5       0.5
3        0.0       0.5  300.0 2019-01-01        234.5       0.5
4        0.0       1.0  250.0 2019-01-01        234.5       0.5
```

classmethod from_zarr(*store*, ***kwargs*)

Create a *MetDataset* from a path to a Zarr store.

Parameters

- **store** (Any) – Path to Zarr store. Passed into `xarray.open_zarr()`.
- ****kwargs** (Any) – Other keyword only arguments passed into `xarray.open_zarr()`.

Returns

MetDataset – MetDataset with data from Zarr store.

get(*key*, *default_value=None*)

Shortcut to `data.get(k, v)()` method.

Parameters

- **key** (`str`) – Key to get from *data*
- **default_value** (Any, *optional*) – Return *default_value* if *key* not in *data*, by default *None*

Returns
Any – Values returned from `data.get(key, default_value)`

classmethod load(*hash*, *cachestore=None*, *chunks=None*)

Load saved intermediate from *cachestore*.

Parameters

- **hash** (str) – Saved hash to load.
- **cachestore** (CacheStore, *optional*) – Cache datastore to use for sourcing files. Defaults to DiskCacheStore.
- **chunks** (dict[str: int], *optional*) – Chunks kwarg passed to xarray.open_mfdataset() when opening files.

Returns
MetDataset – New MetDataArray with loaded data.

property product_attr

Look up the 'product' attribute with a custom error message.

Returns
str – Product of the data. If not one of 'forecast', 'ensemble', or 'reanalysis', a warning is issued.

property provider_attr

Look up the 'provider' attribute with a custom error message.

Returns
str – Provider of the data. If not one of 'ECMWF' or 'NCEP', a warning is issued.

save(***kwargs*)

Save intermediate to *cachestore* as netcdf.

Load and restore using *load()*.

Parameters
****kwargs** (Any) – Keyword arguments passed directly to xarray.Dataset.to_netcdf()

Returns
list[str] – Returns filenames saved

property shape

Return the shape of the dimensions.

Returns
tuple[int, int, int, int] – Shape of underlying data

property size

Return the size of (each) array in underlying *data*.

Returns
int – Total number of grid points in underlying data

standardize_variables(*variables*)

Standardize variables **in-place**.

Parameters
variables (Iterable[MetVariable]) – Data source variables

See also:

standardize_variables()

to_vector(*transfer_attrs=True*)

Convert a *MetDataset* to a GeoVectorDataset by raveling data.

If *data* is lazy, it will be loaded.

Parameters

transfer_attrs (bool, *optional*) – Transfer attributes from *data* to output GeoVectorDataset. By default, True, meaning that attributes are transferred.

Returns

GeoVectorDataset – Converted GeoVectorDataset. The variables on the returned instance include all of those on the input instance, plus the four core spatial temporal variables.

Examples

```
>>> from pycontrails.datalib.ecmwf import ERA5
>>> times = "2022-03-01",  "2022-03-01T01"
>>> variables = ["air_temperature", "specific_humidity"]
>>> levels = [250, 200]
>>> era5 = ERA5(time=times, variables=variables, pressure_levels=levels)
>>> met = era5.open_metdataset()
>>> met.to_vector(transfer_attrs=False)
GeoVectorDataset [6 keys x 4152960 length, 1 attributes]
    Keys: longitude, latitude, level, time, air_temperature, ..., specific_
↪humidity
    Attributes:
    time                [2022-03-01 00:00:00, 2022-03-01 01:00:00]
    longitude           [-180.0, 179.75]
    latitude            [-90.0, 90.0]
    altitude            [10362.8, 11783.9]
    crs                 EPSG:4326
```

update(*other=None*, ***kwargs*)

Shortcut to data.update().

See xarray.Dataset.update() for reference.

Parameters

- **other** (MutableMapping) – Variables with which to update this dataset
- ****kwargs** (Any) – Variables defined by keyword arguments. If a variable exists both in other and as a keyword argument, the keyword argument takes precedence.

See also:

-

meth:*xarray.Dataset.update*

wrap_longitude()

Wrap longitude coordinates.

Returns

MetDataset – Copy of MetDataset with wrapped longitude values. Returns copy of current MetDataset when longitude values are already wrapped

pycontrails.core.met.**downselect**(*data*, *bbox*)

Downselect `xr.Dataset` or `xr.DataArray` with spatial bounding box.

Parameters

- **data** (`XArrayType`) – xr.Dataset or xr.DataArray to downselect
- **bbox** (`tuple[float, ]`) – Tuple of coordinates defining a spatial bounding box in WGS84 coordinates.
  - For 2D queries, `bbox` takes the form `(west, south, east, north)`
  - For 3D queries, `bbox` takes the form `(west, south, min-level, east, north, max-level)`

  with level defined in $[hPa]$.

Returns

`XArrayType` – Downselected xr.Dataset or xr.DataArray

Raises

`ValueError` – If parameter `bbox` has wrong length.

pycontrails.core.met.**originates_from_ecmwf**(*met*)

Check if data appears to have originated from an ECMWF source.

New in version 0.27.0: Experimental. Implementation is brittle.

Parameters

met (`MetDataset | MetDataArray`) – Dataset or array to inspect.

Returns

`bool` – True if data appears to be derived from an ECMWF source.

See also:

-

class:*ERA5*

-

class:*HRES*

pycontrails.core.met.**shift_longitude**(*data*, *bound=-180.0*)

Shift longitude values from any input domain to [bound, 360 + bound) domain.

Sorts data by ascending longitude values.

Parameters

- **data** (`XArrayType`) – `xr.Dataset` or `xr.DataArray` with longitude dimension
- **bound** (`float`, *optional*) – Lower bound of the domain. Output domain will be [bound, 360 + bound). Defaults to -180, which results in longitude domain [-180, 180).

Returns

`XArrayType` – `xr.Dataset` or `xr.DataArray` with longitude values on [a, 360 + a).

pycontrails.core.met.**standardize_variables**(*ds*, *variables*)

Rename all variables in dataset from short name to standard name.

This function does not change any variables in `ds` that are not found in `variables`.

When there are multiple variables with the same short name, the last one is used.

Parameters

- **ds** (`DatasetType`) – An `xr.Dataset` or *MetDataset*. When a *MetDataset* is passed, the underlying `xr.Dataset` is modified in place.
- **variables** (`Iterable[MetVariable]`) – Data source variables

Returns

`DatasetType` – Dataset with variables renamed to standard names

11.6.10 pycontrails.core.met_var

Module containing core met variables.

Classes

MetVariable(short_name, standard_name[, ...])	Met variable defined using CF, ECMWF, and WMO conventions.

class pycontrails.core.met_var.**MetVariable**(*short_name*, *standard_name*, *long_name=None*, *level_type=None*, *ecmwf_id=None*, *grib1_id=None*, *grib2_id=None*, *units=None*, *amip=None*, *description=None*)

Bases: `object`

Met variable defined using CF, ECMWF, and WMO conventions.

When there is a conflict between CF, ECMWF, and WMO conventions, CF takes precedence, then WMO, then ECMWF.

References

- CF Standard Names, version 77
- ECMWF Parameter Database
- NCEP Grib v1 Code Table
- WMO Codes Registry, Grib Edition 2
- NCEP Grib v2 Code Table

Used for defining support parameters in a grib-like fashion.

amip = None

AMIP identifier, if defined.

property attrs

Return a dictionary of met variable attributes.

Compatible with xr.Dataset or xr.DataArray attrs.

Returns

`dict[str, str]` – Dictionary with MetVariable attributes.

description = None

Description

ecmwf_id = None

ECMWF Grib variable id, if defined. See ECMWF Parameter Database

property ecmwf_link

Database link in the ECMWF Parameter Database if *ecmwf_id* is defined.

Returns

str – Database link in the ECMWF Parameter Database

grib1_id = None

WMO Grib v1 variable id, if defined. See WMO Codes Registry, Grib Edition 1 and CF Standard Names, version 77 # noqa: E501

grib2_id = None

WMO Grib 2 variable id, if defined. See WMO Codes Registry, Grib Edition 2 Tuple represents (disciple, category, number)

level_type = None

Level type One of "surface", "isobaricInhPa", "nominalTop"

long_name = None

Long variable name.

short_name

Short variable name. Chosen for greatest consistency between data sources.

standard_name

CF standard name, if defined. Otherwise a standard name is chosen for consistency.

units = None

Canonical CF units, if defined.

11.6.11 pycontrails.core.models

Physical model data structures.

Module Attributes

ModelInput	Model input source types
ModelOutput	Model output source types
SourceType	Model attribute source types

Functions

interpolate_gridded_specific_humidity(mda, ...)	Interpolate specific humidity against `vector` with experimental `q_method`.
interpolate_met(met, vector, met_key[, ...])	Interpolate `vector` against `met` gridded data.
raise_invalid_q_method_error(q_method)	Raise error for invalid `q_method`.
update_param_dict(param_dict, new_params)	Update parameter dictionary in place.

Classes

Model([met, params])	Base class for physical models.
ModelParams([copy_source, ...])	Class for constructing model parameters.

class pycontrails.core.models.**Model**(*met=None*, *params=None*, ***params_kwargs*)

Bases: ABC

Base class for physical models.

Implementing classes must implement the *eval()* method

default_params

Default model parameter dataclass

alias of *ModelParams*

downselect_met()

Downselect *met* domain to the max/min bounds of *source*.

Override this method if special handling is needed in met down-selection.

- *source* must be defined before calling *downselect_met()*.
- This method copies and re-assigns *met* using `met.copy()` to avoid side-effects.

Raises

- **ValueError** – Raised if *source* is not defined. Raised if *source* is not a `GeoVectorDataset`.
- **TypeError** – Raised if *met* is not a `MetDataset`.

abstract eval(*source=None*, ***params*)

Abstract method to handle evaluation.

Implementing classes should override call signature to overload `source` inputs and model outputs.

Parameters

- **source** (*ModelInput*, *optional*) – Dataset defining coordinates to evaluate model. Defined by implementing class, but must be a subset of ModelInput. If None, *met* is assumed to be evaluation points.
- ****params** (`Any`) – Overwrite model parameters before evaluation.

Returns

ModelOutput – Return type depends on implementing model

`get_source_param`(*key*, *default=<object object>*, ***, *set_attr=True*)

Get source data with default set by parameter key.

Retrieves data with the following hierarchy:

1. `source.data[key]`. Returns `np.ndarray | xr.DataArray`.
2. `source.attrs[key]`
3. `params[key]`
4. `default`

In case 3., the value of `params[key]` is attached to `source.attrs[key]`.

Parameters

- **key** (`str`) – Key to retrieve
- **default** (`Any`, *optional*) – Default value if key is not found.
- **set_attr** (`bool`, *optional*) – If True (default), set `source.attrs[key]` to `params[key]` if found. This allows for better post model evaluation tracking.

Returns

`Any` – Value(s) found for key in source data, source attrs, or model params

Raises

`KeyError` – Raises KeyError if key is not found in any location and `default` is not provided.

See also:

-

`property hash`

Generate a unique hash for model instance.

Returns

`str` – Unique hash for model instance (sha1)

`property interp_kwargs`

Shortcut to create interpolation arguments from *params*.

Returns

`dict[str, Any]` – Dictionary with keys

- "method"
- "bounds_error"
- "fill_value"
- "localize"
- "use_indices"
- "q_method"

as determined by *params*.

`abstract property long_name`

Get long name descriptor, annotated on `xr.DataArray` outputs.

`met`

Meteorology data

met_required = False

Require meteorology is not None on __init__()

met_variables

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

abstract property name

class`Flight`.

Type
Get model name for use as a data key in `xr.DataArray` or

optional_met_variables

Optional meteorology variables

params

Instantiated model parameters, in dictionary form

processed_met_variables

Set of required parameters if processing already complete on `met` input.

require_met()

Ensure that *met* is a MetDataset.

Returns
MetDataset – Returns reference to *met*. This is helpful for type narrowing *met* when meteorology is required.

Raises
ValueError – Raises when *met* is None.

require_source_type(*type_*)

Ensure that *source* is `type_`.

Returns
`_Source` – Returns reference to *source*. This is helpful for type narrowing *source* to specific type(s).

Raises
ValueError – Raises when *source* is not `_type_`.

set_source(*source=None*)

Attach original or copy of input `source` to *source*.

Parameters
source (`MetDataset | GeoVectorDataset | Flight | Iterable[Flight] | None`) – Parameter `source` passed in *eval()*. If None, an empty MetDataset with coordinates like *met* is set to *source*.

See also:

-
meth:*eval*

set_source_met(*optional=False*, *variable=None*)

Ensure or interpolate each required *met_variables* on *source* .

For each variable in *met_variables*, check *source* for data variable with the same name.

For `GeoVectorDataset` sources, try to interpolate *met* if variable does not exist.

For `MetDataset` sources, try to get data from *met* if variable does not exist.

Parameters

- **optional** (`bool`, *optional*) – Include *optional_met_variables*
- **variable** (`MetVariable | Sequence[MetVariable] | None`, *optional*) – MetVariable to set, from *met_variables*. If None, set all variables in *met_variables* and *optional_met_variables* if `optional` is True.

Raises

- **ValueError** – Variable does not exist and *source* is a MetDataset.
- **KeyError** – Variable not found in *source* or *met*.

source

Data evaluated in model

transfer_met_source_attrs(*source=None*)

Transfer met source metadata from *met* to `source`.

update_params(*params=None*, ***params_kwargs*)

Update model parameters on *params*.

Parameters

- **params** (`dict[str, Any]`, *optional*) – Model parameters to update, as dictionary. Defaults to {}
- ****params_kwargs** (`Any`) – Override keys in `params` with keyword arguments.

pycontrails.core.models.**ModelInput**

Model input source types

alias of `Optional[Union[MetDataset, GeoVectorDataset, Flight, Sequence[Flight]]]`

pycontrails.core.models.**ModelOutput**

Model output source types

alias of `Union[MetDataArray, MetDataset, GeoVectorDataset, Flight, list[Flight], NoReturn]`

class pycontrails.core.models.**ModelParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*)

Bases: `object`

Class for constructing model parameters.

Implementing classes must still use the `@dataclass` operator.

as_dict()

Convert object to dictionary.

We use this method instead of *dataclasses.asdict* to use a shallow/unrecursive copy. This will return values as Any instead of dict.

Returns
`dict[str, Any]` – Dictionary version of self.

copy_source = True

Copy input `source` data on eval

downselect_met = True

Downselect input `MetDataset` to region around `source`.

interpolation_bounds_error = False

If True, points lying outside interpolation will raise an error

interpolation_fill_value = nan

Used for outside interpolation value if *interpolation_bounds_error* is False

interpolation_localize = False

Experimental. See *pycontrails.core.interpolation*.

interpolation_method = 'linear'

Interpolation method. Supported methods include "linear", "nearest", "slinear", "cubic", and "quintic". See `scipy.interpolate.RegularGridInterpolator` for the description of each method. Not all methods are supported by all met grids. For example, the "cubic" method requires at least 4 points per dimension.

interpolation_q_method = None

Experimental. Alternative interpolation method to account for specific humidity lapse rate bias. Must be one of `None`, `"cubic-spline"`, or `"log-q-log-p"`. If `None`, no special interpolation is used for specific humidity. The `"cubic-spline"` method applies a custom stretching of the met interpolation table to account for the specific humidity lapse rate bias. The `"log-q-log-p"` method interpolates in the log of specific humidity and pressure, then converts back to specific humidity. Only used by models calling to *interpolate_met()*.

interpolation_use_indices = False

Experimental. See *pycontrails.core.interpolation*.

met_latitude_buffer = (0.0, 0.0)

Met latitude buffer for input to `Flight.downselect_met()`, in WGS84 coordinates. Only applies when *downselect_met* is True.

met_level_buffer = (0.0, 0.0)

Met level buffer for input to `Flight.downselect_met()`, in $[hPa]$. Only applies when *downselect_met* is True.

met_longitude_buffer = (0.0, 0.0)

Met longitude buffer for input to `Flight.downselect_met()`, in WGS84 coordinates. Only applies when *downselect_met* is True.

met_time_buffer = (numpy.timedelta64(0,'h'), numpy.timedelta64(0,'h'))

Met time buffer for input to `Flight.downselect_met()` Only applies when *downselect_met* is True.

verify_met = True

Call `_verify_met()` on model instantiation.

pycontrails.core.models.**SourceType**

Model attribute source types

alias of Union[*MetDataset*, *GeoVectorDataset*, *Flight*, *Fleet*]

pycontrails.core.models.**interpolate_gridded_specific_humidity**(*mda*, *vector*, *q_method*, *log_applied*, ***interp_kwargs*)

Interpolate specific humidity against `vector` with experimental `q_method`.

Parameters

- **mda** (*MetDataArray*) – MetDataArray of specific humidity.
- **vector** (*GeoVectorDataset*) – Flight or GeoVectorDataset instance
- **q_method** ({None, "cubic-spline", "log-q-log-p"}) – Experimental method to use for interpolating specific humidity.
- **log_applied** (bool) – Whether or not a log transform was applied to specific humidity.
- ****interp_kwargs** (Any,) – Additional keyword only arguments passed to *intersect_met*.

Returns

numpy.ndarray – Interpolated values.

pycontrails.core.models.**interpolate_met**(*met*, *vector*, *met_key*, *vector_key=None*, *q_method=None*, ***interp_kwargs*)

Interpolate `vector` against `met` gridded data.

If `vector_key` (=``met_key`` by default) already exists, return values at `vector_key`.

Mutates parameter `vector` in place by attaching new key and returns values.

Parameters

- **met** (MetDataset | None) – Met data to interpolate against
- **vector** (*GeoVectorDataset*) – Flight or GeoVectorDataset instance
- **met_key** (str) – Key of met variable in `met`.
- **vector_key** (str, *optional*) – Key of variable to attach to `vector`. By default, use `met_key`.
- **q_method** (str, *optional*) – Experimental method to use for interpolating specific humidity. See *ModelParams* for more information.
- ****interp_kwargs** (Any,) – Additional keyword only arguments passed to `GeoVectorDataset.intersect_met()`. For example, `level=[...]`.

Returns

npt.NDArray[np.float64] – Interpolated values.

Raises

KeyError – Parameter `met_key` not found in `met`.

pycontrails.core.models.**raise_invalid_q_method_error**(*q_method*)

Raise error for invalid `q_method`.

Parameters

q_method (str) – `q_method` to raise error for.

Raises

ValueError – `q_method` is not one of `None`, `"log-q-log-p"`, or `"cubic-spline"`.

pycontrails.core.models.**update_param_dict**(*param_dict*, *new_params*)

Update parameter dictionary in place.

Parameters

- **param_dict** (dict[str, Any]) – Active model parameter dictionary
- **new_params** (dict[str, Any]) – Model parameters to update, as a dictionary

Raises

KeyError – Raises when new_params key is not found in param_dict

11.6.12 pycontrails.core.polygon

Algorithm support for grid to polygon conversion.

See also:

pycontrails.MetDataArray.to_polygon_feature(), *pycontrails.MetDataArray.to_polygon_feature_collection()*

Functions

buffer_and_clean(contour, min_area, ...)	Buffer and clean a contour.
determine_buffer(longitude, latitude)	Determine the proper buffer size to use when converting to polygons.
find_multipolygon(arr, threshold, min_area, ...)	Compute a multipolygon from a 2d array.
multipolygon_to_geojson(multipolygon, altitude)	Convert a shapely multipolygon to a GeoJSON feature.

pycontrails.core.polygon.**buffer_and_clean**(*contour*, *min_area*, *convex_hull*, *epsilon*, *precision*, *buffer*, *is_exterior*)

Buffer and clean a contour.

Parameters

- **contour** (npt.NDArray[np.float64]) – Contour to buffer and clean. A 2d array of shape (n, 2) where n is the number of vertices in the contour.
- **min_area** (float) – Minimum area of the polygon. If the area of the buffered contour is less than this, return None.
- **convex_hull** (bool) – Whether to take the convex hull of the buffered contour.
- **epsilon** (float) – Epsilon value for polygon simplification. If 0, no simplification is performed.
- **precision** (int | None) – Precision of the output polygon. If None, no rounding is performed.
- **buffer** (float) – Buffer distance.
- **is_exterior** (bool, *optional*) – Whether the contour is an exterior contour. If True, the contour is buffered with a larger buffer distance. The polygon orientation is CCW iff this is True.

Returns

shapely.Polygon | None – Buffered and cleaned polygon. If the area of the buffered contour is less than min_area, return None.

pycontrails.core.polygon.**determine_buffer**(*longitude*, *latitude*)

Determine the proper buffer size to use when converting to polygons.

pycontrails.core.polygon.**find_multipolygon**(*arr*, *threshold*, *min_area*, *epsilon*, *lower_bound=True*, *interiors=True*, *convex_hull=False*, *longitude=None*, *latitude=None*, *precision=None*)

Compute a multipolygon from a 2d array.

Parameters

- **arr** (`npt.NDArray[np.float64]`) – Array to convert to a multipolygon. The array will be converted to a binary array by comparing each element to `threshold`. This binary array is then passed into `cv2.findContours()` to find the contours.
- **threshold** (`float`) – Threshold to use when converting `arr` to a binary array.
- **min_area** (`float`) – Minimum area of a polygon to be included in the output.
- **epsilon** (`float`) – Epsilon value to use when simplifying the polygons. Passed into shapely's `shapely.geometry.Polygon.simplify()` method.
- **lower_bound** (`bool`, *optional*) – Whether to treat `threshold` as a lower or upper bound on values in polygon interiors. By default, True.
- **interiors** (`bool`, *optional*) – Whether to include interior polygons. By default, True.
- **convex_hull** (`bool`, *optional*) – Experimental. Whether to take the convex hull of each polygon. By default, False.
- **longitude** (`npt.NDArray[np.float64] | None`, *optional*) – If provided, the coordinates values corresponding to the longitude dimensions of `arr`. The contour coordinates will be converted to longitude-latitude values by indexing into this array. Defaults to None.
- **latitude** (`npt.NDArray[np.float64] | None`, *optional*) – If provided, the coordinates values corresponding to the latitude dimensions of `arr`.
- **precision** (`int | None`, *optional*) – If provided, the precision to use when rounding the coordinates. Defaults to None.

Returns

`shapely.MultiPolygon` – A multipolygon of the contours.

Raises

`ValueError` – If `arr` is not 2d.

pycontrails.core.polygon.**multipolygon_to_geojson**(*multipolygon*, *altitude*, *properties=None*)

Convert a shapely multipolygon to a GeoJSON feature.

Parameters

- **multipolygon** (`shapely.MultiPolygon`) – Multipolygon to convert.
- **altitude** (`float | None`) – Altitude of the multipolygon. If provided, the multipolygon coordinates will be given a z-coordinate.
- **properties** (`dict[str, Any] | None`, *optional*) – Properties to add to the GeoJSON feature.

Returns

`dict[str, Any]` – GeoJSON feature with geometry type "MultiPolygon".

11.6.13 pycontrails.core.vector

Lightweight data structures for vector paths.

Module Attributes

VectorDatasetType	Vector types

Functions

vector_to_lon_lat_grid(vector, agg, *[, ...])	Convert vectors to a longitude-latitude grid.

Classes

AttrDict	Thin wrapper around dict to warn when setting a key that already exists.
GeoVectorDataset([data, longitude, ...])	Base class to hold 1D geospatial arrays of consistent size.
VectorDataDict([data])	Thin wrapper around `dict[str, np.ndarray]` to ensure consistency.
VectorDataset([data, attrs, copy])	Base class to hold 1D arrays of consistent size.

class pycontrails.core.vector.**AttrDict**

Bases: dict[str, Any]

Thin wrapper around dict to warn when setting a key that already exists.

setdefault(*k*, *default=None*)

Thin wrapper around `dict.setdefault`.

Overwrites value if value is None.

Parameters

- **k** (str) – Key
- **default** (Any, *optional*) – Default value for key k

Returns

Any – Value at k

class pycontrails.core.vector.**GeoVectorDataset**(*data=None*, ***, *longitude=None*, *latitude=None*, *altitude=None*, *altitude_ft=None*, *level=None*, *time=None*, *attrs=None*, *copy=True*, ***attrs_kwargs*)

Bases: *VectorDataset*

Base class to hold 1D geospatial arrays of consistent size.

GeoVectorDataset is required to have geospatial coordinate keys defined in *required_keys*.

Expect latitude-longitude CRS in WGS 84. Expect altitude in $[m]$. Expect level in $[hPa]$.

Each spatial variable is expected to have "float32" or "float64" `dtype`. The time variable is expected to have "datetime64[ns]" `dtype`.

Use the attribute `attr["crs"]` to specify coordinate reference system using PROJ or EPSG syntax.

Parameters

- **data** (`dict[str, npt.ArrayLike] | pd.DataFrame | VectorDataDict | VectorDataset | None`, *optional*) – Data dictionary or `pandas.DataFrame` . Must include keys/columns `time`, `latitude`, `longitude`, `altitude` or `level`. Keyword arguments for `time`, `latitude`, `longitude`, `altitude` or `level` override `data` inputs. Expects `altitude` in meters and `time` as a DatetimeLike (or array that can processed with `pd.to_datetime()`). Additional waypoint-specific data can be included as additional keys/columns.
- **longitude** (`npt.ArrayLike`, *optional*) – Longitude data. Defaults to None.
- **latitude** (`npt.ArrayLike`, *optional*) – Latitude data. Defaults to None.
- **altitude** (`npt.ArrayLike`, *optional*) – Altitude data, [m]. Defaults to None.
- **altitude_ft** (`npt.ArrayLike`, *optional*) – Altitude data, [ft]. Defaults to None.
- **level** (`npt.ArrayLike`, *optional*) – Level data, [hPa]. Defaults to None.
- **time** (`npt.ArrayLike`, *optional*) – Time data. Expects an array of DatetimeLike values, or array that can processed with `pd.to_datetime()`. Defaults to None.
- **attrs** (`dict[Hashable, Any] | AttrDict`, *optional*) – Additional properties as a dictionary. Defaults to {}.
- **copy** (`bool`, *optional*) – Copy data on class creation. Defaults to True.
- ****attrs_kwargs** (`Any`) – Additional properties passed as keyword arguments.

Raises

`KeyError` – Raises if `data` input does not contain at least `time`, `latitude`, `longitude`, (`altitude` or `level`).

`T_isa()`

Calculate the ICAO standard atmosphere temperature at each point.

Returns

`npt.NDArray[np.float64]` – ISA temperature, [K]

See also:

pycontrails.physics.units.m_to_T_isa()

`property air_pressure`

Get `air_pressure` values for points.

Returns

`npt.NDArray[np.float64]` – Point air pressure values, [Pa]

`property altitude`

Get altitude.

Automatically calculates altitude using `units.pl_to_m()` using `level` key.

Note that if `altitude` key exists in `data`, the data at the `altitude` key will be returned. This allows an override of the default calculation of altitude from pressure level.

Returns

`npt.NDArray[np.float64]` – Altitude, [m]

property altitude_ft

Get altitude in feet.

Returns
npt.NDArray[np.float64] – Altitude, $[ft]$

property constants

Return a dictionary of constant attributes and data values.

Includes attrs and values from columns in data with a unique value.

Returns
dict[str, Any] – Properties and their constant values

property coords

Get geospatial coordinates for compatibility with MetDataArray.

Returns
pandas.DataFrame – pd.DataFrame with columns *longitude*, *latitude*, *level*, and *time*.

coords_intersect_met(*met*)

Return boolean mask of data inside the bounding box defined by met.

Parameters
met (MetDataset | MetDataArray) – MetDataset or MetDataArray to compare.

Returns
npt.NDArray[np.bool_] – True if point is inside the bounding box defined by met.

classmethod create_empty(*keys=None*, *attrs=None*, ***attrs_kwargs*)

Create instance with variables defined by *keys* and size 0.

If instance requires additional variables to be defined, these keys will automatically be attached to returned instance.

Parameters

- **keys** (Iterable[str]) – Keys to include in empty VectorDataset instance.
- **attrs** (dict[str, Any] | None, *optional*) – Attributes to attach instance.
- ****attrs_kwargs** (Any) – Define attributes as keyword arguments.

Returns
VectorDatasetType – Empty VectorDataset instance.

downselect_met(*met*, ***, *longitude_buffer=(0.0, 0.0)*, *latitude_buffer=(0.0, 0.0)*, *level_buffer=(0.0, 0.0)*, *time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *copy=True*)

Downselect met to encompass a spatiotemporal region of the data.

Parameters

- **met** (MetDataset | MetDataArray) – MetDataset or MetDataArray to downselect.
- **longitude_buffer** (tuple[float, float], *optional*) – Extend longitude domain past by longitude_buffer[0] on the low side and longitude_buffer[1] on the high side. Units must be the same as class coordinates. Defaults to (0, 0) degrees.
- **latitude_buffer** (tuple[float, float], *optional*) – Extend latitude domain past by latitude_buffer[0] on the low side and latitude_buffer[1] on the high side. Units must be the same as class coordinates. Defaults to (0, 0) degrees.

- **level_buffer** (`tuple[float, float]`, *optional*) – Extend level domain past by `level_buffer[0]` on the low side and `level_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(0, 0)` [hPa].
- **time_buffer** (`tuple[np.timedelta64, np.timedelta64]`, *optional*) – Extend time domain past by `time_buffer[0]` on the low side and `time_buffer[1]` on the high side. Units must be the same as class coordinates. Defaults to `(np.timedelta64(0, "h"), np.timedelta64(0, "h"))`.
- **copy** (`bool`) – If returned object is a copy or view of the original. True by default.

Returns

`MetDataset | MetDataArray` – Copy of downselected MetDataset or MetDataArray.

intersect_met(*mda*, *, *longitude=None*, *latitude=None*, *level=None*, *time=None*, *use_indices=False*, ***interp_kwargs*)

Intersect waypoints with MetDataArray.

Parameters

- **mda** (*MetDataArray*) – MetDataArray containing a meteorological variable at spatio-temporal coordinates.
- **longitude** (`npt.NDArray[np.float64]`, *optional*) – Override existing coordinates for met interpolation
- **latitude** (`npt.NDArray[np.float64]`, *optional*) – Override existing coordinates for met interpolation
- **level** (`npt.NDArray[np.float64]`, *optional*) – Override existing coordinates for met interpolation
- **time** (`npt.NDArray[np.datetime64]`, *optional*) – Override existing coordinates for met interpolation
- **use_indices** (`bool`, *optional*) – Experimental.
- ****interp_kwargs** (`Any`) – Additional keyword arguments to pass to `MetDataArray.intersect_met()`. Examples include `method`, `bounds_error`, and `fill_value`. If an error such as

  ```
  ValueError: One of the requested xi is out of bounds in dimension 2
  ```

  occurs, try calling this function with `bounds_error=False`. In addition, setting `fill_value=0.0` will replace NaN values with 0.0.

Returns

`npt.NDArray[np.float64]` – Interpolated values

Examples

```
>>> from datetime import datetime
>>> import pandas as pd
>>> import numpy as np
>>> from pycontrails.datalib.ecmwf import ERA5
>>> from pycontrails import Flight
```

```
>>> # Get met data
>>> times = (datetime(2022, 3, 1, 0),  datetime(2022, 3, 1, 3))
>>> variables = ["air_temperature", "specific_humidity"]
>>> levels = [300, 250, 200]
>>> era5 = ERA5(time=times, variables=variables, pressure_levels=levels)
>>> met = era5.open_metdataset()
```

```
>>> # Example flight
>>> df = pd.DataFrame()
>>> df['longitude'] = np.linspace(0, 50, 10)
>>> df['latitude'] = np.linspace(0, 10, 10)
>>> df['altitude'] = 11000
>>> df['time'] = pd.date_range("2022-03-01T00", "2022-03-01T02", periods=10)
>>> fl = Flight(df)
```

```
>>> # Intersect
>>> fl.intersect_met(met['air_temperature'], method='nearest')
array([231.62969892, 230.72604651, 232.24318771, 231.88338483,
       231.06429438, 231.59073409, 231.65125393, 231.93064004,
       232.03344087, 231.65954432])
```

```
>>> fl.intersect_met(met['air_temperature'], method='linear')
array([225.77794552, 225.13908414, 226.231218  , 226.31831528,
       225.56102321, 225.81192149, 226.03192642, 226.22056121,
       226.03770174, 225.63226188])
```

```
>>> # Interpolate and attach to `Flight` instance
>>> for key in met:
...     fl[key] = fl.intersect_met(met[key])
```

```
>>> # Show the final three columns of the dataframe
>>> fl.dataframe.iloc[:, -3:].head()
                 time  air_temperature  specific_humidity
0 2022-03-01 00:00:00       225.777946           0.000132
1 2022-03-01 00:13:20       225.139084           0.000132
2 2022-03-01 00:26:40       226.231218           0.000107
3 2022-03-01 00:40:00       226.318315           0.000171
4 2022-03-01 00:53:20       225.561022           0.000109
```

property level

Get pressure `level` values for points.

Automatically calculates pressure level using `units.m_to_pl()` using `altitude` key.

Note that if `level` key exists in `data`, the data at the `level` key will be returned. This allows an override of the default calculation of pressure level from altitude.

Returns

`npt.NDArray[np.float64]` – Point pressure level values, $[hPa]$

required_keys = ('longitude', 'latitude', 'time')

Required keys for creating GeoVectorDataset

to_geojson_points()

Return dataset as GeoJSON FeatureCollection of Points.

Each Feature has a properties attribute that includes time and other data besides latitude, longitude, and altitude in data.

Returns
dict[str, Any] – Python representation of GeoJSON FeatureCollection

to_lon_lat_grid(*agg*, *, *spatial_bbox=(-180.0, -90.0, 180.0, 90.0)*, *spatial_grid_res=0.5*)

Convert vectors to a longitude-latitude grid.

See also:

vector_to_lon_lat_grid

to_pseudo_mercator(*copy=True*)

Convert data from attrs["crs"] to Pseudo Mercator (EPSG:3857).

Parameters
copy (bool, *optional*) – Copy data on transformation. Defaults to True.

Returns
GeoVectorDatasetType

transform_crs(*crs*, *copy=True*)

Transform trajectory data from one coordinate reference system (CRS) to another.

Parameters

- **crs** (str) – Target CRS. Passed into to pyproj.Transformer. The source CRS is inferred from the attrs["crs"] attribute.
- **copy** (bool, *optional*) – Copy data on transformation. Defaults to True.

Returns
GeoVectorDatasetType – Converted dataset with new coordinate reference system. attrs["crs"] reflects new crs.

vertical_keys = ('altitude', 'level', 'altitude_ft')

At least one of these vertical-coordinate keys must also be included

class pycontrails.core.vector.VectorDataDict(*data=None*)

Bases: dict[str, ndarray]

Thin wrapper around dict[str, np.ndarray] to ensure consistency.

Parameters
data (dict[str, np.ndarray], *optional*) – Dictionary input

setdefault(*k*, *default=None*)

Thin wrapper around dict.setdefault.

The main purpose of overriding is to run _validate_array() on set.

Parameters

- **k** (str) – Key
- **default** (npt.ArrayLike, *optional*) – Default value for key k

Returns
Any – Value at k

update(*other=None*, ***kwargs*)

Update values without warning if overwriting.

This method casts values in `other` to `numpy.ndarray` and ensures that the array sizes are consistent with the instance.

Parameters

- **other** (`dict[str, npt.ArrayLike] | None`, *optional*) – Fields to update as dict
- ****kwargs** (`npt.ArrayLike`) – Fields to update as kwargs

class pycontrails.core.vector.**VectorDataset**(*data=None*, ***, *attrs=None*, *copy=True*, ***attrs_kwargs*)

Bases: `object`

Base class to hold 1D arrays of consistent size.

Parameters

- **data** (`dict[str, npt.ArrayLike] | pd.DataFrame | VectorDataDict | VectorDataset | None`, *optional*) – Initial data, by default None
- **attrs** (`dict[str, Any] | AttrDict`, *optional*) – Dictionary of attributes, by default None
- **copy** (`bool`, *optional*) – Copy data on class creation, by default True
- ****attrs_kwargs** (`Any`) – Additional attributes passed as keyword arguments

Raises

`ValueError` – If "time" variable cannot be converted to numpy array.

attrs

Generic dataset attributes

broadcast_attrs(*keys*, *overwrite=False*, *raise_error=True*)

Attach values from `keys` in *attrs* onto *data*.

If possible, use `dtype = np.float32` when broadcasting. If not possible, use whatever `dtype` is inferred from the data by `numpy.full()`.

Parameters

- **keys** (`str | Iterable[str]`) – Keys to broadcast
- **overwrite** (`bool`, *optional*) – If True, overwrite existing values in *data*. By default False.
- **raise_error** (`bool`, *optional*) – Raise KeyError if `self.attrs` does not contain some of `keys`.

Raises

`KeyError` – Not all `keys` found in *attrs*.

broadcast_numeric_attrs(*ignore_keys=None*, *overwrite=False*)

Attach numeric values in *attrs* onto *data*.

Iterate through values in *attrs* and attach `float` and `int` values to `data`.

This method modifies object in place.

Parameters

- **ignore_keys** (`str | Iterable[str]`, *optional*) – Do not broadcast selected keys. Defaults to None.
- **overwrite** (`bool`, *optional*) – If True, overwrite existing values in *data*. By default False.

`copy`(***kwargs*)

Return a copy of this VectorDatasetType class.

Parameters

****kwargs** (`Any`) – Additional keyword arguments passed into the constructor of the returned class.

Returns

VectorDatasetType – Copy of class

`classmethod create_empty`(*keys*, *attrs=None*, ***attrs_kwargs*)

Create instance with variables defined by *keys* and size 0.

If instance requires additional variables to be defined, these keys will automatically be attached to returned instance.

Parameters

- **keys** (`Iterable[str]`) – Keys to include in empty VectorDataset instance.
- **attrs** (`dict[str, Any] | None`, *optional*) – Attributes to attach instance.
- ****attrs_kwargs** (`Any`) – Define attributes as keyword arguments.

Returns

VectorDatasetType – Empty VectorDataset instance.

`data`

Vector data with labels as keys and `numpy.ndarray` as values

`property dataframe`

Shorthand property to access *to_dataframe()* with `copy=False`.

Returns

`pandas.DataFrame` – Equivalent to the output from *to_dataframe()*

`ensure_vars`(*vars*, *raise_error=True*)

Ensure variables exist in column of *data* or *attrs*.

Parameters

- **vars** (`str | Iterable[str]`) – A single string variable name or a sequence of string variable names.
- **raise_error** (`bool`, *optional*) – Raise KeyError if data does not contain variables. Defaults to True.

Returns

`bool` – True if all variables exist. False otherwise.

Raises

KeyError – Raises when dataset does not contain variable in `vars`

`filter`(*mask*, *copy=True*, ***kwargs*)

Filter *data* according to a boolean array `mask`.

Entries corresponding to `mask == True` are kept.

Parameters

- **mask** (`npt.NDArray[np.bool_]`) – Boolean array with compatible shape.
- **copy** (`bool`, *optional*) – Copy data on filter. Defaults to True. See numpy best practices for insight into whether copy is appropriate.

- ****kwargs** (Any) – Additional keyword arguments passed into the constructor of the returned class.

Returns
VectorDatasetType – Containing filtered data

Raises
TypeError – If mask is not a boolean array.

classmethod from_dict(*obj*, *copy=True*, ***obj_kwargs*)

Create instance from dict representation containing data and attrs.

Parameters

- **obj** (dict[str, Any]) – Dict representation of VectorDataset (e.g. *to_dict()*)
- **copy** (bool, *optional*) – Passed to *VectorDataset* constructor. Defaults to True.
- ****obj_kwargs** (Any) – Additional properties passed as keyword arguments.

Returns
VectorDatasetType – VectorDataset instance.

See also:

to_dict()

generate_splits(*n_splits*, *copy=True*)

Split instance into n_split sub-vectors.

Parameters

- **n_splits** (int) – Number of splits.
- **copy** (bool, *optional*) – Passed into *filter()*. Defaults to True. Recommend to keep as True based on numpy best practices.

Returns
Generator[VectorDatasetType, None, None] – Generator of split vectors.

See also:

numpy.array_split()

get(*key*, *default_value=None*)

Get values from *data* with default_value if key not in *data*.

Parameters

- **key** (str) – Key to get from *data*
- **default_value** (Any, *optional*) – Return default_value if *key* not in *data*, by default None

Returns
Any – Values at data[key] or default_value

get_data_or_attr(*key*, *default=<object object>*)

Get value from *data* or *attrs*.

This method first checks if key is in *data* and returns the value if so. If key is not in *data*, then this method checks if key is in *attrs* and returns the value if so. If key is not in *data* or *attrs*, then the default value is returned if provided. Otherwise a KeyError is raised.

Parameters

- **key** (str) – Key to get from *data* or *attrs*
- **default** (Any, *optional*) – Default value to return if key is not in *data* or *attrs*.

Returns
Any – Value at data[key] or attrs[key]

Raises
KeyError – If key is not in *data* or *attrs* and default is not provided.

Examples

```
>>> vector = VectorDataset({"a": [1, 2, 3]}, attrs={"b": 4})
>>> vector.get_data_or_attr("a")
array([1, 2, 3])
```

```
>>> vector.get_data_or_attr("b")
4
```

```
>>> vector.get_data_or_attr("c")
Traceback (most recent call last):
...
KeyError: "Key 'c' not found in data or attrs."
```

```
>>> vector.get_data_or_attr("c", default=5)
5
```

property hash

Generate a unique hash for this class instance.

Returns
str – Unique hash for flight instance (sha1)

select(*keys*, *copy=True*)

Return new class instance only containing specified keys.

Parameters

- **keys** (Iterable[str]) – An iterable of keys to filter by.
- **copy** (bool, *optional*) – Copy data on selection. Defaults to True.

Returns
VectorDataset – VectorDataset containing only data associated to keys. Note that this method always returns a *VectorDataset*, even if the calling class is a proper subclass of *VectorDataset*.

setdefault(*key*, *default=None*)

Shortcut to *VectorDataDict.setdefault()*.

Parameters

- **key** (str) – Key in *data* dict.
- **default** (npt.ArrayLike, *optional*) – Values to use as default, if key is not defined

Returns
numpy.ndarray – Values at key

property shape

Shape of each array in *data*.

Returns
`tuple[int]` – Shape of each array in *data*.

property size

Length of each array in *data*.

Returns
`int` – Length of each array in *data*.

sort(*by*)

Sort data by key(s).

This method always creates a copy of the data by calling `pandas.DataFrame.sort_values()`.

Parameters
by (`str | list[str]`) – Key or list of keys to sort by.

Returns
VectorDatasetType – Instance with sorted data.

classmethod sum(*vectors*, *infer_attrs=True*, *fill_value=None*)

Sum a list of *VectorDataset* instances.

Parameters

- **vectors** (`Sequence[VectorDataset]`) – List of *VectorDataset* instances to concatenate.
- **infer_attrs** (`bool`, *optional*) – If True, infer attributes from the first element in the sequence.
- **fill_value** (`float`, *optional*) – Fill value to use when concatenating arrays. By default None, which raises an error if incompatible keys are found.

Returns
VectorDataset – Sum of all instances in `vectors`.

Raises
KeyError – If incompatible *data* keys are found among `vectors`.

Examples

```
>>> from pycontrails import VectorDataset
>>> v1 = VectorDataset({"a": [1, 2, 3], "b": [4, 5, 6]})
>>> v2 = VectorDataset({"a": [7, 8, 9], "b": [10, 11, 12]})
>>> v3 = VectorDataset({"a": [13, 14, 15], "b": [16, 17, 18]})
>>> v = VectorDataset.sum([v1, v2, v3])
>>> v.dataframe
    a   b
0   1   4
1   2   5
2   3   6
3   7  10
4   8  11
5   9  12
```

(continues on next page)

(continued from previous page)

```
6  13  16
7  14  17
8  15  18
```

to_dataframe(*copy=True*)

Create `pd.DataFrame` in which each key-value pair in *data* is a column.

DataFrame does **not** copy data by default. Use the `copy` parameter to copy data values on creation.

Parameters

copy (`bool`, *optional*) – Copy data on DataFrame creation.

Returns

`pandas.DataFrame` – DataFrame holding key-values as columns.

to_dict()

Create dictionary with *data* and *attrs*.

If geo-spatial coordinates (e.g. `"latitude"`, `"longitude"`, `"altitude"`) are present, round to a reasonable precision. If a `"time"` variable is present, round to unix seconds. When the instance is a *GeoVectorDataset*, disregard any `"altitude"` or `"level"` coordinate and only include `"altitude_ft"` in the output.

Returns

`dict[str, Any]` – Dictionary with *data* and *attrs*.

See also:

from_dict()

Examples

```
>>> import pprint
>>> from pycontrails import Flight
>>> fl = Flight(
...     longitude=[-100, -110],
...     latitude=[40, 50],
...     level=[200, 200],
...     time=[np.datetime64("2020-01-01T09"), np.datetime64("2020-01-01T09:30
↪")],
...     aircraft_type="B737",
... )
>>> fl = fl.resample_and_fill("5min")
>>> pprint.pprint(fl.to_dict())
{'aircraft_type': 'B737',
 'altitude_ft': [38661.0, 38661.0, 38661.0, 38661.0, 38661.0, 38661.0, 38661.0],
 'crs': 'EPSG:4326',
 'latitude': [40.0, 41.724, 43.428, 45.111, 46.769, 48.399, 50.0],
 'longitude': [-100.0,
               -101.441,
               -102.959,
               -104.563,
               -106.267,
               -108.076,
```

(continues on next page)

(continued from previous page)

```
                -110.0],
 'time': [1577869200,
          1577869500,
          1577869800,
          1577870100,
          1577870400,
          1577870700,
          1577871000]}
```

update(*other=None*, ***kwargs*)

Update values in *data* dict without warning if overwriting.

Parameters

- **other** (dict[str, npt.ArrayLike] | None, *optional*) – Fields to update as dict
- ****kwargs** (npt.ArrayLike) – Fields to update as kwargs

class pycontrails.core.vector.**VectorDatasetType**

Vector types

alias of TypeVar('VectorDatasetType', bound=*VectorDataset*)

pycontrails.core.vector.**vector_to_lon_lat_grid**(*vector*, *agg*, ***, *spatial_bbox=(-180.0, -90.0, 180.0, 90.0)*, *spatial_grid_res=0.5*)

Convert vectors to a longitude-latitude grid.

Parameters

- **vector** (*GeoVectorDataset*) – Contains the longitude, latitude and variables for aggregation.
- **agg** (dict[str, str]) – Variable name and the function selected for aggregation, i.e. { "segment_length": "sum"}.
- **spatial_bbox** (tuple[float, float, float, float]) – Spatial bounding box, (lon_min, lat_min, lon_max, lat_max), [deg]. By default, the entire globe is used.
- **spatial_grid_res** (float) – Spatial grid resolution, [deg]

Returns

xarray.Dataset – Aggregated variables in a longitude-latitude grid.

Examples

```
>>> rng = np.random.default_rng(234)
>>> vector = GeoVectorDataset(
...     longitude=rng.uniform(-10, 10, 10000),
...     latitude=rng.uniform(-10, 10, 10000),
...     altitude=np.zeros(10000),
...     time=np.zeros(10000).astype("datetime64[ns]"),
... )
>>> vector["foo"] = rng.uniform(0, 1, 10000)
>>> ds = vector.to_lon_lat_grid({"foo": "sum"}, spatial_bbox=(-10, -10, 9.5, 9.5))
>>> da = ds["foo"]
>>> da.coords
```

(continues on next page)

(continued from previous page)

```
Coordinates:
  * longitude  (longitude) float64 320B -10.0 -9.5 -9.0 -8.5 ... 8.0 8.5 9.0 9.5
  * latitude   (latitude) float64 320B -10.0 -9.5 -9.0 -8.5 ... 8.0 8.5 9.0 9.5
```

```
>>> da.values.round(2)
array([[2.23, 0.67, 1.29, ..., 4.66, 3.91, 1.93],
       [4.1 , 3.84, 1.34, ..., 3.24, 1.71, 4.55],
       [0.78, 3.25, 2.33, ..., 3.78, 2.93, 2.33],
       ...,
       [1.97, 3.02, 1.84, ..., 2.37, 3.87, 2.09],
       [3.74, 1.6 , 4.01, ..., 4.6 , 4.27, 3.4 ],
       [2.97, 0.12, 1.33, ..., 3.54, 0.74, 2.59]])
```

```
>>> da.sum().item() == vector["foo"].sum()
True
```

11.7 Utilities

utils.types	Convienence types.
utils.iteration	Utilites for iterating of sequences.
utils.temp	Temp utilities.
utils.json	JSON utilities.

11.7.1 pycontrails.utils.types

Convienence types.

Module Attributes

ArrayLike	Array like (np.ndarray, xr.DataArray)
ArrayOrFloat	Array or Float (np.ndarray, float)
ArrayScalarLike	Array like input (np.ndarray, xr.DataArray, np.float64, float)
DatetimeLike	Datetime like input (datetime, pd.Timestamp, np.datetime64)

Functions

apply_nan_mask_to_arraylike(arr, nan_mask)	Apply nan_mask to arr while maintaining the type.
support_arraylike(func)	Extend a numpy universal function operating on arrays of floats.
type_guard(obj, type_[, error_message])	Shortcut utility to type guard a variable with custom error message.

class pycontrails.utils.types.**ArrayLike**

Array like (np.ndarray, xr.DataArray)

alias of TypeVar('ArrayLike', ~numpy.ndarray, ~xarray.core.dataarray.DataArray, ~xarray.core.dataarray.DataArray | ~numpy.ndarray)

class pycontrails.utils.types.**ArrayOrFloat**

Array or Float (np.ndarray, float)

alias of TypeVar('ArrayOrFloat', ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy.float64]], float, float | ~numpy.ndarray[~typing.Any, ~numpy.dtype[~numpy.float64]])

class pycontrails.utils.types.**ArrayScalarLike**

Array like input (np.ndarray, xr.DataArray, np.float64, float)

alias of TypeVar('ArrayScalarLike', ~numpy.ndarray, ~xarray.core.dataarray.DataArray, ~numpy.float64, float, ~numpy.ndarray | float, ~xarray.core.dataarray.DataArray | ~numpy.ndarray)

class pycontrails.utils.types.**DatetimeLike**

Datetime like input (datetime, pd.Timestamp, np.datetime64)

alias of TypeVar('DatetimeLike', ~datetime.datetime, ~pandas._libs.tslibs.timestamps.Timestamp, ~numpy.datetime64, str)

pycontrails.utils.types.**apply_nan_mask_to_arraylike**(*arr*, *nan_mask*)

Apply nan_mask to arr while maintaining the type.

The parameter arr should have a float dtype.

This function is tested against xr.DataArray, pd.Series, and np.ndarray types.

Parameters

- **arr** (*ArrayLike*) – Array with np.float64 entries
- **nan_mask** (numpy.ndarray) – Boolean array of the same shape as arr

Returns

ArrayLike – Array arr with values in nan_mask set to np.nan. The arr is mutated in place if it is a np.ndarray. For xr.DataArray, a copy is returned.

Notes

When `arr` is a `xr.DataArray`, this function keeps any `attrs` from `arr` in the returned instance.

`pycontrails.utils.types.`**`support_arraylike`**(*func*)

Extend a numpy universal function operating on arrays of floats.

This decorator allows *func* to support any ArrayScalarLike parameter and keeps the return type consistent with the parameter.

Parameters
func (`Callable[[ArrayScalarLike], np.ndarray]`) – A numpy *ufunc* taking in a single array with *float*-like dtype. This decorator assumes *func* returns a numpy array.

Returns
`Callable[[ArrayScalarLike], ArrayScalarLike]` – Extended function.

See also:

-

`pycontrails.utils.types.`**`type_guard`**(*obj*, *type_*, *error_message=None*)

Shortcut utility to type guard a variable with custom error message.

Parameters

- **obj** (`Any`) – Any variable object
- **type_** (`Type[_Object]`) – Type of variable. Can be a tuple of types
- **error_message** (`str`, *optional*) – Custom error message

Returns
`_Object` – Returns the input object ensured to be `type_`

Raises
`ValueError` – Raises ValueError if `obj` is not `type_`

11.7.2 pycontrails.utils.iteration

Utilites for iterating of sequences.

Functions

chunk_list(lst, n)	Yield successive n-sized chunks from list.

`pycontrails.utils.iteration.`**`chunk_list`**(*lst*, *n*)

Yield successive n-sized chunks from list.

11.7.3 pycontrails.utils.temp

Temp utilities.

Functions

remove_tempfile(temp_filename)	Remove temp file.
temp_file()	Get context manager for temp file creation and cleanup.
temp_filename()	Get a filename in the host computers temp directory.

pycontrails.utils.temp.**remove_tempfile**(*temp_filename*)

Remove temp file.

Parameters
temp_filename (str) – Temp filename

pycontrails.utils.temp.**temp_file**()

Get context manager for temp file creation and cleanup.

pycontrails.utils.temp.**temp_filename**()

Get a filename in the host computers temp directory.

More robust than using tempfile.NamedTemporaryFile()

Returns
str – Temp filename

See also:

temp_file
Context manager for temp file creation and cleanup

11.7.4 pycontrails.utils.json

JSON utilities.

Functions

dataframe_to_geojson_points(df[, ...])	Convert a pandas DataFrame to a GeoJSON-like dictionary.

Classes

NumpyEncoder(*[, skipkeys, ensure_ascii, ...])	Custom JSONEncoder for numpy data types.

class pycontrails.utils.json.**NumpyEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: JSONEncoder

Custom JSONEncoder for numpy data types.

Examples

```
>>> import json
>>> import numpy as np
>>> from pycontrails.utils.json import NumpyEncoder
```

```
>>> data = np.array([0, 1, 2, 3])
>>> json.dumps(data, cls=NumpyEncoder)
'[0, 1, 2, 3]'
```

```
>>> data = np.datetime64(1234567890, "s")
>>> json.dumps(data, cls=NumpyEncoder)
'"2009-02-13T23:31:30"'
```

Notes

Adapted https://github.com/hmallen/numpyencoder/blob/master/numpyencoder/numpyencoder.py

default(*obj*)

Encode numpy data types.

This method overrides *default()* on the JSONEncoder class.

Parameters

obj (Any) – Object to encode.

Returns

Any – Encoded object.

pycontrails.utils.json.**dataframe_to_geojson_points**(*df*, *properties=None*, *filter_nan=False*)

Convert a pandas DataFrame to a GeoJSON-like dictionary.

This function create a Python representation of a GeoJSON FeatureCollection with Point features based on a pandas.DataFrame with geospatial coordinate columns.

Parameters

- **df** (pandas.DataFrame) – Base dataframe. Must contain geospatial coordinate columns ["longitude", "latitude", "altitude", "time"]
- **properties** (list[str], *optional*) – Specify columns to include feature properties. By default, will use all column data that is not in the coordinate columns.

- **filter_nan** (`bool | list[str]`, *optional*) – Filter out points with nan values in any columns, including coordinate columns. If *list of str* is input, only *filter_nan* columns will be used for filtering, allowing null values in the other columns.

Returns
`dict[str, Any]` – Description

Raises
`KeyError` – Raises if *properties* or *filter_nan* input contains a column label that does not exist in *df*

11.8 Extensions

11.8.1 BADA

Requires pycontrails-bada extension and data files obtained through BADA license. See *BADA Extension* for more information.

ext.bada.bada_model	BADA models along flight and on a regular grid.
ext.bada.BADAFlight([met, params])	Compute aircraft properties and fuel consumption.
ext.bada.BADAFlightParams([copy_source, ...])	`BADAFlight` model parameters.
ext.bada.BADAGrid([met, params])	Compute nominal BADA values for a large grid of independent points.
ext.bada.BADAGridParams([copy_source, ...])	`BADAGrid` model parameters.
ext.bada.BADA3([bada_path])	BADA 3.15 Support.
ext.bada.BADA4([bada_path])	BADA 4.2 Support.

pycontrails.ext.bada.bada_model

BADA models along flight and on a regular grid.

Functions

`bada_nominal_grid`(atyp_bada, bada, q_fuel[, ...])	Calculate nominal BADA-derived fuel flow and engine efficiency over grid.
`get_bada`(aircraft_type[, bada3_path, ...])	Check BADA databases for `aircraft_type`.

Classes

`BADAFlight`([met, params])	Compute aircraft properties and fuel consumption.
`BADAFlightParams`([copy_source, ...])	`BADAFlight` model parameters.
`BADAGrid`([met, params])	Compute nominal BADA values for a large grid of independent points.
`BADAGridParams`([copy_source, ...])	`BADAGrid` model parameters.
`BADAParams`([copy_source, ...])	Base parameters for BADA models.

pycontrails.ext.bada.BADAFlight

class pycontrails.ext.bada.**BADAFlight**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *AircraftPerformance*

Compute aircraft properties and fuel consumption.

Parameters

- **met** (MetDataset | None, *optional*) – Dataset containing "air_temperature", "eastward_wind", and "northward_wind" variables. Only used if these variables are not already found on parameter source in *eval()*. By default None.
- **params** (dict[str, Any], *optional*) – Override model parameters with dictionary. See *BADAFlightParams* for model parameters.
- ****params_kwargs** – Override model parameters with keyword arguments. See *BADAFlightParams* for model parameters.

See also:

-
meth:*eval*

-
class:*BADAFlightParams*

__init__(*met=None*, *params=None*, ***params_kwargs*)

Methods

__init__([met, params])	
calculate_aircraft_performance(*, ...)	Calculate aircraft performance along a trajectory.
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
ensure_true_airspeed_on_source()	Add true_airspeed field to *source* data if not already present.
eval([source])	Extract aircraft properties and calculate the fuel consumption.
get_bada(aircraft_type)	Check BADA databases for aircraft_type.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is type_.
set_source([source])	Attach original or copy of input source to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
simulate_fuel_and_performance(*, ...)	Calculate aircraft mass, fuel mass flow rate, and overall propulsion efficiency.
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to source.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Data evaluated in model
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
optional_met_variables	Optional meteorology variables
processed_met_variables	Set of required parameters if processing already complete on `met` input.

calculate_aircraft_performance(*, *aircraft_type*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *engine_efficiency*, *fuel_flow*, *thrust*, *q_fuel*, ***kwargs*)

Calculate aircraft performance along a trajectory.

When `time` is not None, this method should be used for a single flight trajectory. Waypoints are coupled via the `time` parameter.

This method computes the rate of climb and descent (ROCD) to determine flight phases: "cruise", "climb", and "descent". Performance metrics depend on this phase.

When `time` is None, this method can be used to simulate flight performance over an arbitrary sequence of flight waypoints by assuming nominal flight characteristics. In this case, each point is treated independently and all points are assumed to be in a "cruise" phase of the flight.

Parameters

- **aircraft_type** (`str`) – Used to query the underlying model database for aircraft engine parameters.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude at each waypoint, $[ft]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **time** (`npt.NDArray[np.datetime64] | None`) – Waypoint time in `np.datetime64` format. If None, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (`npt.NDArray[np.float64] | float | None`) – True airspeed for each waypoint, $[ms^{-1}]$. If None, a nominal value is used.
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – Aircraft mass for each waypoint, $[kg]$.
- **engine_efficiency** (`npt.NDArray[np.float64] | float | None`) – Override the engine efficiency at each waypoint.
- **fuel_flow** (`npt.NDArray[np.float64] | float | None`) – Override the fuel flow at each waypoint, $[kgs^{-1}]$.

- **thrust** (npt.NDArray[np.float64] | float | None) – Override the thrust setting at each waypoint, [:math: N].
- **q_fuel** (float) – Lower calorific value (LCV) of fuel, [$J\ kg_{fuel}^{-1}$].
- ****kwargs** (Any) – Additional keyword arguments to pass to the model.

Returns
AircraftPerformanceData – Derived performance metrics at each waypoint.

default_params

alias of *BADAFlightParams*

eval(*source=None*, ***params*)

Extract aircraft properties and calculate the fuel consumption.

Input source must contain a valid aircraft_type in its source.attrs["aircraft_type"].

If met data is not passed into instance constructor, parameter source must contain the following data variables. - true_airspeed - air_temperature

If the following variables are not provided in the flight attribute, the default aircraft-engine assignment and aircraft mass properties from BADA will be assumed: - engine_uid - max_takeoff_weight - operating_empty_weight - max_payload

If 'load_factor' (ranging between 0 and 1) is not provided in the flight attribute, the aircraft mass at the first waypoint will be set to the reference mass for the specific aircraft type from the BADA database.

This method extracts the following aircraft properties from the BADA database.

- aircraft_type_bada
- wingspan
- max_mach
- max_altitude
- engine_name
- n_engine

From these BADA derived quantities, each of the following is computed.

- fuel_flow: fuel mass flow rate, [kgs^{-1}]
- fuel_burn: total fuel burn between two waypoints, [kg]
- aircraft_mass
- engine_efficiency
- thrust

Parameters

- **source** (*Flight*) – Flight to evaluate
- ****params** (Any) – Overwrite model parameters before eval

Returns
Flight – Flight with additional properties and fuel consumption variables listed above.

get_bada(*aircraft_type*)

Check BADA databases for `aircraft_type`.

Parameters
aircraft_type (`str`) – Aircraft type to check for.

Returns
`BADA` – BADA database object.

long_name = 'Base of aircraft data flight model'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'),)

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

name = 'bada'

optional_met_variables = (MetVariable(short_name='u', standard_name='eastward_wind', long_name='Eastward Wind', level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m s-1', amip='ua', description='"Eastward" indicates a vector component which is positive when directed eastward (negative westward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3), units='m s**-1', amip='va', description='"Northward" indicates a vector component which is positive when directed northward (negative southward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'))**

Optional meteorology variables

params

Instantiated model parameters, in dictionary form

source

Data evaluated in model

pycontrails.ext.bada.BADAFlightParams

class pycontrails.ext.bada.**BADAFlightParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *correct_fuel_flow=True*, *n_iter=3*, *bada3_path=None*, *bada4_path=None*, *bada_priority=4*, *model_choice='total_energy_model'*)

Bases: BADAParams

BADAFlight model parameters.

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h'), numpy.timedelta64(0, 'h'))*, *correct_fuel_flow=True*, *n_iter=3*, *bada3_path=None*, *bada4_path=None*, *bada_priority=4*, *model_choice='total_energy_model'*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

bada3_path	Default paths to BADA3 and BADA4 data directories Setting path to None will use default path on BADA3 and BADA4 instances
bada4_path	
bada_priority	Default BADA database used to search for aircraft characteristics.
copy_source	Copy input source data on eval
correct_fuel_flow	Whether to correct fuel flow to ensure it remains within the operational limits of the aircraft type.
downselect_met	Downselect input MetDataset` to region around source.
interpolation_bounds_error	If True, points lying outside interpolation will raise an error
interpolation_fill_value	Used for outside interpolation value if interpolation_bounds_error is False
interpolation_localize	Experimental.
interpolation_method	Interpolation method.
interpolation_q_method	Experimental.
interpolation_use_indices	Experimental.
met_latitude_buffer	Met latitude buffer for input to Flight. downselect_met(), in WGS84 coordinates.
met_level_buffer	Met level buffer for input to Flight. downselect_met(), in $[hPa]$.
met_longitude_buffer	Met longitude buffer for input to Flight. downselect_met(), in WGS84 coordinates.
met_time_buffer	Met time buffer for input to Flight. downselect_met() Only applies when downselect_met is True.
model_choice	BADA3 and BADA4 fuel flow model choice
n_iter	The number of iterations used to calculate aircraft mass and fuel flow.
verify_met	Call _verify_met() on model instantiation.

pycontrails.ext.bada.BADAGrid

class pycontrails.ext.bada.**BADAGrid**(*met=None*, *params=None*, ***params_kwargs*)

Bases: *AircraftPerformanceGrid*

Compute nominal BADA values for a large grid of independent points.

This model automatically corrects engine efficiency values to ensure that they remain realistic by clipping to a nominal grid of BADA-derived values.

Parameters

- **met** (MetDataset | None, *optional*) – Dataset containing "air_temperature" variable. Only used if these variables are not already found on parameter source in *eval()*. By default None.
- **params** (dict[str, Any], *optional*) – Override model parameters with dictionary. See *BADAGridParams* for model parameters.

- ****params_kwargs** – Override model parameters with keyword arguments. See *BADAFlightParams* for model parameters.

See also:

-
meth:*eval*

-
class:*BADAGridParams*

__init__(*met=None*, *params=None*, ***params_kwargs*)

Methods

__init__([met, params])	
downselect_met()	Downselect *met* domain to the max/min bounds of *source*.
eval([source])	Extract aircraft properties and calculate the fuel consumption.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
require_met()	Ensure that *met* is a MetDataset.
require_source_type(type_)	Ensure that *source* is type_.
set_source([source])	Attach original or copy of input source to *source*.
set_source_met([optional, variable])	Ensure or interpolate each required *met_variables* on *source* .
transfer_met_source_attrs([source])	Transfer met source metadata from *met* to source.
update_params([params])	Update model parameters on *params*.

Attributes

params	Instantiated model parameters, in dictionary form
met	Meteorology data
source	Evaluated data source
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from *params*.
long_name	
met_required	Require meteorology is not None on __init__()
met_variables	Required meteorology pressure level variables.
name	
optional_met_variables	Optional meteorology variables
processed_met_variables	Set of required parameters if processing already complete on met input.

default_params

alias of *BADAGridParams*

eval(*source=None*, ***params*)

Extract aircraft properties and calculate the fuel consumption.

Parameters

- **source** (`GeoVectorDataset | None`, *optional*) – Vector dataset defining coordinates to evaluate model. If None, the coordinates of *met* are used as evaluation points.
- ****params** (`Any`) – Overwrite model parameters before eval

Returns

GeoVectorDataset –

Data with variables:

- ”engine_efficiency”
- ”true_airspeed”
- ”fuel_flow”
- ”thrust”
- ”aircraft_mass”

and attributes:

- ”aircraft_type”
- ”bada_model”
- ”aircraft_type_bada”
- ”wingspan”
- ”max_mach”
- ”max_altitude”
- ”engine_name”
- ”n_engine”

long_name = 'Base of aircraft data evaluated at arbitrary points'

met

Meteorology data

met_variables = (MetVariable(short_name='t', standard_name='air_temperature', long_name='Air Temperature', level_type='isobaricInhPa', ecmwf_id=130, grib1_id=11, grib2_id=(0, 0, 0), units='K', amip='ta', description='Air temperature is the bulk temperature of the air, not the surface (skin) temperature.'),)

Required meteorology pressure level variables. Each element in the list is a `MetVariable` or a `tuple[MetVariable]`. If element is a `tuple[MetVariable]`, the variable depends on the data source. Only one variable in the tuple is required.

name = 'bada-points'

optional_met_variables = (MetVariable(short_name='u', standard_name='eastward_wind', long_name='Eastward Wind', level_type='isobaricInhPa', ecmwf_id=131, grib1_id=33, grib2_id=(0, 2, 2), units='m s-1', amip='ua', description='"Eastward" indicates a vector component which is positive when directed eastward (negative westward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'), MetVariable(short_name='v', standard_name='northward_wind', long_name='Northward Wind', level_type='isobaricInhPa', ecmwf_id=132, grib1_id=34, grib2_id=(0, 2, 3), units='m s**-1', amip='va', description='"Northward" indicates a vector component which is positive when directed northward (negative southward). Wind is defined as a two-dimensional (horizontal) air velocity vector, with no vertical component.'))**

Optional meteorology variables

params

Instantiated model parameters, in dictionary form

source

Evaluated data source

pycontrails.ext.bada.BADAGridParams

class pycontrails.ext.bada.**BADAGridParams**(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h')*, *numpy.timedelta64(0, 'h'))*, *fuel=<factory>*, *aircraft_type='B737'*, *mach_number=None*, *aircraft_mass=None*, *correct_fuel_flow=True*, *n_iter=3*, *bada3_path=None*, *bada4_path=None*, *bada_priority=4*, *model_choice='total_energy_model'*, *engine_uid=None*)

Bases: BADAParams, *AircraftPerformanceGridParams*

BADAGrid model parameters.

See also:

BADAParams

__init__(*copy_source=True*, *interpolation_method='linear'*, *interpolation_bounds_error=False*, *interpolation_fill_value=nan*, *interpolation_localize=False*, *interpolation_use_indices=False*, *interpolation_q_method=None*, *verify_met=True*, *downselect_met=True*, *met_longitude_buffer=(0.0, 0.0)*, *met_latitude_buffer=(0.0, 0.0)*, *met_level_buffer=(0.0, 0.0)*, *met_time_buffer=(numpy.timedelta64(0, 'h')*, *numpy.timedelta64(0, 'h'))*, *fuel=<factory>*, *aircraft_type='B737'*, *mach_number=None*, *aircraft_mass=None*, *correct_fuel_flow=True*, *n_iter=3*, *bada3_path=None*, *bada4_path=None*, *bada_priority=4*, *model_choice='total_energy_model'*, *engine_uid=None*)

Methods

__init__([copy_source, ...])	
as_dict()	Convert object to dictionary.

Attributes

aircraft_mass	Aircraft mass, [kg] If None, a nominal value is determined by the implementation.
aircraft_type	ICAO code designating simulated aircraft type.
bada3_path	Default paths to BADA3 and BADA4 data directories Setting path to None will use default path on BADA3 and BADA4 instances
bada4_path	
bada_priority	Default BADA database used to search for aircraft characteristics.
copy_source	Copy input source data on eval
correct_fuel_flow	Whether to correct fuel flow to ensure it remains within the operational limits of the aircraft type.
downselect_met	Downselect input MetDataset` to region around source.
engine_uid	Engine unique identification number for the ICAO Aircraft Emissions Databank (EDB) If None, the assumed engine_uid from BADA is used.
interpolation_bounds_error	If True, points lying outside interpolation will raise an error
interpolation_fill_value	Used for outside interpolation value if interpolation_bounds_error is False
interpolation_localize	Experimental.
interpolation_method	Interpolation method.
interpolation_q_method	Experimental.
interpolation_use_indices	Experimental.
mach_number	Mach number, [Ma] If None, a nominal cruise value is determined by the implementation.
met_latitude_buffer	Met latitude buffer for input to Flight.downselect_met(), in WGS84 coordinates.
met_level_buffer	Met level buffer for input to Flight.downselect_met(), in [hPa].
met_longitude_buffer	Met longitude buffer for input to Flight.downselect_met(), in WGS84 coordinates.
met_time_buffer	Met time buffer for input to Flight.downselect_met() Only applies when downselect_met is True.
model_choice	BADA3 and BADA4 fuel flow model choice
n_iter	The number of iterations used to calculate aircraft mass and fuel flow.
verify_met	Call _verify_met() on model instantiation.
fuel	Fuel type

engine_uid = None

Engine unique identification number for the ICAO Aircraft Emissions Databank (EDB) If None, the assumed engine_uid from BADA is used.

pycontrails.ext.bada.BADA3

class pycontrails.ext.bada.**BADA3**(*bada_path=None*)

Bases: BADA

BADA 3.15 Support.

Base of Aircraft Data (BADA) provides a set of ASCII files containing performance and operating procedure coefficients for 294 different aircraft types. This class implements the "total energy model" as described in BADA User Manual.

Parameters

bada_path (str | pathlib.Path, *optional*) – Path to BADA 3.15 Model files. Default path set to /bada/bada3 in the parent directory of the pycontrails repository.

References

- Eurocontrol. User Manual for the Base of Aircraft Data (BADA) Revision 3.15. Vol EEC Techni. Eurocontrol Experimental Centre; 2019.

__init__(*bada_path=None*)

Methods

__init__([bada_path])	
calculate_aircraft_performance(*, ...)	Calculate aircraft performance along a trajectory.
check_aircraft_type_availability(aircraft_typ	Check if aircraft type designator is available in BADA database.
clip_fuel_flow_by_ptf_bounds(atyp_icao, ...)	Clip array of fuel flow by the BADA PTF-defined thresholds.
correct_fuel_flow(atyp_icao, fuel_flow, ...)	Correct unrealistic fuel mass flow rate by clipping to PTF nominals.
downselect_met()	Downselect `met` domain to the max/min bounds of `source`.
ensure_true_airspeed_on_source()	Add `true_airspeed` field to `source` data if not already present.
eval([source])	Evaluate the aircraft performance model.
get_aircraft_engine_properties(atyp_icao[, ...])	Extract the aircraft performance and engine properties from the BADA database.
get_aircraft_params(aircraft_type)	Get aircraft params associated to aircraft type.
get_ptf_params(aircraft_type)	Get PTF params associated to aircraft type.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
is_within_thrust_limits(*, atyp_bada, ...)	Determine whether thrust required at each waypoint is within bounds of BADA model.
nominal_cruising_speed(aircraft_type, alt_ft)	Compute nominal cruising speed at altitude by interpolating over PTF data.
nominal_fuel_flow(aircraft_type, alt_ft, phase)	Compute nominal fuel flow depending on *phase* based on PTF data.
nominal_fuel_flow_from_flight_phase(...)	Call `nominal_fuel_flow()` for each phase according to `flight_phase`.
nominal_roc(aircraft_type, alt_ft)	Compute nominal rate of climb at altitude by interpolating over PTF data.
nominal_rod(aircraft_type, alt_ft)	Compute nominal rate of descent at altitude by interpolating over PTF data.
require_met()	Ensure that `met` is a MetDataset.
require_source_type(type_)	Ensure that `source` is `type_`.
set_source([source])	Attach original or copy of input `source` to `source`.
set_source_met([optional, variable])	Ensure or interpolate each required `met_variables` on `source` .
simulate_fuel_and_performance(*, ...)	Calculate aircraft mass, fuel mass flow rate, and overall propulsion efficiency.
transfer_met_source_attrs([source])	Transfer met source metadata from `met` to `source`.
update_params([params])	Update model parameters on `params`.

Attributes

path	Path to BADA data directory
synonym_dict	Aircraft type synonyms
aircraft_engine_dataframe	Available/assumed aircraft-engine combinations
ptf_params_dict	
aircraft_param_dict	Engine and aircraft properties common to BADA3 and BADA4
default_path	Default path to BADA data directories
version	BADA version.
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from `params`.
long_name	
met	Meteorology data
met_required	Require meteorology is not None on __init__()
name	
params	Instantiated model parameters, in dictionary form
source	Data evaluated in model
ptf_param_dict	Coefficients and properties extracted from BADA3 and BADA4 PTF files
met_variables	Required meteorology pressure level variables.
processed_met_variables	Set of required parameters if processing already complete on `met` input.
optional_met_variables	Optional meteorology variables

aircraft_engine_dataframe

Available/assumed aircraft-engine combinations

aircraft_param_dict

Engine and aircraft properties common to BADA3 and BADA4

calculate_aircraft_performance(*, *aircraft_type*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *engine_efficiency*, *fuel_flow*, *thrust*, *q_fuel*, ***kwargs*)

Calculate aircraft performance along a trajectory.

When `time` is not None, this method should be used for a single flight trajectory. Waypoints are coupled via the `time` parameter.

This method computes the rate of climb and descent (ROCD) to determine flight phases: "cruise", "climb", and "descent". Performance metrics depend on this phase.

When `time` is None, this method can be used to simulate flight performance over an arbitrary sequence of flight waypoints by assuming nominal flight characteristics. In this case, each point is treated independently and all points are assumed to be in a "cruise" phase of the flight.

Parameters

- **aircraft_type** (`str`) – Used to query the underlying model database for aircraft engine parameters.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude at each waypoint, $[ft]$

- **air_temperature** (npt.NDArray[np.float64]) – Ambient temperature for each waypoint, [K]
- **time** (npt.NDArray[np.datetime64] | None) – Waypoint time in np.datetime64 format. If None, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (npt.NDArray[np.float64] | float | None) – True airspeed for each waypoint, [ms^{-1}]. If None, a nominal value is used.
- **aircraft_mass** (npt.NDArray[np.float64] | float) – Aircraft mass for each waypoint, [kg].
- **engine_efficiency** (npt.NDArray[np.float64] | float | None) – Override the engine efficiency at each waypoint.
- **fuel_flow** (npt.NDArray[np.float64] | float | None) – Override the fuel flow at each waypoint, [kgs^{-1}].
- **thrust** (npt.NDArray[np.float64] | float | None) – Override the thrust setting at each waypoint, [:math: N].
- **q_fuel** (float) – Lower calorific value (LCV) of fuel, [$J\ kg_{fuel}^{-1}$].
- ****kwargs** (Any) – Additional keyword arguments to pass to the model.

Returns

AircraftPerformanceData – Derived performance metrics at each waypoint.

default_path

Default path to BADA data directories

get_aircraft_engine_properties(*atyp_icao*, *engine_uid=None*)

Extract the aircraft performance and engine properties from the BADA database.

Parameters

- **atyp_icao** (str) – ICAO aircraft type designator.
- **engine_uid** (str) – Engine unique identification number from the ICAO EDB. If None is provided or engine_uid is unidentified, default aircraft-engine combination from BADA will be used. This parameter is unused for *BADA3*; it is only considered for *BADA4*.

Returns

AircraftProperties

is_within_thrust_limits(*, *atyp_bada*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *thrust*, *flight_phase*)

Determine whether thrust required at each waypoint is within bounds of BADA model.

If thrust is not provided as input, it will be computed according to BADA standards. Thrust limits are only defined for the BADA3 model. For BADA4, this function will raise a NotImplementedError.

Parameters

- **atyp_bada** (str) – ICAO aircraft type designator (BADA 3), or long aircraft type designator (BADA 4).
- **altitude_ft** (npt.NDArray[np.float_]) – Altitude at each waypoint, [ft]
- **air_temperature** (npt.NDArray[np.float_]) – Ambient temperature for each waypoint, [K]

- **time** (npt.NDArray[np.datetime64] | None) – Waypoint time in *np.datetime64* format. If *None*, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (npt.NDArray[np.float_] | float | None) – True airspeed for each waypoint, [ms^{-1}]. If *None*, the nominal BADA cruise value is used.
- **aircraft_mass** (npt.NDArray[np.float_] | float | None) – Aircraft mass for each waypoint, [kg]. If *None*, the nominal BADA value is used.
- **thrust** (npt.NDArray[np.float_] | float | None) – Override the thrust setting at each waypoint, [:math: N].
- **flight_phase** (npt.NDArray[np.uint8] | flight.FlightPhase | None) – Flight phase for each waypoint. If *None*, the flight phase is assumed to be cruise.

Returns

npt.NDArray[np.bool_] – Boolean array telling whether the thrust at each waypoint is within BADA trust limits

long_name = 'Base of Aircraft Data (BADA) Revision 3.15'

name = 'BADA3'

path

Path to BADA data directory

ptf_params_dict

synonym_dict

Aircraft type synonyms

version

BADA version. Currently only used on BADA3 class.

pycontrails.ext.bada.BADA4

class pycontrails.ext.bada.**BADA4**(*bada_path=None*)

Bases: BADA

BADA 4.2 Support.

Base of Aircraft Data (BADA) provides a set of ASCII files containing performance and operating procedure coefficients.`

Parameters

bada_path (str | pathlib.Path, *optional*) – Path to BADA 4.2 Model files Default path set to /bada/bada4 in the parent directory of the pycontrails repository.

References

- **Eurocontrol. User Manual for the Base of Aircraft Data (BADA) Family 4.**
  Vol EEC Techni. Eurocontrol Experimental Centre; 2016.

__init__(*bada_path=None*)

Methods

__init__([bada_path])	
assumed_aircraft_engine_type_bada4(...)	Get the assumed aircraft type for BADA4.
calculate_aircraft_performance(*, ...)	Calculate aircraft performance along a trajectory.
check_aircraft_type_availability(aircraft_typ	Check if aircraft type designator is available in BADA database.
clip_fuel_flow_by_ptf_bounds(atyp_icao, ...)	Clip array of fuel flow by the BADA PTF-defined thresholds.
correct_fuel_flow(atyp_icao, fuel_flow, ...)	Correct unrealistic fuel mass flow rate by clipping to PTF nominals.
downselect_met()	Downselect `met` domain to the max/min bounds of `source`.
ensure_true_airspeed_on_source()	Add `true_airspeed` field to `source` data if not already present.
eval([source])	Evaluate the aircraft performance model.
get_aircraft_engine_properties(atyp_icao[, ...])	Extract the aircraft performance and engine properties from the BADA database.
get_aircraft_params(aircraft_type)	Get aircraft params associated to aircraft type.
get_ptf_params(aircraft_type)	Get PTF params associated to aircraft type.
get_source_param(key[, default, set_attr])	Get source data with default set by parameter key.
is_within_thrust_limits(*, atyp_bada, ...)	Determine whether thrust required at each waypoint is within bounds of BADA model.
nominal_cruising_speed(aircraft_type, alt_ft)	Compute nominal cruising speed at altitude by interpolating over PTF data.
nominal_fuel_flow(aircraft_type, alt_ft, phase)	Compute nominal fuel flow depending on *phase* based on PTF data.
nominal_fuel_flow_from_flight_phase(...)	Call `nominal_fuel_flow()` for each phase according to `flight_phase`.
nominal_roc(aircraft_type, alt_ft)	Compute nominal rate of climb at altitude by interpolating over PTF data.
nominal_rod(aircraft_type, alt_ft)	Compute nominal rate of descent at altitude by interpolating over PTF data.
require_met()	Ensure that `met` is a MetDataset.
require_source_type(type_)	Ensure that `source` is `type_`.
set_source([source])	Attach original or copy of input `source` to `source`.
set_source_met([optional, variable])	Ensure or interpolate each required `met_variables` on `source` .
simulate_fuel_and_performance(*, ...)	Calculate aircraft mass, fuel mass flow rate, and overall propulsion efficiency.
transfer_met_source_attrs([source])	Transfer met source metadata from `met` to `source`.
update_params([params])	Update model parameters on `params`.

Attributes

path	Path to BADA data directory
synonym_dict	Aircraft type synonyms
aircraft_engine_dataframe	Available/assumed aircraft-engine combinations
ptf_params_dict	
aircraft_param_dict	Engine and aircraft properties common to BADA3 and BADA4
default_path	Default path to BADA data directories
version	BADA version.
hash	Generate a unique hash for model instance.
interp_kwargs	Shortcut to create interpolation arguments from `params`.
long_name	
met	Meteorology data
met_required	Require meteorology is not None on __init__()
name	
params	Instantiated model parameters, in dictionary form
source	Data evaluated in model
aircraft_engine_options	
ptf_param_dict	Coefficients and properties extracted from BADA3 and BADA4 PTF files
met_variables	Required meteorology pressure level variables.
processed_met_variables	Set of required parameters if processing already complete on `met` input.
optional_met_variables	Optional meteorology variables

aircraft_engine_dataframe

Available/assumed aircraft-engine combinations

aircraft_engine_options

aircraft_param_dict

Engine and aircraft properties common to BADA3 and BADA4

assumed_aircraft_engine_type_bada4(*atyp_icao*, *engine_uid*)

Get the assumed aircraft type for BADA4.

Parameters

- **atyp_icao** (`str`) – ICAO aircraft type designator
- **engine_uid** (`None | str`) – Engine unique identification number from the ICAO EDB. If None is provided or engine_uid is unidentified, the default aircraft-engine combination from BADA4 is used.

Returns

`pandas.Series` – Assumed aircraft and engine type

calculate_aircraft_performance(*, *aircraft_type*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *engine_efficiency*, *fuel_flow*, *thrust*, *q_fuel*, ***kwargs*)

Calculate aircraft performance along a trajectory.

When `time` is not None, this method should be used for a single flight trajectory. Waypoints are coupled via the `time` parameter.

This method computes the rate of climb and descent (ROCD) to determine flight phases: "cruise", "climb", and "descent". Performance metrics depend on this phase.

When `time` is None, this method can be used to simulate flight performance over an arbitrary sequence of flight waypoints by assuming nominal flight characteristics. In this case, each point is treated independently and all points are assumed to be in a "cruise" phase of the flight.

Parameters

- **aircraft_type** (`str`) – Used to query the underlying model database for aircraft engine parameters.
- **altitude_ft** (`npt.NDArray[np.float64]`) – Altitude at each waypoint, $[ft]$
- **air_temperature** (`npt.NDArray[np.float64]`) – Ambient temperature for each waypoint, $[K]$
- **time** (`npt.NDArray[np.datetime64] | None`) – Waypoint time in `np.datetime64` format. If None, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (`npt.NDArray[np.float64] | float | None`) – True airspeed for each waypoint, $[ms^{-1}]$. If None, a nominal value is used.
- **aircraft_mass** (`npt.NDArray[np.float64] | float`) – Aircraft mass for each waypoint, $[kg]$.
- **engine_efficiency** (`npt.NDArray[np.float64] | float | None`) – Override the engine efficiency at each waypoint.
- **fuel_flow** (`npt.NDArray[np.float64] | float | None`) – Override the fuel flow at each waypoint, $[kgs^{-1}]$.
- **thrust** (`npt.NDArray[np.float64] | float | None`) – Override the thrust setting at each waypoint, [:math: N].
- **q_fuel** (`float`) – Lower calorific value (LCV) of fuel, $[J\ kg_{fuel}^{-1}]$.
- ****kwargs** (`Any`) – Additional keyword arguments to pass to the model.

Returns

`AircraftPerformanceData` – Derived performance metrics at each waypoint.

`default_path`

Default path to BADA data directories

`get_aircraft_engine_properties`(*atyp_icao*, *engine_uid=None*)

Extract the aircraft performance and engine properties from the BADA database.

Parameters

- **atyp_icao** (`str`) – ICAO aircraft type designator.
- **engine_uid** (`str`) – Engine unique identification number from the ICAO EDB. If None is provided or engine_uid is unidentified, default aircraft-engine combination from BADA will be used. This parameter is unused for *BADA3*; it is only considered for *BADA4*.

Returns
AircraftProperties

is_within_thrust_limits(*, *atyp_bada*, *altitude_ft*, *air_temperature*, *time*, *true_airspeed*, *aircraft_mass*, *thrust*, *flight_phase*)

Determine whether thrust required at each waypoint is within bounds of BADA model.

If thrust is not provided as input, it will be computed according to BADA standards. Thrust limits are only defined for the BADA3 model. For BADA4, this function will raise a NotImplementedError.

Parameters

- **atyp_bada** (str) – ICAO aircraft type designator (BADA 3), or long aircraft type designator (BADA 4).
- **altitude_ft** (npt.NDArray[np.float_]) – Altitude at each waypoint, [ft]
- **air_temperature** (npt.NDArray[np.float_]) – Ambient temperature for each waypoint, [K]
- **time** (npt.NDArray[np.datetime64] | None) – Waypoint time in *np.datetime64* format. If *None*, only drag force will is used in thrust calculations (ie, no vertical change and constant horizontal change). In addition, aircraft is assumed to be in cruise.
- **true_airspeed** (npt.NDArray[np.float_] | float | None) – True airspeed for each waypoint, [ms^{-1}]. If *None*, the nominal BADA cruise value is used.
- **aircraft_mass** (npt.NDArray[np.float_] | float | None) – Aircraft mass for each waypoint, [kg]. If *None*, the nominal BADA value is used.
- **thrust** (npt.NDArray[np.float_] | float | None) – Override the thrust setting at each waypoint, [:math: N].
- **flight_phase** (npt.NDArray[np.uint8] | flight.FlightPhase | None) – Flight phase for each waypoint. If *None*, the flight phase is assumed to be cruise.

Returns
npt.NDArray[np.bool_] – Boolean array telling whether the thrust at each waypoint is within BADA trust limits

long_name = 'Base of Aircraft Data (BADA) Revision 4.2'

name = 'BADA4'

path

Path to BADA data directory

ptf_params_dict

synonym_dict

Aircraft type synonyms

version

BADA version. Currently only used on BADA3 class.

CHAPTER

TWELVE

LITERATURE

12.1 Publications

Publications including `pycontrails`:

- [Teoh *et al.*, 2022]
- [Teoh *et al.*, 2022]
- [Chevallier *et al.*, 2023]
- [Frías *et al.*, 2023]
- [Geraedts *et al.*, 2023]

If you use `pycontrails` in your work, please cite using the Zenodo DOI:

DOI 10.5281/zenodo.11263606

12.2 References

References managed in public Zotero Library

CHAPTER

THIRTEEN

CONTRIBUTING

Contributions (bug reports, fixes, documentation, enhancements, ideas, ...) are welcome and appreciated.

To get started, find the best path for your contribution:

- Ask questions, discuss models, and present ideas in Discussions.
- Report bugs or suggest changes as Issues.
- Contribute fixes or improvements as Pull Requests.

Please follow the Github Community Guidelines when participating in any of these forums.

The following emulates the xarray contributing guidelines.

13.1 Contributing to documentation

Documentation is written in reStructuredText and synthesized with Sphinx.

For small changes, fork and edit files directly in the Github interface.

For larger changes:

- Set up a local development environment.
- Edit documents and notebooks following existing conventions.
- Build and review the documentation locally:

```
# docs build to directory docs/_build/html
$ make docs-build
```

- Submit changes as a Pull Request.

13.2 Contributing to the code base

If you are new to development, see xarray's Working with the code. This reference provides an introduction to version control, git, Github, Forking, and creating branches.

For more involved changes, create a Github Issue describing the intended changes first.

Once you're ready to develop:

- Set up a local development environment.
- Implement updates. Make sure code is documented using existing conventions.

- Ensure tests pass locally:

```
$ make test
```

- Submit changes as a Pull Request.

CHAPTER

FOURTEEN

CHANGELOG

14.1 v0.51.0

14.1.1 Breaking changes

- Geodesic interpolation is now used in `Flight.resample_and_fill` when the great circle distance between waypoints (rather than the total segment length including vertical displacement) exceeds a threshold. This may change the interpolation method used when resampling flight segments with lengths close to the geodesic interpolation threshold.
- Fixed typo in `thermo.c_pm` will decrease computed values of moist heat capacity with non-zero specific humidity. We expect the downstream impact on contrail predictions by `ISSR`, `SAC`, `PCR`, and `Cocip` models to be minimal.
- `np.nan` is now used as the default `fill_value` in `MetDataArray.to_polygon_feature` and `MetDataArray.to_polygon_feature_collection`. This ensures that NaN values are never included in polygon interiors unless a non-NaN `fill_value` is explicitly passed as a keyword argument.

14.1.2 Features

- Add `ERA5ModelLevel` and `HRESModelLevel` interfaces for accessing ERA5 and HRES data on model levels.
- Update ECMWF tutorial notebook with instructions for using model-level datalibs.
- Add `HistogramMatching` humidity scaling calibrated for model-level ERA5 data.
- Modify `polygon.find_multipolygon`, `MetDataArray.to_polygon_feature`, `MetDataArray.to_polygon_feature_collection`, and `MetDataArray.to_polyhedra` to permit finding regions with values above or below a threshold.

14.1.3 Fixes

- Use horizontal great circle distance to determine whether geodesic interpolation is used in `Flight.resample_and_fill`. This ensures geodesic interpolation is used between sufficiently distant waypoints even when one or both waypoints contain NaN altitude data.
- Fix typo in moist heat capacity equation `thermo.c_pm`.

14.1.4 Internals

- Create static copy of dataframe for determining pressure at ECMWF model levels.
- Extract model-level utilities in `ARCOERA5` to their own module for reuse in `ERA5ModelLevel` and `HRESModelLevel`.
- Update Makefile so `make ensure-era5-cached` uses default cache directory when run locally.
- Bump pinned black and ruff versions.
- Disable mypy type checking on `functools.wraps` in `support_arraylike` decorator to avoid error that appears starting on mypy 1.10.0.
- Update pinned `Cocip` test output values after moist heat capacity bugfix.
- Add static files with ECMWF model levels and model-level ERA5 RHI quantiles to packaged data.
- Pass `exc_type=ImportError` to `pytest.importorskip` in test fixtures that use pycontrails extensions to suppress pytest warning when extensions are not installed.
- Bump minimum pytest version 8.2 to ensure the `exc_type` kwarg is available.

14.2 v0.50.2

14.2.1 Breaking changes

- Replaces engine-uid `01P10IA024` with `04P10IA027` (superseded in the IACO EDB).

14.2.2 Features

- Adds support for the `E190` aircraft type in the PS model
- Adds `Flight.distance_to_coords` which takes in a distance along a flight trajectory in meters and returns geodesic coordinates.
- Adds methods to `ps_operational_limits` to find the maximum and minimum mach numbers for a given set of operating conditions.
- Updates ICAO aircraft engine emissions databank (EDB) from v28c to v29b.

14.2.3 Internals

- New data for gaseous and nvPM emissions for PW812D, PW812GA, RR Trent 7000 with improved nvPM combustor
- Update nvPM emissions for IAE V2530-A5, PW1500G and PW1900G
- Update PS model coeffiencts to match the latest version provided by Ian Poll

14.3 v0.50.1

14.3.1 Breaking changes

- Updates to flight resampling logic now ensure that resampled waypoints include any and all times between flight start and end times that are a multiple of the resampling frequency. This may add an additional waypoint to some flights after resampling, and may result in `Flight.resample_and_fill` returning a flight with a single waypoint rather than an empty flight.

14.3.2 Features

- Refine CoCiP contrail initialization model based on the work of Unterstrasser (2016, doi:10.5194/acp-16-2059-2016) and Karcher (2018, doi:10.1038/s41467-018-04068-0).
  - This update implements a refined parameterization of the survival fraction of contrail ice crystal number after the wake vortex phase (`f_surv`). The parameterised model was developed by Unterstrasser (2016) based on outputs provided by large eddy simulations, and improves agreement with LES output relative to the default survival fraction parameterization in CoCiP.
  - These changes replicate Fig. 4 of Karcher (2018), where `f_surv` now depends on the initial number of ice crystals. These effects are particularly important, especially in the "soot-poor" scenario where the number fraction of contrail ice crystals that survives the wake vortex phase could be larger than the mass fraction, because the particles are larger in size.
  - This also improves upon the existing assumption in CoCiP, where the survival fraction is estimated as the change in contrail ice water content (by mass) before and after the wake vortex phase. The Unterstrasser (2016) parameterization can be used in CoCiP by setting a new parameter, `unterstrasser_ice_survival_fraction`, to `True`.
- Adds optional ATR20 to CoCiPGrid model.

14.3.3 Fixes

- Update flight resampling logic to align with expected behavior for very short flights, which is now detailed in the `Flight.resample_and_fill` docstring.

14.3.4 Internals

- Adds a parameter to `CoCipParams`, `unterstrasser_ice_survival_fraction`, that activates the Unterstrasser (2016) survival parameterization when set to `True`. This is disabled by default, and only implemented for `CoCiP`. `CoCiPGrid` will produce an error if run with `unterstrasser_ice_surival_fraction=True`.
- Modifies `CoCiPGrid` so that setting `compute_atr_20` (defined in `CoCipParams`) to `True` adds `global_yearly_mean_rf` and `atr20` to CoCiP-grid output.
- Replaces `pycontrails.datalib.GOES` ash convention label "MIT" with "SEVIRI"
- Modifies meteorology time step selection logic in `CoCiPGrid` to reduce duplicate chunk downloads when reading from remote zarr stores.
- Updates unit tests for xarray v2024.03.0, which introduced changes to netCDF decoding that slightly alter decoded values. Note that some unit tests will fail for earlier xarray versions.
- Updates `RegularGridInterpolator` to fall back on legacy scipy implementations of tensor-product spline methods when using scipy versions 1.13.0 and later.

14.4 v0.50.0

14.4.1 Features

- Add `ARCOERA5` interface for accessing ARCO ERA5 model level data. This interface requires the metview python package.
- Add ARCO ERA5 tutorial notebook highlighting the new interface.
- Add support to output contrail warming impact in ATR20

14.4.2 Breaking changes

- Reduce `CocipParams.met_level_buffer` from `(200, 200)` to `(40, 40)`. This change is motivated by the observation that the previous buffer was unnecessarily large and caused additional memory overhead. The new buffer is more in line with the typical vertical advection path of a contrail.

14.4.3 Fixes

- Raise ValueError when `list[Flight]` source is provided to `Cocip` and the `copy_source` parameter is set to `False`. Previously the source was copied in this case regardless of the `copy_source` parameter.
- Fix broken link in the model level notebook.

14.4.4 Internals

- The `datalib.parse_pressure_levels` now sorts the pressure levels in ascending order and raises a ValueError if the input pressure levels are duplicated or have mixed signs.
- Add new `MetDataSource.is_single_level` property.
- Add `ecmwf.Divergence` (a subclass of `MetVariable`) for accessing ERA5 divergence data.
- Update the specific humidity interpolation notebook to use the new `ARCOERA5` interface.
- Adds two parameters to `CoCipParams`, `compute_atr20` and `global_rf_to_atr20_factor`. Setting the former to `True` will add both `global_yearly_mean_rf` and `atr20` to the CoCiP output.
- Bump minimum pytest version to 8.1 to avoid failures in release workflow.

14.5 v0.49.5

14.5.1 Fixes

- Fix bug in which `Cocip._process_rad` dropped radiation dataset attributes introduced in v0.49.4.

14.6 v0.49.4

14.6.1 Breaking changes

- Remove the `CocipGridParams.met_slice_dt` parameter. Now met downselection is handled automatically during contrail evolution. When the `met` and `rad` data passed into `CocipGrid` are not already loaded into memory, this update may make `CocipGrid` slightly more performant.
- No longer explicitly load `met` and `rad` time slices into memory in `CocipGrid`. This now only occurs downstream when interpolation is performed. This change better aligns `CocipGrid` with other pycontrails models.
- Remove the `cocipgrid.cocip_time_handling` module. Any useful tooling has been moved directly to the `cocipgrid.cocip_grid` module.
- Remove the `CocipGrid.timedict` attribute. Add a `CocipGrid.timesteps` attribute. This is now applied in the same manner that the `Cocip` model uses its `timesteps` attribute.
- Simplify the runtime estimate used in constructing the `CocipGrid` `tqdm` progress bar. The new estimate is less precise than the previous estimate and should not be trusted for long-running simulations.
- Deprecate `MetBase.variables` in favor of `MetBase.indexes`.

14.6.2 Features

- Add support for 9 additional aircraft types in the Poll-Schumann (PS) aircraft performance model. The new aircraft types are:
  - A338
  - A339
  - A35K
  - B37M
  - B38M
  - B39M
  - B78X
  - BCS1
  - BCS3
- Modify PS coefficients for B788, B789, and A359.
- Support running `CocipGrid` on meteorology data without a uniformly-spaced time dimension. The `CocipGrid` implementation now no longer assumes `met["time"].diff()` is constant.
- Add a `MetDataset.downselect_met` method. This performs a met downselection in analogy with `GeoVectorDataset.downselect_met`.

14.6.3 Fixes

- Improve clarity of warnings produced when meteorology data doesn't cover the time range required by a gridded CoCiP model.
- No longer emit `pandas` warning when `Flight.resample_and_fill(..., drop=True, ...)` is called with non-float data.
- Correctly handle `CocipGrid` `rad` data with non-uniform time steps.

14.7 v0.49.3

14.7.1 Features

- Re-organize notebooks in documentation.
- Add new model level tutorial notebook.
- Add new high-level `Flight.clean_and_resample` method. This method parallels the `Flight.resample_and_fill` method but performs additional altitude filtering. In essence, this method is a combination of `Flight.filter_altitude` and `Flight.resample_and_fill`.

14.7.2 Breaking changes

- Remove `Flight.fit_altitude` method in favor of `Flight.filter_altitude`. The new method now only applies a median filter during cruise flight phase.

14.7.3 Fixes

- Remove opaque warning issued when all tau_contrail values are nan in `Cocip` evolution.
- Emit warning in `Cocip.eval` if the advected contrail is blown outside of the domain of the met data.
- Remove empty flights in `Fleet.from_seq`. Issue warning if an empty flight is encountered.
- Emit warning when `Flight.resample_and_fill` returns an empty flight.

14.7.4 Internals

- Modify test workflow to use Makefile recipes and ensure early failures are detected in CI.
- Pin `black` and `ruff` versions for consistency between local and CI/CD environments.
- Improve development documentation.
- Improve handling of missing credentials in tests (`make nb-test`, `make doctest`).
- Update time frequency aliases for `pandas` 2.2 compatibility.
- Update cython annotations for `scipy` 1.12 compatibility.
- Improve notebook output testing capabilities (`make nb-test`).
- Add new convenience Make recipe to execute all notebooks in the docs (`make nb-execute`).
- Add new Make recipe to cleanup notebooks (`make nb-clean`).

- Add pre-commit hook to check if notebooks are *clean*.
- Re-organize notebooks in documentation.
- Clean up contributing and develop documentation.
- Automatically parse `np.timedelta64`-like model params in `Model.update_params`.

14.8 v0.49.2

14.8.1 Features

- Support pandas Copy-on-Write. This can be enabled with `pd.set_option("mode.copy_on_write", True)` or by setting the `PANDAS_COPY_ON_WRITE` environment variable.

14.8.2 Fixes

- Ensure the `Flight.fuel` attribute is preserved for the `Flight.filter` method.
- Ensure the `Fleet.fl_attrs` attribute is preserved for the `Fleet.filter` method.
- Raise `ValueError` when `Flight.sort` or `Fleet.sort` is called. Both of these subclasses assume a custom sorting order that is enforced in their constructors.
- Always correct intermediate thrust coefficients computed in the `PSGrid` model. This correction is already enabled by default in the `PSFlight` model.
- Include `attr` fields in the ValueError message raised in `CocipGrid` when not all aircraft performance variables are present on the `source` parameter.
- Allow `mach_number` as a replacement for `true_airspeed` in `CocipGrid` aircraft performance determination.

14.8.3 Internals

- Make `Fuel` and its subclasses `JetA`, `SAFBlend`, and `HydrogenFuel` frozen.
- No longer copy `met` when `Models.downselect_met` is called. In some modes of operation, this reduces the memory footprint of the model.
- Update codebase for more harmony with PDEP 8 and Copy-on-Write semantics.
- Add `default` parameter to the `VectorDataset.get_data_or_attr` method.

14.9 v0.49.1

14.9.1 Fixes

- Fix memory bottleneck in `CocipGrid` simulation by avoiding expensive call to `pd.concat`.
- Require `oldest-supported-numpy` for python 3.12. Remove logic for `numpy` 1.26.0rc1 in the `pyproject.toml` build system.

14.10 v0.49.0

This release updates the Poll-Schumann (PS) aircraft performance model to version 2.0. It also includes a number of bug fixes and internal improvements.

14.10.1 Features

- Add convenience `Fleet.resample_and_fill`.
- Update the PS model aircraft-engine parameters.
- Improve PS model accuracy in fuel consumption estimates.
- Improve PS model accuracy in overall propulsive efficiency estimates.
- Include additional guardrails in the PS model to constrain the operational limits of different aircraft-engine types, i.e., the maximum permitted Mach number by altitude, maximum available thrust coefficient, maximum lift coefficient, and maximum allowable aircraft mass.

14.10.2 Fixes

- Update polygon algorithm to use `shapely.Polygon` instead of `shapely.LinearRing` for contours with at least 4 vertices.
- Fix `Fleet.to_flight_list` to avoid duplicating global attributes on the child `Flight` instances.
- Add `__slots__` to `GeoVectorDataset`, `Flight`, and `Fleet`. The base `VectorDataset` class already uses `__slots__`.
- Add `Fleet.copy` method.
- Improve `Fleet.__init__` implementation.
- Ensure `source` parameter is mutated in `CocipGrid.eval` when the model parameter `copy_source=False`.

14.11 v0.48.1

14.11.1 Features

- Generalize `met.shift_longitude()` to translate longitude coordinates onto any domain bounds.
- Add `VectorDataset.to_dict()` methods to output Vector data as dictionary. This method enables `Flight.to_dict()` objects to be serialized for input to the Contrails API.
- Add `VectorDataset.from_dict()` class method to create `VectorDataset` class from dictionary.
- Support more time formats, including timezone aware times, in `VectorDataset` creation. All timezone aware ``"time"`` coordinates are converted to UTC and stripped of timezone identifier.

14.11.2 Fixes

- Fix issue in the `wake_vortex.max_downward_displacement` function in which float32 dtypes were promoted to float64 dtypes in certain cases.
- Ignore empty vectors in `VectorDataset.sum`.

14.11.3 Internals

- Set `frozen=True` on the `MetVariable` dataclass.
- Test against python 3.12 in the GitHub Actions CI. Use python 3.12 the docs and doctest workflows.

14.12 v0.48.0

This release includes a number of breaking changes and new features. If upgrading from a previous version of `pycontrails`, please read the changelog carefully. Open an issue if you experience problems.

14.12.1 Breaking changes

- When running `Cocip` and other `pycontrails` models, the `met` and `rad` parameter must now contain predefined metadata attributes `provider`, `dataset`, and `product` describing the met source. An error will now be raised in `Cocip` if these attributes are not present.
- Deprecate passing arbitrary `kwargs` into the `MetDataArray` constructor.
- No longer convert accumulated radiation data to average instantaneous data in `ERA5` and `HRES` interfaces. This logic is now handled downstream by the model (e.g., `Cocip`). This change allows for more flexibility in the `rad` data passed into the model and avoids unnecessary computation in the `MetDataSource` interfaces.
- Add new `MetDataSource.set_met_source_metadata` abstract method. This should be called within the implementing class `open_metdataset` method.
- No longer take a finite difference in the time dimension for HRES radiation data. This is now also handled natively in `Cocip`.
- No longer convert relative humidity from a percentage to a fraction in `ERA5` and `HRES` interfaces.
- Require the `HRES` `stream` parameter to be one of `["oper", "enfo"]`. Require the `field_type` parameter to be one of `["fc", "pf", "cf", "an"]`.
- Remove the `steps` and `step_offset` properties in the `GFSForecast` interface. Now the `timesteps` attribute is the only source of truth for determining AWS S3 keys. Change the `filename` method to take in a `datatime` timestep instead of an `int` step. No longer assign the first step radiation data to the zeroth step.
- Change the return type of `ISSR.eval`, `SAC.eval`, and `PCR.eval` from `MetDataArray` to `MetDataset`. This is more consistent with the return type of other `pycontrails` models and more closely mirrors the behavior of vector models. Set output `attrs` metadata on the global `MetDataset` instead of the individual `MetDataArray` in each case.

14.12.2 Features

- Rewrite parts of the `pycontrails.core.datalib` module for higher performance and readability.
- Add optional `attrs` and `attrs_kwargs` parameters to `MetDataset` constructor. This allows the user to customize the attributes on the underlying `xarray.Dataset` object. This update makes `MetDataset` more consistent with `VectorDataset`.
- Add three new properties `provider_attr`, `dataset_attr`, and `product_attr` to `MetDataset`. These properties give metadata describing the underlying meterological data source.
- Add new `Model.transfer_met_source_attrs` method for more consistent handling of met source metadata on the `source` parameter passed into `Model.eval`.
- No longer require `geopotential` data when computing `tau_cirrus`. If neither `geopotential` nor `geopotential_height` are available, geopotential is approximated from the geometric height. No longer require geopotential on the `met` parameter in `Cocip` or `CocipGrid`.
- Remove the `Cocip` `shift_radiation_time` parameter. This is now inferred directly from the `rad` metadata. An error is raised if the necessary metadata is not present.
- Allow `Cocip` to run with both instantaneous (`W m-2`) and accumulated (`J m-2`) radiation data.
- Allow `Cocip` to run with accumulated ECMWF HRES radiation data.

14.12.3 Fixes

- Correct radiation unit in the `ACCF` wrapper model [#64]. Both instantaneous (`W m-2`) and accumulated (`J m-2`) radiation data are now supported, and the `ACCF` wrapper will handle each appropriately.
- Avoid unnecessary writing and reading of temporary files in `ERA5.cache_dataset` and `HRES.cache_dataset`.
- Fix timestep resolution bug in `GFSForecast`. When the `grid` parameter is 0.5 or 1.0, forecasts are only available every 3 hours. Previously, the `timesteps` property would define an hourly timestep.

14.12.4 Internals

- Include `name` parameter in `MetDataArray` constructor.
- Make the `coordinates.slice_domain` function slightly more performant by explicitly dropping nan values from the `request` parameter.
- Round unwieldy floating point numbers in `GeoVectorDataset._display_attrs`.
- Remove the `ecmwflibs` package from the `ecmwf` optional dependencies.
- Add NPY to `ruff` rules.
- Add convenience `MetDataset.standardize_variables` method.
- Remove the `p_settings` attribute on the `ACCF` interface. This is now constructed internally within `ACCF.eval`. Replace the `ACCF._update_accf_config` method with a `_get_accf_config` function.

14.13 v0.47.3

14.13.1 Fixes

- Strengthen `correct_fuel_flow` in the `PSmodel` to account for descent conditions.
- Clip the denominator computed in `pycontrails.physics.jet.equivalent_fuel_flow_rate_at_cruise`.
- Ensure the token used within GitHub workflows has the fewest privileges required. Set top-level permissions to `none` in each workflow file. Remove unnecessary permissions previously used in the `google-github-actions/auth` action.
- Fix bug in `radiative_forcing.effective_tau_contrail` identified in #99.
- Fix the unit for `vertical_velocity` in `geo.advect_level`.
- Fix bug appearing in `Flight._geodesic_interpolation` in which a single initial large gap was not interpolated with a geodesic path.

14.13.2 Internals

- Add `FlightPhase` to the `pycontrails` namespace.

14.14 v0.47.2

14.14.1 Features

- New experimental `GOES` interface for downloading and visualizing GOES-16 satellite imagery.
- Add new GOES example notebook highlighting the interface.
- Build python 3.12 wheels for Linux, macOS, and Windows on release. This is in addition to the existing python 3.9, 3.10, and 3.11 wheels.

14.14.2 Fixes

- Use the experimental version number parameter `E` in `pycontrails.ecmwf.hres.get_forecast_filename`. Update the logic involved in setting the dissemination data stream indicator `S`.
- Change the behavior of `_altitude_interpolation` method that is called within `resample_and_fill`. Step climbs are now placed in the middle of long flight segments. Descents continue to occur at the end of segments.

14.14.3 Internals

- Provide consistent `ModuleNotFoundError` messages when optional dependencies are not installed.
- Move the `synthetic_flight` module into the `pycontrails.ext` namespace.

14.15 v0.47.1

14.15.1 Fixes

- Fix bug in `PSGrid` in which the `met` data was assumed to be already loaded into memory. This caused errors when running `PSGrid` with a `MetDataset` source.
- Fix bug (#86) in which `Cocip.eval` loses the `source` fuel type. Instead of instantiating a new `Flight` or `Fleet` instance with the default fuel type, the `Cocip._bundle_results` method now overwrites the `self.source.data` attribute with the bundled predictions.
- Avoid a memory explosion when running `Cocip` on a large non-dask-backed `met` parameter. Previously the `tau_cirrus` computation would be performed in memory over the entire `met` dataset.
- Replace `datetime.utcfromtimestamp` (deprecated in python 3.12) with `datetime.fromtimestamp`.
- Explicitly support python 3.12 in the `pyproject.toml` build system.

14.15.2 Internals

- Add `compute_tau_cirrus_in_model_init` parameter to `CocipParams`. This controls whether to compute the cirrus optical depth in `Cocip.__init__` or `Cocip.eval`. When set to `"auto"` (the default), the `tau_cirrus` is computed in `Cocip.__init__` if and only if the `met` parameter is dask-backed.
- Change data requirements for the `EmpiricalGrid` aircraft performance model.
- Consolidate `ERA5.cache_dataset` and `HRES.cache_dataset` onto common `ECMWFAPI.cache_dataset` method. Previously the child implementations were identical.
- No longer require the `pyproj` package as a dependency. This is now an optional dependency, and can be installed with `pip install pycontrails[pyproj]`.

14.16 v0.47.0

Implement a Poll-Schumann (`PSGrid`) theoretical aircraft performance over a grid.

14.16.1 Breaking changes

- Move the `pycontrails.models.aircraft_performance` module to `pycontrails.core.aircraft_performance`.
- Rename `PSModel` -> `PSFlight`.

14.16.2 Fixes

- Use the instance `fuel` attribute in the `Fleet.to_flight_list` method. Previously, the default `JetA` fuel was always used.
- Ensure the `Fleet.fuel` attribute is inferred from the underlying sequence of flights in the `Fleet.from_seq` method.

14.16.3 Features

- Implement the `PSGrid` model. For a given aircraft type and position, this model computes optimal aircraft performance at cruise conditions. In particular, this model can be used to estimate fuel flow, engine efficiency, and aircraft mass at cruise. In particular, the `PSGrid` model can now be used in conjunction with `CocipGrid` to simulate contrail formation over a grid.
- Refactor the `Emissions` model so that `Emissions.eval` runs with `source: GeoVectorDataset`. Previously, the `eval` method required a `Flight` instance for the `source` parameter. This change allows the `Emissions` model to run more seamlessly as a sub-model of a gridded model (ie, `CocipGrid`),
- No longer require `pycontrails-bada` to import or run the `CocipGrid` model. Instead, the `CocipGridParams.aircraft_performance` parameter can be set to any `AircraftPerformanceGrid` instance. This allows the `CocipGrid` model to run with any aircraft performance model that implements the `AircraftPerformanceGrid` interface.
- Add experimental `EmpiricalAircraftPerformanceGrid` model.
- Add convenience `GeoVectorDataset.T_isa` method to compute the ISA temperature at each point.

14.16.4 Internals

- Add optional `climb_descend_at_end` parameter to the `Flight.resample_and_fill` method. If True, the climb or descent will be placed at the end of each segment rather than the start.
- Define `AircraftPerformanceGridParams`, `AircraftPerformanceGrid`, and `AircraftPerformanceGridData` abstract interfaces for gridded aircraft performance models.
- Add `set_attr` parameter to `Models.get_source_param`.
- Better handle `source`, `source.attrs`, and `params` customizations in `CocipGrid`.
- Include additional classes and functions in the `pycontrails.models.emissions` module.
- Hardcode the paths to the static data files used in the `Emissions` and `PSFlight` models. Previously these were configurable by model parameters.
- Add `altitude_ft` parameter to the `GeoVectorDataset` constructor. Warn if at least two of `altitude_ft`, `altitude`, and `level` are provided.
- Allow instantiation of `Model` instances with `params: ModelParams`. Previously, the `params` parameter was required to be a `dict`. The current implementation checks that the `params` parameter is either a `dict` or has type `default_params` on the `Model` class.

14.17 v0.46.0

Support “dry advection” simulation.

14.17.1 Features

- Add new `DryAdvection` model to simulate sediment-free advection of an aircraft's exhaust plume. This model is experimental and may change in future releases. By default, the current implementation simulates plume geometry as a cylinder with an elliptical cross section (the same geometry assumed in CoCiP). Wind shear perturbs the ellipse azimuth, width, and depth over the plume evolution. The `DryAdvection` model may also be used to simulate advection without wind-shear effects by setting the model parameters `azimuth`, `width`, and `depth` to None.
- Add new Dry Advection example notebook highlighting the new `DryAdvection` model and comparing it to the `Cocip` model.
- Add optional `fill_value` parameter to `VectorDataset.sum`.

14.17.2 Fixes

- (#80) Fix unit error in `wake_vortex.turbulent_kinetic_energy_dissipation_rate`. This fix affects the estimate of wake vortex max downward displacement and slightly changes `Cocip` predictions.
- Change the implementation of `Flight.resample_and_fill` so that lat-lon interpolation is linear in time. Previously, the timestamp associated to a waypoint was floored according to the resampling frequency without updating the position accordingly. This caused errors in segment calculations (e.g., true airspeed).

14.17.3 Internals

- Add optional `keep_original_index` parameter to the `Flight.resample_and_fill` method. If `True`, the time original index is preserved in the output in addition to the new time index obtained by resampling.
- Improve docstrings in `wake_vortex` module
- Rename `wake_vortex` functions to remove `get_` prefix at the start of each function name.
- Add pytest command line parameter `--regenerate-results` to regenerate static test fixture results. Automate in make recipe `make pytest-regenerate-results`.
- Update handling of `GeoVectorDataset.required_keys` the `GeoVectorDataset` constructor. Add class variable `GeoVectorDataset.vertical_keys` for handing the vertical dimension.
- Rename `CocipParam.max_contrail_depth` -> `CocipParam.max_depth`.
- Add `units.dt_to_seconds` function to convert `np.timedelta64` to `float` seconds.
- Rename `thermo.p_dz` -> `thermo.pressure_dz`.

14.18 v0.45.0

Add experimental support for simulating radiative effects due to contrail-contrail overlap.

14.18.1 Features

- Support simulating contrail contrail overlap when running the `Cocip` model with a `Fleet` source. The `contrail_contrail_overlapping` and `dz_overlap_m` parameters govern the overlap calculation. This mode of calculation is still experimental and may change in future releases.
- Rewrite the `pycontrails.models.cocip.output` modules into a single `pycontrails.cocip.output_formats` module. The new module supports flight waypoint summary statistics, contrail waypoints summary statistics, gridded outputs, and time-slice outputs.
- Add new `GeoVectorDataset.to_lon_lat_grid` method. This method can be thought of as a partial inverse to the `MetDataset.to_vector` method. The current implementation is brittle and may be revised in a future release.

14.18.2 Fixes

- Extend `Models.set_source_met` to allow `interpolation_q_method="cubic-spline"` when working with `MetDataset` source (ie, so-called gridded models). Previously a `NotImplementedError` was raised.
- Ensure the `Flight.copy` implementation works with `Fleet` instances as well.
- Avoid looping over `keys` twice in `VectorDataset.broadcast_attrs`. This is a slight performance enhancement.
- Fix `Fleet` signature for compatibility with `Flight`.
- Fix a few hard-coded assumptions in broadcasting aircraft performance and emissions when running `Cocip` with a `Fleet` source. The previous implementation did not consider the possibility of aircraft performance variables on `Flight.data` and `Flight.attrs` separately.

14.18.3 Internals

- Add optional `raise_error` parameter to the `VectorDataset.broadcast_attrs` method.
- Update `Fleet` internals.

14.19 v0.44.2

14.19.1 Fixes

- Narrow type hints on the ABC `AircraftPerformance` model. The `AircraftPerformance.eval` method requires a `Flight` object for the `source` parameter.
- In `PSFlight.eval`, explicitly set any aircraft performance data at waypoints with zero true airspeed to `np.nan`. This avoids numpy `RuntimeWarning`s without affecting the results.
- Fix corner-case in the `polygon.buffer_and_clean` function in which the polygon created by buffering the `opencv` contour is not valid. Now a second attempt to buffer the polygon is made with a smaller buffer distance.
- Ignore RuntimeError raised in `scipy.optimize.newton` if the maximum number of iterations is reached before convergence. This is a workaround for occasional false positive convergence warnings. The pycontrails use-case may be related to this GitHub issue.
- Update the `Models.__init__` warnings when `humidity_scaling` is not provided. The previous warning provided an outdated code example.

- Ensure the `interpolation_q_method` used in a parent model is passed into the `humidity_scaling` child model in the `Models.__init__` method. If the two `interpolation_q_method` values are different, a warning is issued. This could be extended to other model parameters in the future.

14.19.2 Features

- Enable `ExponentialBoostLatitudeCorrectionHumidityScaling` humidity scaling for the model parameter `interpolation_q_method="cubic_spline"`.
- Add GFS notebook example.

14.19.3 Breaking changes

- Remove `ExponentialBoostLatitudeCorrectionHumidityScalingParams`. These parameters are now hard-coded in the `ExponentialBoostLatitudeCorrectionHumidityScaling` model.

14.20 v0.44.1

14.20.1 Breaking changes

- By default, call `xr.open_mfdataset` with `lock=False` in the `MetDataSource.open_dataset` method. This helps alleviate a `dask` threading issue similar to this GitHub issue.

14.20.2 Fixes

- Support `MetDataset` source in the `HistogramMatching` humidity scaling model. Previously only `GeoVectorDataset` sources were explicitly supported.
- Replace `np.gradient` with `dask.array.gradient` in the `tau_cirrus` module. This ensures that the computation is done lazily for dask-backed arrays.
- Round to 6 digits in the `polygon.determine_buffer` function. This avoid unnecessary warnings for rounding errors.
- Fix type hint for `opencv-python` 4.8.0.74.

14.20.3 Internals

- Take more care with `float` and `int` types in the `contrail_properties` module. Prefer `np.clip` to `np.where` or `np.maximum` for clipping values.
- Support `air_temperature` in `CocipGrid` verbose formation outputs.
- Remove `pytest-timeout` dev dependency.

14.21 v0.44.0

Support for the Poll-Schumann aircraft performance model.

14.21.1 Features

- Implement a basic working version of the Poll-Schumann (PS) aircraft performance model. This is experimental and may undergo revision in future releases. The PS Model currently supports the following 53 aircraft types:
  - A30B
  - A306
  - A310
  - A313
  - A318
  - A319
  - A320
  - A321
  - A332
  - A333
  - A342
  - A343
  - A345
  - A346
  - A359
  - A388
  - B712
  - B732
  - B733
  - B734
  - B735
  - B736
  - B737
  - B738
  - B739
  - B742
  - B743
  - B744
  - B748

  - B752
  - B753
  - B762
  - B763
  - B764
  - B77L
  - B772
  - B77W
  - B773
  - B788
  - B789
  - E135
  - E145
  - E170
  - E195
  - MD82
  - MD83
  - GLF5
  - CRJ9
  - DC93
  - RJ1H
  - B722
  - A20N
  - A21N

  The "gridded" version of this model is not yet implemented. This will be added in a future release.

- Improve the runtime of instantiating the `Emissions` model by a factor of 10-15x. This translates to a time savings of several hundred milliseconds on modern hardware. This improvement is achieved by more efficient parsing of the underlying static data and by deferring the construction of the interpolation artifacts until they are needed.
- Automatically use a default engine type from the aircraft type in the `Emissions` model if an `engine_uid` parameter is not included on the `source`. This itself is configurable via the `use_default_engine_uid` parameter on the `Emissions` model. The default mappings from aircraft types to engines is included in `pycontrails/models/emissions/static/default-engine-uids.csv`.

14.21.2 Breaking changes

- Remove the `Aircraft` dataclass from `Flight` instantiation. Any code previously using this should instead directly pass additional `attrs` to the `Flight` constructor.
- The `load_factor` is now required in `AircraftPerformance` models. The global `DEFAULT_LOAD_FACTOR` constant in `pycontrails.models.aircraft_performance` provides a reasonable default. This is currently set to 0.7.
- Use a `takeoff_mass` parameter in `AircraftPerformance` models if provided in the `source.attrs` data.
- No longer use a reference mass `ref_mass` in `AircraftPerformance` models. This is replaced by the `takeoff_mass` parameter if provided, or calculated from operating empty operating mass, max payload mass, total fuel consumption mass, reserve fuel mass, and the load factor.

14.21.3 Fixes

- Remove the `fuel` parameter from the `Emissions` model. This is inferred directly from the `source` parameter in `Emissions.eval`.
- Fix edge cases in the `jet.reserve_fuel_requirements` implementation. The previous version would return `nan` for some combinations of `fuel_flow` and `segment_phase` variables.
- Fix a spelling mistake: `units.kelvin_to_celcius` -> `units.kelvin_to_celsius`.

14.21.4 Internals

- Use `ruff` in place of `pydocstyle` for linting docstrings.
- Use `ruff` in place of `isort` for sorting imports.
- Update the `AircraftPerformance` template based on the patterns used in the new `PSFlight` class. This may change again in the future.

14.22 v0.43.0

Support experimental interpolation against gridded specific humidity. Add new data-driven humidity scaling models.

14.22.1 Features

- Add new experimental `interpolation_q_method` field to the `ModelParams` data class. This parameter controls the interpolation methodology when interpolation against gridded specific humidity. The possible values are:
  - `None`: Interpolate linearly against specific humidity. This is the default behavior and is the same as the previous behavior.
  - `"cubic-spline"`: Apply cubic-spline scaling to the interpolation table vertical coordinate before interpolating linearly against specific humidity.
  - `"log-q-log-p"`: Interpolate in the log-log domain against specific humidity and pressure.

  This interpolation parameter is used when calling `pycontrails.core.models.interpolate_met`. It can also be used directly with the new lower-level `pycontrails.core.models.interpolate_gridded_specific_humidity` function.

- Add new experimental `HistogramMatching` humidity scaling model to match RHi values against IAGOS observations. The previous `HistogramMatchingWithEckel` scaling is still available when working with ERA5 ensemble members.
- Add new tutorial discussing the new specific humidity interpolation methodology.

14.22.2 Breaking changes

- Add an optional `q_method` parameter to the `pycontrails.core.models.interpolate_met` function. The default value `None` agrees with the previous behavior.
- Change function signatures in the `cocip.py` module for consistency. The `interp_kwargs` parameter is now unpacked in the `calc_timestep_meterology` signature. Rename `kwargs` to `interp_kwargs` where appropriate.
- Remove the `cache_size` parameter in `MetDataset.from_zarr`. Previously this parameter allowed the user to wrap the Zarr store in a `LRUCacheStore` to improve performance. Changes to Zarr internals have broken this approach. Custom Zarr patterns should now be handled outside of `pycontrails`.

14.22.3 Fixes

- Recompute and extend quantiles for histogram matching humidity scaling. Quantiles are now available for each combination of `q_method` and the following ERA5 data products: reanalysis and ensemble members 0-9. This data is available as a parquet file and is packaged with `pycontrails`.
- Fix the precomputed Eckel coefficients. Previous values where computed for different interpolation assumptions and were not correct for the default interpolation method.
- Clip the scaled humidity values computed by the `humidity_scaling.eckel_scaling` function to ensure that they are non-negative. Previously, both relative and specific humidity values arising from Eckel scaling could be negative.
- Handle edge case of all NaN values in the `T_critical_sac` function in the `sac.py` module.
- Avoid extraneous copy when calling `VectorDataset.sum`.
- Officially support `numpy` v1.25.0.
- Set a `pytest-timeout` limit for tests in `tests/unit/test_ecmwf.py` to avoid hanging tests.
- Add `forecast_step` parameter to the `ACCF` model.

14.22.4 Internals

- Refactor auxillary functions used by `HistogramMatchingWithEckel` to better isolated histogram matching from Eckel scaling.
- Refactor `intersect_met` method in `pycontrails.core.models` to handle experimental `q_method` parameter.
- Include a `q_method` field in `Model.interp_kwargs`.
- Include precomputed humidity lapse rate values in the new `pycontrails.core.models._load_spline` function.
- Move the `humidity_scaling.py` module into its own subdirectory within `pycontrails/models`.

14.23 v0.42.2

Re-release of *v0.42.1*.

14.24 v0.42.1

14.24.1 Features

- Add new `HistogramMatchingWithEckel` experimental humidity scaling model. This is still a work in progress.
- Add new `Flight.fit_altitude` method which uses piecewise linear fitting to smooth a flight profile.
- Add new `pycontrails.core.flightplan` module for parsing ATC flight plans between string and dictionary representations.
- Add new *airports* and *flightplan* examples.

14.24.2 Breaking changes

- No longer attach empty fields "sdr", "rsr", "olr", "rf_sw", "rf_lw", "rf_net" onto the `source` parameter in `Cocip.eval` when the flight doesn't generate any persistent contrails.
- Remove params `humidity_scaling`, `rhi_adj_uncertainty`, and `rhi_boost_exponent_uncertainty` from `CocipUncertaintyParams`.
- Change the default value for `parallel` from True to False in `xr.open_mfdataset`. This can be overridden by setting the `xr_kwargs` parameter in `ERA5.open_metdataset`.

14.24.3 Fixes

- Fix a unit test (`test_dtypes.py::test_issr_sac_grid_output`) that occasionally hangs. There may be another test in `test_ecmwf.py` that suffers from the same issue.
- Fix issue encountered in `Cocip.eval` when concatenating contrails with inconsistent values for `_out_of_bounds`. This is only relevant when running the model with the experimental parameter `interpolation_use_indices=True`.
- Add a `Fleet.max_distance_gap` property. The previous property on the `Flight` class was not applicable to `Fleet` instances.
- Fix warning in `Flight` class to correctly suggest adding kwarg `drop_duplicated_times`.
- Fix an issue in the `VectorDataset` constructor with a `data` parameter of type `pd.DataFrame`. Previously, time data was rewritten to the underlying DataFrame. This could cause copy-on-write issues if the DataFrame was a view of another DataFrame. This is now avoided.

14.24.4 Internals

- When possible, replace type hints `np.ndarray -> np.typing.NDArray[np.float_]` in the `cocip`, `cocip_params`, `cocip_uncertainty`, `radiative_forcing`, and `wake_vortex` modules.
- Slight performance enhancements in the `radiative_forcing` module.
- Change the default value of `u_wind` and `v_wind` from None to 0 in `Flight.segment_true_airspeed`. This makes more sense semantically.

14.25 v0.42.0

Phase 1 of the Spire datalib, which contains functions to identify unique flight trajectories from the raw Spire ADS-B data.

14.25.1 Features

- Add a `pycontrails.core.airport` module to read and process the global airport database, which can be used to identify the nearest airport to a given coordinate.
- Add a `pycontrails.datalib.spire.clean` function to remove and address erroneous waypoints in the raw Spire ADS-B data.
- Add a `pycontrails.datalib.spire.filter_altitude` function to remove noise in cruise altitude.
- Add a `pycontrails.datalib.spire.identify_flights` function to identify unique flight trajectories from ADS-B messages.
- Add a `pycontrails.datalib.spire.validate_trajectory` function to check the validity of the identified trajectories from ADS-B messages.
- Add a `FlightPhase` integer `Enum` in the `flight` module. This includes a new `level_flight` flight phase.

14.25.2 Internals

- Add unit tests providing examples to identify unique flights.
- Rename `flight._dt_waypoints` -> `flight.segment_duration`.
- Move `jet.rate_of_climb_descent` -> `flight.segment_rocd`.
- Move `jet.identify_phase_of_flight` -> `flight.segment_phase`.
- Update `FlightPhase` to be a dictionary enumeration of flight phases.
- Add references to `traffic` library.

14.26 v0.41.0

Improve polygon algorithms.

14.26.1 Features

- Rewrite the `polygon` module to run computation with opencv in place of scikit-image for finding contours. This change improves the algorithm runtime and fixes some previous unstable behavior in finding nested contours. For an introduction to the methodology, see the OpenCV contour tutorial.

14.26.2 Breaking changes

- Completely rewrite the `polygon` module. Replace the "main" public function `polygon.find_contours_to_depth` with `polygon.find_multipolygon`. Replace the `polygon.contour_to_lat_lon` function with `polygon.multipolygon_to_geojson`. Return `shapely` objects when convenient to do so.
- Convert continuous data to binary for polygon computation.
- Remove parameters `min_area_to_iterate` and `depth` in the `MetDataArray.to_polygon_feature` method. The `depth` parameter has been replaced by the boolean `interiors` parameter. Add a `properties` parameter for adding properties to the `Polygon` and `MultiPolygon` features. The `max_area` and `epsilon` parameters are now expressed in terms of latitude-longitude degrees.

14.26.3 Internals

- Add `opencv-python-headless>=4.5` as an optional "vis" dependency. Some flavor of `opencv` is required for the updated polygon algorithms.

14.27 v0.40.1

14.27.1 Fixes

- Use oldest-supported-numpy for building pycontrails wheels. This allows pycontrails to be compatible with environments that use old versions of numpy. The pycontrails v0.40.0 wheels are not compatible with numpy 1.22.

14.28 v0.40.0

Support scipy 1.10, improve interpolation performance, and fix many windows issues.

14.28.1 Features

- Improve interpolation performance by cythonizing linear interpolation. This extends the approach taken in scipy 1.10. The pycontrails *cython routines* allow for both float64 and float32 grids via cython fused types (the current scipy implementation assumes float64). In addition, interpolation up to dimension 4 is supported (the scipy implementation supports dimension 1 and 2).
- Officially support scipy 1.10.
- Officially test on windows in the GitHub Actions CI.
- Build custom wheels for python 3.9, 3.10, and 3.11 for the following platforms:
  - Linux (x86_64)
  - macOS (arm64 and x86_64)
  - Windows (x86_64)

14.28.2 Breaking changes

- Change `MetDataset` and `MetDataArray` conventions: underlying dimension coordinates are automatically promoted to float64.
- Change how datetime arrays are converted to floating values for interpolation. The new approach introduces small differences compared with the previous implementation. These differences are significant enough to see relative differences in CoCiP predictions on the order of 1e-4.

14.28.3 Fixes

- Unit tests no longer raise errors when the `pycontrails-bada` package is not installed. Instead, some tests are skipped.
- Fix many numpy casting issues encountered on windows.
- Fix temp file issues encountered on windows.
- Officially support changes in `xarray` 2023.04 and `pandas` 2.0.

14.28.4 Internals

- Make the `interpolation` module more aligned with scipy 1.10 enhancements to the `RegularGridInterpolator`. In particular, grid coordinates now must be float64.
- Use cibuildwheel to build wheels for Linux, macOS (arm64 and x86_64), and Windows on *release* in Github Actions. Allow this workflow to be triggered manually to test the release process without actually publishing to PyPI.
- Simplify interpolation with pre-computed indices (invoked with the model parameter `interpolation_use_indices`) via a `RGIArtifacts` interface.
- Overhaul much of the interpolation module to improve performance.
- Slight performance enhancements to the `met` module.

14.29 v0.39.6

14.29.1 Features

- Add `geo.azimuth` and `geo.segment_azimuth` functions to calculate the azimuth between coordinates. Azimuth is the angle between coordinates relative to true north on the interval `[0, 360)`.

14.29.2 Fixes

- Fix edge case in polygon algorithm by utilizing the `fully_connected` parameter in `measure.find_contours`. This update leads to slight changes in interior contours in some cases.
- Fix hard-coded POSIX path in `conftest.py` for windows compatibility.
- Address `PermissionError` raised by `shutil.copy` when the destination file is open in another thread. The copy is skipped and a warning is logged.
- Fix some unit tests in `test_vector.py` and `test_ecmwf.py` for windows compatibility. There are still a few tests that fail on windows (unrelated to changes in this release) that will be fixed in v0.40.0.
- Allow `cachestore=None` to skip caching in `ERA5`, `HRES`, and `GFS` interfaces. Previously a default `DiskCacheStore` was created even when `cachestore=None`. By default, caching is still enabled.

14.29.3 Internals

- Clean up `geo` module docstrings.
- Add Wikipedia reference for Azimuth.
- Convert `MetBase._load` to from a class method to a function.

14.30 v0.39.5

14.30.1 Fixes

- Fix `docs/examples/CoCiP.ipynb` example demonstrating aircraft performance integration
- Fix unit test caused by breaking change in pyproj 3.5.0

14.30.2 Internals

- Add additional Zenodo metadata
- Execute notebook examples in Docs Action

14.31 v0.39.4

14.31.1 Internals

- Add additional Zenodo metadata
- Add Doc / Notebook test Action to run notebook test (`make nb-test`) and doctests (`make doctest`) on pull requests.
- Update Docs Action to use python 3.11.

14.32 v0.39.3

14.32.1 Fixes

- Update links in the *README*.
- Update the *release guide* to include a checklist of steps to follow when cutting a release.

14.33 v0.39.2

14.33.1 Fixes

- Fix links in documentation website.
- Deploy build to PyPI (in addition to Test PyPI) on release.

14.34 v0.39.1

14.34.1 Features

- Use setuptools_scm to manage the `pycontrails` version.

14.35 v0.39.0

14.35.1 Features

- Add Apache-2 LICENSE and NOTICE files (#21)
- Add CONTRIBUTING document
- Update documentation website py.contrails.org. Includes `install` and `develop` guides, update citations and many other small improvements (#19).
- Initiate the Github Discussion forum (#22).

14.35.2 Fixes

- Fix erroneous docstrings in `emissions.py` (#25)

14.35.3 Internals

- Add Github action to push to `pypi` on tag (#3)
- Replace `flake8` with `ruff` linter
- Add `nb-clean` to pre-commit hooks for example notebooks
- Add `doc8` rst linter and pre-commit hook

14.36 v0.38.0

14.36.1 Breaking changes

- Change default value of `epsilon` parameter in method `MetDataArray.to_polygon_feature` from 0.15 to 0.0.
- Change the polygon simplification algorithm. The new implementation uses `shapely.buffer` and doesn't try to preserve the topology of the simplified polygon. This change may result in slightly different polygon geometries.

14.36.2 Internals

- Add `depth` parameter to `MetDataArray.to_polygon_feature` to control the depth of the contour searching.
- Add experimental `convex_hull` parameter to `MetDataArray.to_polygon_feature` to control whether to take the convex hull of each contour.
- Warn if `iso_value` is not specified in `MetDataArray.to_polygon_feature`.

14.36.3 Fixes

- Consolidate three redundant implementations of standardizing variables into a single `met.standardize_variables`.
- Ensure simplified polygons returned by `MetDataArray.to_polygon_feature` are disjoint. While non-disjoint polygons don't violate the GeoJSON spec, they can cause problems in some applications.

14.37 v0.37.3

14.37.1 Internals

- Add citations for ISA calculations
- Abstract functionality to convert a Dataset or DataArray to longitude coordinates `[-180, 180)` into `core.met.shift_longitude`. Add tests for method.
- Add auto-formatting checks to CI testing

14.38 v0.37.2

ACCF integration updates

14.38.1 Fixes

- Fixes ability to evaluate ACCF model over a `MetDataset` grid by passing in tuple of (dims, data) when assigning data
- Fixes minor issues with ACCF configuration and allows more configuration options
- Updates example ACCF notebook with example of how to set configuration options when evaluating ACCFs over a grid

14.39 v0.37.1

14.39.1 Features

- Include "rhi" and "iwc" variables in `CocipGrid` verbose outputs.

14.40 v0.37.0

14.40.1 Breaking changes

- Update CoCiP unit test static results for breaking changes in tau cirrus calculation. The relative difference in pinned energy forcing values is less than 0.001%.

14.40.2 Fixes

- Fix geopotential height gradient calculation in the `tau_cirrus` module. When calculating finite differences along the vertical axis, the tau cirrus model previously divided the top and bottom differences by 2. To numerically approximate the derivative at the top and bottom levels, these differences should have actually been divided by 1. The calculation now uses `np.gradient` to calculate the derivative along the vertical axis, which handles this correctly.
- Make tau cirrus calculation slightly more performant.
- Include a warning in the suspect `IFS._calc_geopotential` implementation.

14.40.3 Internals

- Remove `_deprecated.tau_cirrus_alt`. This function is now the default `tau_cirrus` calculation. The original `tau_cirrus` calculation is still available in the `_deprecated` module.
- Run `flake8` over test modules.

BIBLIOGRAPHY

[1] A Celikel and F Jelinek. Forecasting civil aviation fuel burn and emissions in Europe. *EUROCONTROL Experimental Centre*, 2001.

[2] D.S. Lee, D.W. Fahey, A. Skowron, M.R. Allen, U. Burkhardt, Q. Chen, S.J. Doherty, S. Freeman, P.M. Forster, J. Fuglestvedt, A. Gettelman, R.R. De León, L.L. Lim, M.T. Lund, R.J. Millar, B. Owen, J.E. Penner, G. Pitari, M.J. Prather, R. Sausen, and L.J. Wilcox. The contribution of global aviation to anthropogenic climate forcing for 2000 to 2018. *Atmospheric Environment*, 244:117834, January 2021. doi:10.1016/j.atmosenv.2020.117834.

[3] M. E. J. Stettler, S. Eastham, and S. R. H. Barrett. Air quality and public health impacts of UK airports. Part I: Emissions. *Atmospheric Environment*, 45(31):5415–5424, October 2011. doi:10.1016/j.atmosenv.2011.07.012.

[4] J. T. Wilkerson, M. Z. Jacobson, A. Malwitz, S. Balasubramanian, R. Wayson, G. Fleming, A. D. Naiman, and S. K. Lele. Analysis of emission data from global commercial aviation: 2004 and 2006. *Atmospheric Chemistry and Physics*, 10(13):6391–6408, July 2010. doi:10.5194/acp-10-6391-2010.

[5] Roger Teoh, Ulrich Schumann, Christiane Voigt, Tobias Schripp, Marc Shapiro, Zebediah Engberg, Jarlath Molloy, George Koudis, and Marc E. J. Stettler. Targeted Use of Sustainable Aviation Fuel to Maximize Climate Benefits. *Environmental Science & Technology*, November 2022. doi:10.1021/acs.est.2c05781.

[6] Tobias Schripp, Bruce E. Anderson, Uwe Bauder, Bastian Rauch, Joel C. Corbin, Greg J. Smallwood, Prem Lobo, Ewan C. Crosbie, Michael A. Shook, Richard C. Miake-Lye, Zhenhong Yu, Andrew Freedman, Philip D. Whitefield, Claire E. Robinson, Steven L. Achterberg, Markus Köhler, Patrick Oßwald, Tobias Grein, Daniel Sauer, Christiane Voigt, Hans Schlager, and Patrick LeClercq. Aircraft engine particulate matter emissions from sustainable aviation fuels: Results from ground-based measurements during the NASA/DLR campaign ECLIF2/ND-MAX. *Fuel*, 325:124764, October 2022. doi:10.1016/j.fuel.2022.124764.

[7] M. Anwar H. Khan, Joel Brierley, Kieran N. Tait, Steve Bullock, Dudley E. Shallcross, and Mark H. Lowenberg. The Emissions of Water Vapour and NOx from Modelled Hydrogen-Fuelled Aircraft and the Impact of NOx Reduction on Climate Compared with Kerosene-Fuelled Aircraft. *Atmosphere*, 13(10):1660, October 2022. doi:10.3390/atmos13101660.

[8] Robert W. Carver and Alex Merose. ARCO-ERA5: An Analysis-Ready Cloud-Optimized Reanalysis Dataset. In *103rd AMS Annual Meeting*. AMS, January 2023.

[9] Luke Kulik. *Satellite-Based Detection of Contrails Using Deep Learning*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, September 2019.

[10] SEVIRI RGB Cal Module - part II. https://resources.eumetrain.org/data/4/410/navmenu.php?tab=5&page=2.0.0.

[11] Ulrich Schumann. On conditions for contrail formation from aircraft exhausts (Über Bedingungen zur Bildung von Kondensstreifen aus Flugzeugabgasen). *Meteorologische Zeitschrift*, 5(1):4–23, March 1996. doi:10.1127/metz/5/1996/4.

[12] Michael Ponater. Contrails in a comprehensive global climate model: Parameterization and radiative forcing results. *Journal of Geophysical Research*, 107(D13):4164, 2002. doi:10.1029/2001JD000429.

[13] U. Schumann. A contrail cirrus prediction model. *Geoscientific Model Development*, 5(3):543–580, May 2012. doi:10.5194/gmd-5-543-2012.

[14] U. Schumann, B. Mayer, K. Graf, and H. Mannstein. A Parametric Radiative Forcing Model for Contrail Cirrus. *Journal of Applied Meteorology and Climatology*, 51(7):1391–1406, July 2012. doi:10.1175/JAMC-D-11-0242.1.

[15] Roger Teoh, Ulrich Schumann, Edward Gryspeerdt, Marc Shapiro, Jarlath Molloy, George Koudis, Christiane Voigt, and Marc E. J. Stettler. Aviation contrail climate effects in the North Atlantic from 2016 to 2021. *Atmospheric Chemistry and Physics*, 22(16):10919–10935, August 2022. doi:10.5194/acp-22-10919-2022.

[16] U. Schumann and P. Wendling. Determination of Contrails from Satellite Data and Observational Results. In U. Schumann, editor, *Air Traffic and the Environment — Background, Tendencies and Potential Global Atmospheric Effects*, Lecture Notes in Engineering, 138–153. Berlin, Heidelberg, 1990. Springer. doi:10.1007/978-3-642-51686-3_9.

[17] Ulrich Schumann. A contrail cirrus prediction tool. In Robert Sausen, Peter F. J. van Velthoven, Claus Brüning, and Anja Blum, editors, *Proceedings of the 2nd International Conference on Transport, Atmosphere and Climate (TAC-2)*, volume 2010–10, 69–74. Aachen, Germany and Maastricht, The Netherlands, 2010. DLR.

[18] C. Voigt, U. Schumann, T. Jurkat, D. Schäuble, H. Schlager, A. Petzold, J.-F. Gayet, M. Krämer, J. Schneider, S. Borrmann, J. Schmale, P. Jessberger, T. Hamburger, M. Lichtenstern, M. Scheibe, C. Gourbeyre, J. Meyer, M. Kübbeler, W. Frey, H. Kalesse, T. Butler, M. G. Lawrence, F. Holzäpfel, F. Arnold, M. Wendisch, A. Döpelheuer, K. Gottschaldt, R. Baumann, M. Zöger, I. Sölch, M. Rautenhaus, and A. Dörnbrack. In-situ observations of young contrails – overview and selected results from the CONCERT campaign. *Atmospheric Chemistry and Physics*, 10(18):9039–9056, September 2010. doi:10.5194/acp-10-9039-2010.

[19] Ulrich Schumann, Kaspar Graf, and Hermann Mannstein. Potential to reduce the climate impact of aviation by flight level changes. In *3rd AIAA Atmospheric Space Environments Conference*. Honolulu, Hawaii, June 2011. American Institute of Aeronautics and Astronautics. doi:10.2514/6.2011-3376.

[20] Ulrich Schumann, Kaspar Graf, Hermann Mannstein, and Bernhard Mayer. Contrails: Visible Aviation Induced Climate Impact. In Ulrich Schumann, editor, *Atmospheric Physics*, pages 239–257. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-30183-4_15.

[21] U. Schumann, B. Mayer, K. Gierens, S. Unterstrasser, P. Jessberger, A. Petzold, C. Voigt, and J.-F. Gayet. Effective Radius of Ice Particles in Cirrus and Contrails. *Journal of the Atmospheric Sciences*, 68(2):300–321, February 2011. doi:10.1175/2010JAS3562.1.

[22] U. Schumann, J. E. Penner, Yibin Chen, Cheng Zhou, and K. Graf. Dehydration effects from contrails in a coupled contrail–climate model. *Atmospheric Chemistry and Physics*, 15(19):11179–11199, October 2015. doi:10.5194/acp-15-11179-2015.

[23] Roger Teoh, Ulrich Schumann, Arnab Majumdar, and Marc E. J. Stettler. Mitigating the Climate Forcing of Aircraft Contrails by Small-Scale Diversions and Technology Adoption. *Environmental Science & Technology*, 54(5):2941–2950, March 2020. doi:10.1021/acs.est.9b05608.

[24] U. Schumann, L. Bugliaro, A. Dörnbrack, R. Baumann, and C. Voigt. Aviation Contrail Cirrus and Radiative Forcing Over Europe During 6 Months of COVID-19. *Geophysical Research Letters*, 48(8):e2021GL092771, 2021. doi:10.1029/2021GL092771.

[25] Ulrich Schumann, Ian Poll, Roger Teoh, Rainer Koelle, Enrico Spinielli, Jarlath Molloy, George S. Koudis, Robert Baumann, Luca Bugliaro, Marc Stettler, and Christiane Voigt. Air traffic and contrail changes over Europe during COVID-19: a model study. *Atmospheric Chemistry and Physics*, 21(10):7429–7450, May 2021. doi:10.5194/acp-21-7429-2021.

[26] Feijia Yin, Volker Grewe, Federica Castino, Pratik Rao, Sigrun Matthes, Katrin Dahlmann, Simone Dietmüller, Christine Frömming, Hiroshi Yamashita, Patrick Peter, Emma Klingaman, Keith P. Shine, Benjamin Lührs, and Florian Linke. Predicting the climate impact of aviation for en-route emissions: the algorithmic climate change function submodel ACCF 1.0 of EMAC 2.53. *Geoscientific Model Development*, 16(11):3313–3334, June 2023. doi:10.5194/gmd-16-3313-2023.

[27] Ulrich Schumann and Kaspar Graf. Aviation-induced cirrus and radiation changes at diurnal timescales. *Journal of Geophysical Research: Atmospheres*, 118(5):2404–2421, March 2013. doi:10.1002/jgrd.50184.

[28] Simon Unterstrasser. Properties of young contrails – a parametrisation based on large-eddy simulations. *Atmospheric Chemistry and Physics*, 16(4):2059–2082, February 2016. doi:10.5194/acp-16-2059-2016.

[29] Bernd Kärcher. Formation and radiative forcing of contrail cirrus. *Nature Communications*, December 2018. doi:10.1038/s41467-018-04068-0.

[30] T. Bräuer, C. Voigt, D. Sauer, S. Kaufmann, V. Hahn, M. Scheibe, H. Schlager, G. S. Diskin, J. B. Nowak, J. P. DiGangi, F. Huber, R. H. Moore, and B. E. Anderson. Airborne Measurements of Contrail Ice Properties—Dependence on Temperature and Humidity. *Geophysical Research Letters*, April 2021. doi:10.1029/2020GL092166.

[31] P. Spichtinger and K. M. Gierens. Modelling of cirrus clouds – Part 1a: Model description and validation. *Atmospheric Chemistry and Physics*, 9(2):685–706, January 2009. doi:10.5194/acp-9-685-2009.

[32] Ulrich Schumann, Robert Baumann, Darrel Baumgardner, Sarah T. Bedka, David P. Duda, Volker Freudenthaler, Jean-Francois Gayet, Andrew J. Heymsfield, Patrick Minnis, Markus Quante, Ehrhard Raschke, Hans Schlager, Margarita Vázquez-Navarro, Christiane Voigt, and Zhien Wang. Properties of individual contrails: a compilation of observations and some comparisons. *Atmospheric Chemistry and Physics*, 17(1):403–438, January 2017. doi:10.5194/acp-17-403-2017.

[33] Frank Holzäpfel. Probabilistic Two-Phase Wake Vortex Decay and Transport Model. *Journal of Aircraft*, 40(2):323–331, March 2003. doi:10.2514/2.3096.

[34] U. Schumann and T. Gerz. Turbulent Mixing in Stably Stratified Shear Flows. *Journal of Applied Meteorology and Climatology*, 34(1):33–48, January 1995. doi:10.1175/1520-0450-34.1.33.

[35] Simone Dietmüller. Dlr-pa/climaccf: Dataset update for GMDD. Zenodo, September 2022. doi:10.5281/zenodo.7074582.

[36] Simone Dietmüller, Sigrun Matthes, Katrin Dahlmann, Hiroshi Yamashita, Abolfazl Simorgh, Manuel Soler, Florian Linke, Benjamin Lührs, Maximilian Mendiguchia Meuser, Christian Weder, Volker Grewe, Feijia Yin, and Federica Castino. A python library for computing individual and merged non-CO₂ algorithmic climate change functions: CLIMaCCF V1.0. Preprint, Atmospheric sciences, October 2022. doi:10.5194/gmd-2022-203.

[37] D.I.A. Poll and U. Schumann. An estimation method for the fuel burn and other performance characteristics of civil transport aircraft in the cruise. Part 1 fundamental quantities and governing relations for a general atmosphere. *The Aeronautical Journal*, 125(1284):257–295, February 2021. doi:10.1017/aer.2020.62.

[38] D.I.A. Poll and U. Schumann. An estimation method for the fuel burn and other performance characteristics of civil transport aircraft during cruise: part 2, determining the aircraft's characteristic parameters. *The Aeronautical Journal*, 125(1284):296–340, February 2021. doi:10.1017/aer.2020.124.

[39] Marc E. J. Stettler, Adam M. Boies, Andreas Petzold, and Steven R. H. Barrett. Global Civil Aviation Black Carbon Emissions. *Environmental Science & Technology*, 47(18):10397–10404, September 2013. doi:10.1021/es401356v.

[40] Joseph P. Abrahamson, Joseph Zelina, M. Gurhan Andac, and Randy L. Vander Wal. Predictive Model Development for Aviation Black Carbon Mass Emissions from Alternative and Conventional Fuels at Ground and Cruise. *Environmental Science & Technology*, 50(21):12048–12055, November 2016. doi:10.1021/acs.est.6b03749.

[41] A Dopelheuer and M Lecht. Influence of engine performance on emission characteristics. *Gas Turbine Engine Combustion, Emissions and Alternative Fuels*, pages 12, 1998.

[42] Roger Teoh, Marc E.J. Stettler, Arnab Majumdar, Ulrich Schumann, Brian Graves, and Adam M. Boies. A methodology to relate black carbon particle number and mass emissions. *Journal of Aerosol Science*, 132:44–59, June 2019. doi:10.1016/j.jaerosci.2019.03.006.

[43] Kihong Park, David B. Kittelson, Michael R. Zachariah, and Peter H. McMurry. Measurement of Inherent Material Density of Nanoparticle Agglomerates. *Journal of Nanoparticle Research*, 6(2):267–272, June 2004. doi:10.1023/B:NANO.0000034657.71309.e6.

[44] Ramin Dastanpour and Steven N. Rogak. Observations of a Correlation Between Primary Particle and Aggregate Size for Soot Particles. *Aerosol Science and Technology*, 48(10):1043–1049, October 2014. doi:10.1080/02786826.2014.955565.

[45] Benjamin T. Brem, Lukas Durdina, Frithjof Siegerist, Peter Beyerle, Kevin Bruderer, Theo Rindlisbacher, Sara Rocci-Denis, M. Gurhan Andac, Joseph Zelina, Olivier Penanhoat, and Jing Wang. Effects of Fuel Aromatic Content on Nonvolatile Particulate Emissions of an In-Production Aircraft Gas Turbine. *Environmental Science & Technology*, 49(22):13149–13157, November 2015. doi:10.1021/acs.est.5b04167.

[46] Doug DuBois and Gerald C. Paynter. "Fuel Flow Method2" for Estimating Aircraft Emissions. *SAE Transactions*, 115:1–14, 2006. arXiv:44657657.

[47] Zhian Sun and Lawrie Rikus. Parametrization of effective sizes of cirrus-cloud particles and its verification against observations: RADIATIVE FOR CING AND CLIMATE SENSITIVITY. *Quarterly Journal of the Royal Meteorological Society*, 125(560):3037–3055, October 1999. doi:10.1002/qj.49712556012.

[48] Philipp Reutter, Patrick Neis, Susanne Rohs, and Bastien Sauvage. Ice supersaturated regions: properties and validation of ERA-Interim reanalysis with IAGOS in situ water vapour measurements. *Atmospheric Chemistry and Physics*, 20(2):787–804, January 2020. doi:10.5194/acp-20-787-2020.

[49] F. Anthony Eckel and Michael K. Walters. Calibrated Probabilistic Quantitative Precipitation Forecasts Based on theMRF Ensemble. *Weather and Forecasting*, 13(4):1132–1147, December 1998. doi:10.1175/1520-0434(1998)013<1132:CPQPFB>2.0.CO;2.

[50] Wikipedia contributors. International Standard Atmosphere. https://en.wikipedia.org/w/index.php?title=International_Standard_Atr 2023.

[51] UO Solar Radiation Monitoring Laboratory. UO SRML: Solar radiation basics. http://solardat.uoregon.edu/SolarRadiationBasics.html, 2022.

[52] G. W. Paltridge, C. Martin R. Platt, and C. M. R. Platt. *Radiative Processes in Meteorology and Climatology*. Elsevier Scientific Publishing Company, 1976. ISBN 978-0-444-41444-1.

[53] D. Sonntag. Advancements in the field of hygrometry. *Meteorologische Zeitschrift*, 3(2):51–66, May 1994. doi:10.1127/metz/3/1994/51.

[54] Nicholas Cumpsty and Andrew Heyes. *Jet Propulsion*. Cambridge University Press, July 2015. ISBN 978-1-107-51122-4.

[55] D. K. Wasiuk, M. H. Lowenberg, and D. E. Shallcross. An aircraft performance model implementation for the estimation of global and regional commercial aviation fuel burn and emissions. *Transportation Research Part D: Transport and Environment*, 35:142–159, March 2015. doi:10.1016/j.trd.2014.11.022.

[56] Eurocontrol. USER MANUAL FOR THE BASE OF Aircraft DATA (BADA) REVISION 3.8. April 2010. doi:10.1163/1570-6664_iyb_SIM_org_39214.

[57] Wikipedia contributors. Azimuth. https://en.wikipedia.org/w/index.php?title=Azimuth&oldid=1131548993, 2023.

[58] Wikipedia contributors. Solar zenith angle. https://en.wikipedia.org/w/index.php?title=Solar_zenith_angle&oldid=1141660912, 2023.

[59] Calculate distance and bearing between two Latitude/Longitude points using haversine formula in JavaScript. https://www.movable-type.co.uk/scripts/latlong.html.

[60] NOAA. Solar Calculation Details. https://gml.noaa.gov/grad/solcalc/calcdetails.html.

[61] John A. Duffie and William A. Beckman. *Solar Engineering of Thermal Processes*. Wiley, 1991. ISBN 978-0-471-51056-7.

[62] Wikipedia contributors. Barometric formula. https://en.wikipedia.org/w/index.php?title=Barometric_formula&oldid=1144039200, 2023.

[63] David Megginson. Open-data downloads for OurAirports. April 2023.

[64] Rémi Chevallier, Marc Shapiro, Zebediah Engberg, Manuel Soler, and Daniel Delahaye. Linear Contrails Detection, Tracking and Matching with Aircraft Using Geostationary Satellite and Air Traffic Data. *Aerospace*, 10(7):578, July 2023. doi:10.3390/aerospace10070578.

[65] Alejandra Martín Frías, \relax FLIGHTKEYS GmbH, and Manuel Soler. Enhancing environmental sustainability in aviation: an implementation of contrail mitigation strategies in commercial flight dispatching. In *ATM Seminar*. 2023.

[66] Scott Geraedts, Erica Brand, Thomas R. Dean, Sebastian Eastham, Carl Elkin, Zebediah Engberg, Ulrike Hager, Ian Langmore, Kevin McCloskey, Joe Yue-Hei Ng, John C. Platt, Tharun Sankar, Aaron Sarna, Marc Shapiro, and Nita Goyal. A scalable system to measure contrail formation on a per-flight basis. August 2023. arXiv:2308.02707.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X