traffic Documentation

Xavier Olive

Jan 31, 2020

Contents

1 Installation 3

2 Quickstart 7

3 Core structure 25

4 Sources of data 27

5 Algorithms 47

6 Exporting and storing data 55

7 Command line interface 57

8 Graphical user interface 59

9 Plugins 67

10 Scenarios and use cases 71

11 Publications 87

Index 113

Source code on github

The traffic library helps working with common sources of air traffic data.

Its main purpose is to provide data analysis methods commonly applied to trajectories and airspaces. When a specific function is not provided, the access to the underlying structure is direct, through an attribute pointing to a pandas dataframe.

The library also offers facilities to parse and/or access traffic data from open sources of ADS-B traffic like the OpenSky Network or Eurocontrol DDR files. It is designed to be easily extendable to other sources of data.

CHAPTER 1

Installation

The traffic library makes an intensive use of pandas DataFrames and of the shapely GIS library.

The library relies on requests for calls to REST APIs. paramiko implements the SSH protocol in Pure Python, giving access to SSH connection independently of the operating system.

Static visualisation tools are accessible with Matplotlib through the cartotools library, which leverages access to more projections and to data from OpenStreetMap. More dynamic visualisations in Jupyter Lab are accessible thanks to the altair and ipyleaflet libraries; other exports to various formats (including CesiumJS or Google Earth) are also available.

1.1 Installation

We recommend cloning the latest version from the repository before installing it.

```
git clone https://github.com/xoolive/traffic
cd traffic/
python setup.py install
```

If you are not comfortable with that option, you can always install the latest release:

```
pip install traffic
```

Warning: *cartopy* and *shapely* have strong dependencies to dynamic libraries which may not be available on your system by default. If possible, install Anaconda, then:

```
conda install cartopy shapely
# then _either_ with pip (stable version)
pip install traffic
# _or_ from sources (dev version)
python setup.py install
```

1.2 Troubleshooting

The following reported problems are not about the traffic library per se. Some are related to limitations set by dependencies.

1.2.1 I can't access resources from the Internet as I am behind a proxy

All network accesses are made with the requests library (or the paramiko library for the Impala shell). Usually, if your environment variables are properly set, you should not encounter any particular proxy issue. However, there are always situations where it may help to manually force the proxy settings in the configuration file.

Edit your configuration file (you may find where it is located in traffic.config_file) and add the following section. Uncomment and edit options according to your network configuration.

```
## This sections contains all the specific options necessary to fit your
## network configuration (including proxy and ProxyCommand settings)
[network]

## input here the arguments you need to pass as is to requests
# http.proxy = http://proxy.company:8080
# https.proxy = http://proxy.company:8080
# http.proxy = socks5h://localhost:1234
# https.proxy = socks5h://localhost:1234

## input here the ProxyCommand you need to login to the Impala Shell
## WARNING:
##    Do not use %h and %p wildcards.
##    Write data.opensky-network.org and 2230 explicitly instead
# ssh.proxycommand = ssh -W data.opensky-network.org:2230 proxy_ip:proxy_port
# ssh.proxycommand = connect.exe -H proxy_ip:proxy_port data.opensky-network.org 2230
```

1.2.2 Python crashes when I try to reproduce plots in the documentation

(or, "My Jupyter kernel crashes…")

There must be something wrong with your Cartopy and/or shapely installation. These libraries strongly depend on the `geos` and `proj` libraries. You must have shapely and Cartopy versions matching the correct versions of these libraries.

The problem is sometimes hard to understand, and you may end up fixing it without really knowing how.

If you don't know how to install these dependencies, start with a **fresh** Anaconda distribution and install the following libraries *the conda way*:

```
conda install cartopy shapely
```

If it still does not work, try something along:

```
conda uninstall cartopy shapely
pip uninstall cartopy shapely
# be sure to remove all previous versions before installing again
conda install cartopy shapely
```

If it still does not work, try again with:

```
conda uninstall cartopy shapely
pip uninstall cartopy shapely
# this forces the recompilation of the packages
pip install --no-binary :all: cartopy shapely
```

1.2.3 Widgets do not display in Jupyter Lab or Jupyter Notebook

After executing a cell, you may see one of the following output:

```
A Jupyter Widget
# or
Error displaying widget
# or
HBox(children=(IntProgress(value=0, max=1), HTML(value='')))
# or
Map(basemap={'url': 'https://{s}.tile.openstreetmap.org/...
```

You will need to activate the widgets extensions:

- with Jupyter Lab:

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
jupyter labextension install jupyter-leaflet
jupyter labextension install keplergl-jupyter
```

- with Jupyter Notebook:

```
jupyter nbextension enable --py --sys-prefix widgetsnbextension
jupyter nbextension enable --py --sys-prefix ipyleaflet
jupyter nbextension enable --py --sys-prefix keplergl
```

CHAPTER 2

Quickstart

The motivation for this page/notebook is to take the reader through all basic functionalities of the traffic library. We will cover:

1. a *basic introduction* about `Flight` and `Traffic` structures;
2. how to produce *visualisations* of trajectory data;
3. how to access basic *sources of data*;
4. a simple *use case* to select trajectories landing at Toulouse airport;
5. an introduction to *declarative descriptions* of data preprocessing through lazy iteration.

This page is also available as a notebook which can be downloaded and executed locally; or loaded and executed in Google Colab.

2.1 Basic introduction

The traffic library provides natural methods and attributes that can be applied on trajectories and collection of trajectories, all represented as pandas DataFrames.

2.1.1 The *Flight* structure

Flight is the core class offering representations, methods and attributes to single trajectories. A comprehensive description of the API is available here.

Sample trajectories are provided in the library: `belevingsvlucht` is one of them, context is explained here.

```
from traffic.data.samples import belevingsvlucht

belevingsvlucht
```

Among available attributes, you may want to access:

- its `callsign` (the identifier of the flight displayed on ATC screens);
- its transponder unique identification number (`icao24`);
- its `registration` number (tail number);
- its `typecode` (i.e. the model of aircraft).

```
(
    belevingsvlucht.callsign,
    belevingsvlucht.icao24,
    belevingsvlucht.registration,
    belevingsvlucht.typecode,
)

# ('TRA051', '484506', 'PH-HZO', 'B738')
```

Methods are provided to select relevant parts of the flight, e.g. based on timestamps.

The `start` and `stop` attributes refer to the timestamps of the first and last recorded samples. Note that all timestamps are by default set to universal time (UTC) as it is common practice in aviation.

```
(belevingsvlucht.start, belevingsvlucht.stop)

# (Timestamp('2018-05-30 15:21:38+0000', tz='UTC'),
#  Timestamp('2018-05-30 20:22:56+0000', tz='UTC'))
```

```
first30 = belevingsvlucht.first(minutes=30)
after19 = belevingsvlucht.after("2018-05-30 19:00", strict=False)

# Notice the "strict" comparison (>) vs. "or equal" comparison (>=)
print(f"between {first30.start:%H:%M:%S} and {first30.stop:%H:%M:%S}")
print(f"between {after19.start:%H:%M:%S} and {after19.stop:%H:%M:%S}")

# between 15:21:38 and 15:51:37
# between 19:00:00 and 20:22:56
```

```
between1920 = belevingsvlucht.between(
    "2018-05-30 19:00", "2018-05-30 20:00"
)
between1920
```

The underlying dataframe is always accessible.

```
between1920.data.head()
```

2.1.2 The *Traffic* structure

Traffic is the core class to represent collections of trajectories, which are all flattened in the same pandas DataFrame. A comprehensive description of the API is available here.

We will demonstrate here with a sample of ADS-B data from the OpenSky Network.

The basic representation of a *Traffic* object is a summary view of the data: the structure tries to infer how to separate trajectories in the data structure based on customizable heuristics, and returns a number of sample points for each trajectory.

```
from traffic.data.samples import quickstart

quickstart
```

Traffic offers the ability to **index** and **iterate** on all flights contained in the structure. *Traffic* will use either:

- a combination of `timestamp`, `icao24` (aircraft identifier) and `callsign` (mission identifier); or
- a customizable flight identifier (`flight_id`);

as a way to separate and identify flights.

Indexation will be made on either of `icao24`, `callsign` (or `flight_id` if available).

```
quickstart["AFR27GH"]  # on callsign
quickstart["393320"]  # on icao24
```

A subset of trajectories can also be selected if a list is passed an index:

```
quickstart[["AFR27GH", "HOP87DJ"]]
```

In many cases, `flight_id` are more convenient to access specific flights yielded by iteration. We may construct custom `flight_id`:

```
from traffic.core import Traffic

quickstart_id = Traffic.from_flights(
    flight.assign(flight_id=f"{flight.callsign}_{i:03}")
    for i, flight in enumerate(quickstart)
)
quickstart_id
```

or use the available `.assign_id()` method, which is implemented exactly that way.
(We will explain `eval()` further below)

```
quickstart.assign_id().eval()
```

2.1.3 Saving and loading data

Some processing operations are computationally expensive and time consuming. Therefore, it may be relevant to store intermediate results in files for sharing and reusing purposes.

One option is to store *Traffic* and *Flight* underlying DataFrames in pickle format. Details about storage formats are presented here.

```
quickstart_id.to_pickle("quickstart_id.pkl")
```

```
from traffic.core import Traffic

# load from file again
quickstart_id = Traffic.from_file("quickstart_id.pkl")
```

2.2 Visualisation of data

traffic offers facilities to leverage the power of common visualisation renderers including Cartopy, a map plotting library built around Matplotlib, and Altair.

More visualisation renderers such as Leaflet are available as plugins.

2.2.1 Visualisation of trajectories

When you choose to plot trajectories on a map, you have to make a choice concerning how to represent points at the surface of a sphere (more precisely, an oblate spheroid) on a 2D plane. This transformation is called a projection.

The choice of the right projection depends on the data. The most basic projection (sometimes wrongly referred to as *no projection*) is the `PlateCarree()`, when you plot latitude on the y-axis and longitude on the x-axis. The famous `Mercator()` projection distorts the latitude so as lines with constant bearing appear as straight lines. Conformal projections are also convenient when plotting smaller areas (countries) as they preserve distances (locally).

Many countries define official projections to produce maps of their territory. In general, they fall either in the conformal or in the Transverse Mercator category. `Lambert93()` projection is defined over France, `GaussKruger()` over Germany, `Amersfoort()` over the Netherlands, `OSGB()` over the British Islands, etc.

When plotting trajectories over Western Europe, `EuroPP()` is a decent choice.

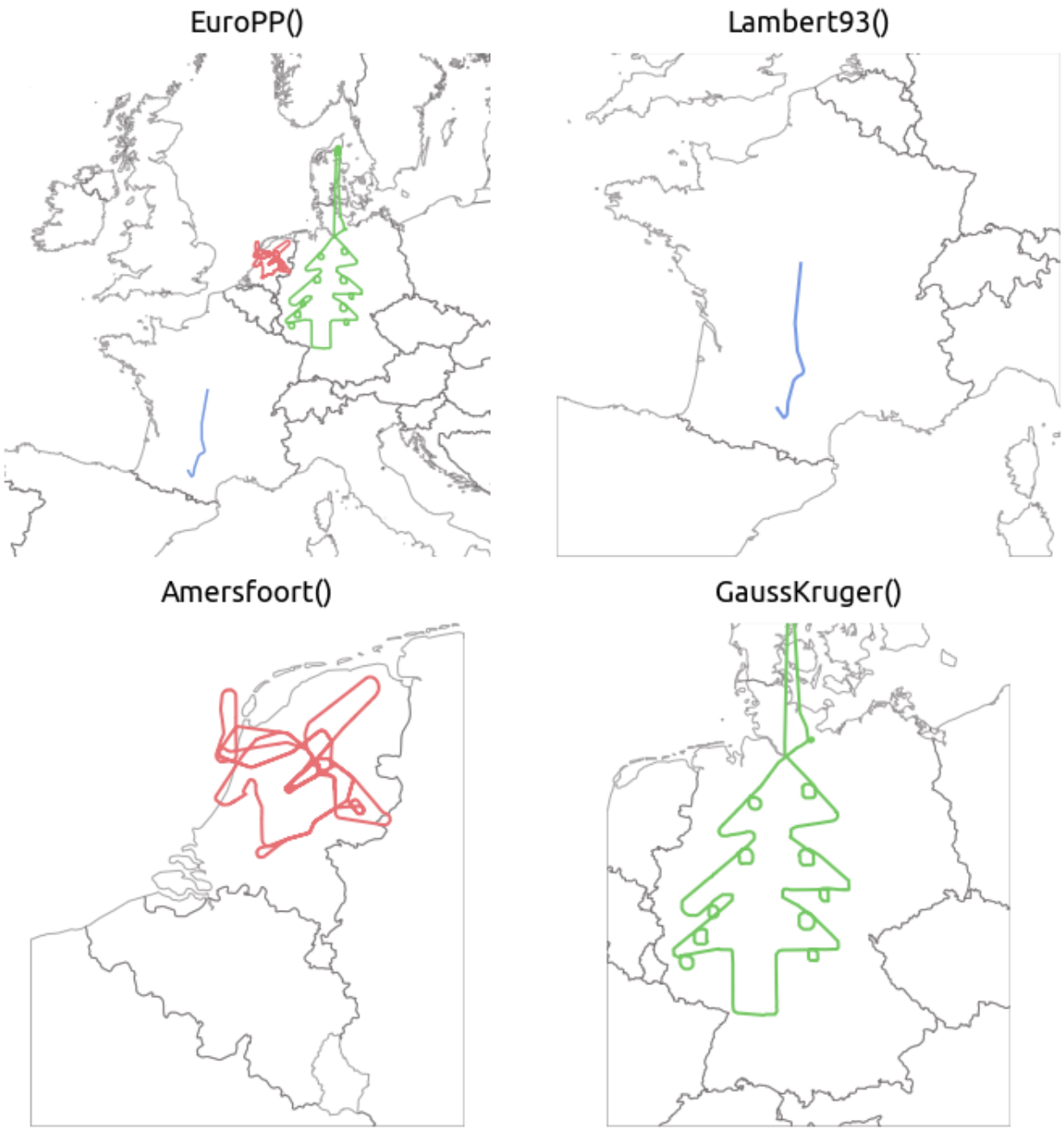
```
from traffic.data.samples import airbus_tree
```

```
%matplotlib inline
import matplotlib.pyplot as plt

from traffic.core.projection import Amersfoort, GaussKruger, Lambert93, EuroPP
from traffic.drawing import countries

with plt.style.context("traffic"):
    fig = plt.figure()

    # Choose the projection type
    ax0 = fig.add_subplot(221, projection=EuroPP())
    ax1 = fig.add_subplot(222, projection=Lambert93())
    ax2 = fig.add_subplot(223, projection=Amersfoort())
    ax3 = fig.add_subplot(224, projection=GaussKruger())

    for ax in [ax0, ax1, ax2, ax3]:
        ax.add_feature(countries())
        # Maximum extent for the map
        ax.set_global()
        # Remove border and set transparency for background
        ax.outline_patch.set_visible(False)
        ax.background_patch.set_visible(False)

    # Flight.plot returns the result from Matplotlib as is
    # Here we catch it to reuse the color of each trajectory
    ret, *_ = quickstart["AFR27GH"].plot(ax0)
    quickstart["AFR27GH"].plot(
        ax1, color=ret.get_color(), linewidth=2
    )

    ret, *_ = belevingsvlucht.plot(ax0)
```

(continues on next page)

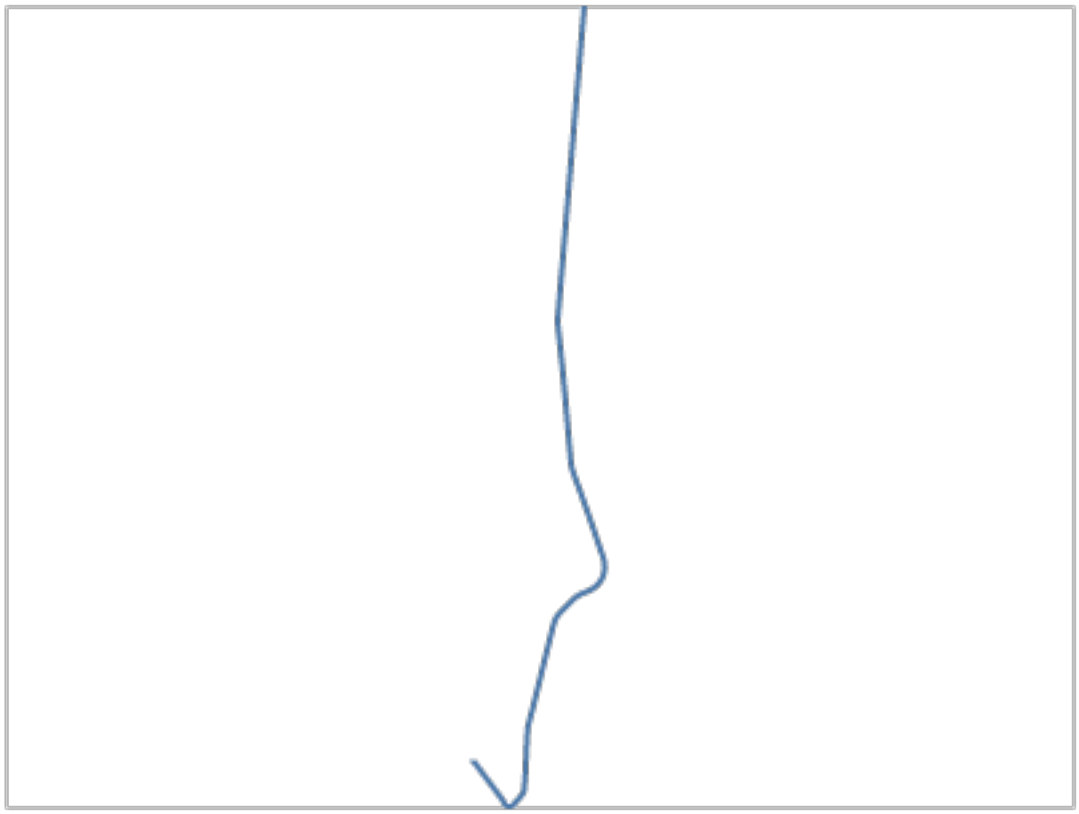
(continued from previous page)

```
    belevingsvlucht.plot(
        ax2, color=ret.get_color(), linewidth=2
    )

    ret, *_ = airbus_tree.plot(ax0)
    airbus_tree.plot(
        ax3, color=ret.get_color(), linewidth=2
    )

    # We reduce here the extent of the EuroPP() map
    # between 8°W and 18°E, and 40°N and 60°N
    ax0.set_extent((-8, 18, 40, 60))

    params = dict(fontname="Ubuntu", fontsize=18, pad=12)

    ax0.set_title("EuroPP()", **params)
    ax1.set_title("Lambert93()", **params)
    ax2.set_title("Amersfoort()", **params)
    ax3.set_title("GaussKruger()", **params)

    fig.tight_layout()
```



Altair API is not very mature yet with geographical data, but basic visualisations are possible.

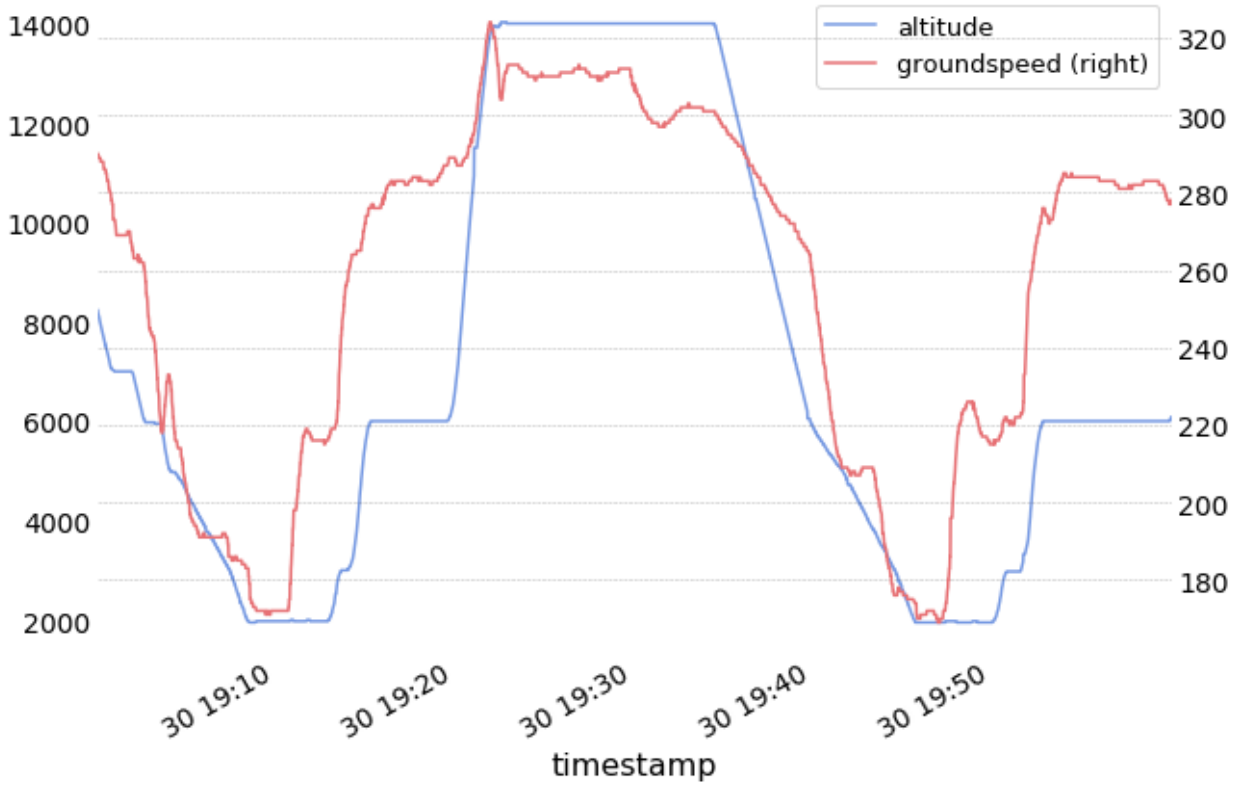
```
# Mercator projection is the default one with Altair
quickstart["AFR27GH"].geoencode().project(type="mercator")
```



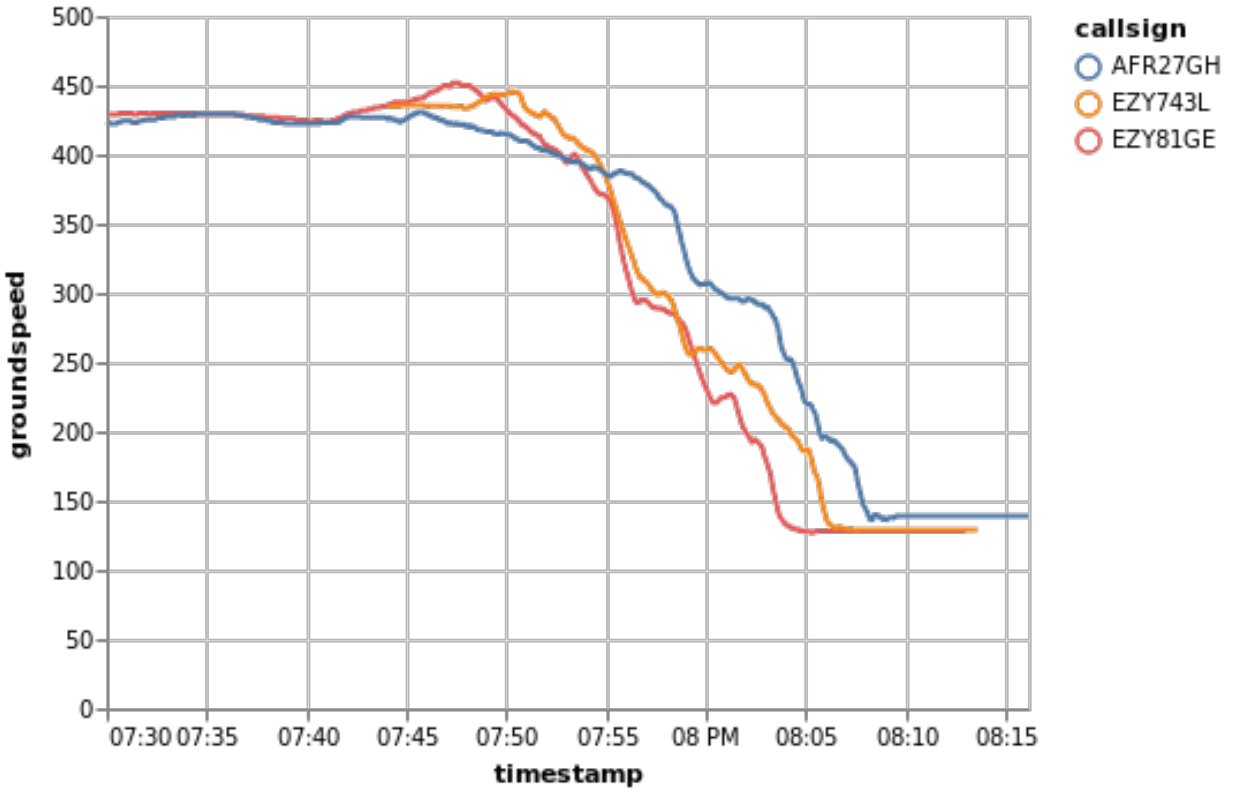
2.2.2 Visualisation of time series

Facilities are provided to plot time series, after a basic cleaning of data (remove NaN values), both with Matplotlib and Altair. The traffic style context offers a convenient first style to customise further.

```
with plt.style.context("traffic"):
    fig, ax = plt.subplots(figsize=(10, 7))
    between1920.plot_time(
        ax, y=["altitude", "groundspeed"], secondary_y=["groundspeed"]
    )
```




```
(
    quickstart["EZY81GE"].encode("groundspeed")
    + quickstart["EZY743L"].encode("groundspeed")
    + quickstart["AFR27GH"].encode("groundspeed")
)
```



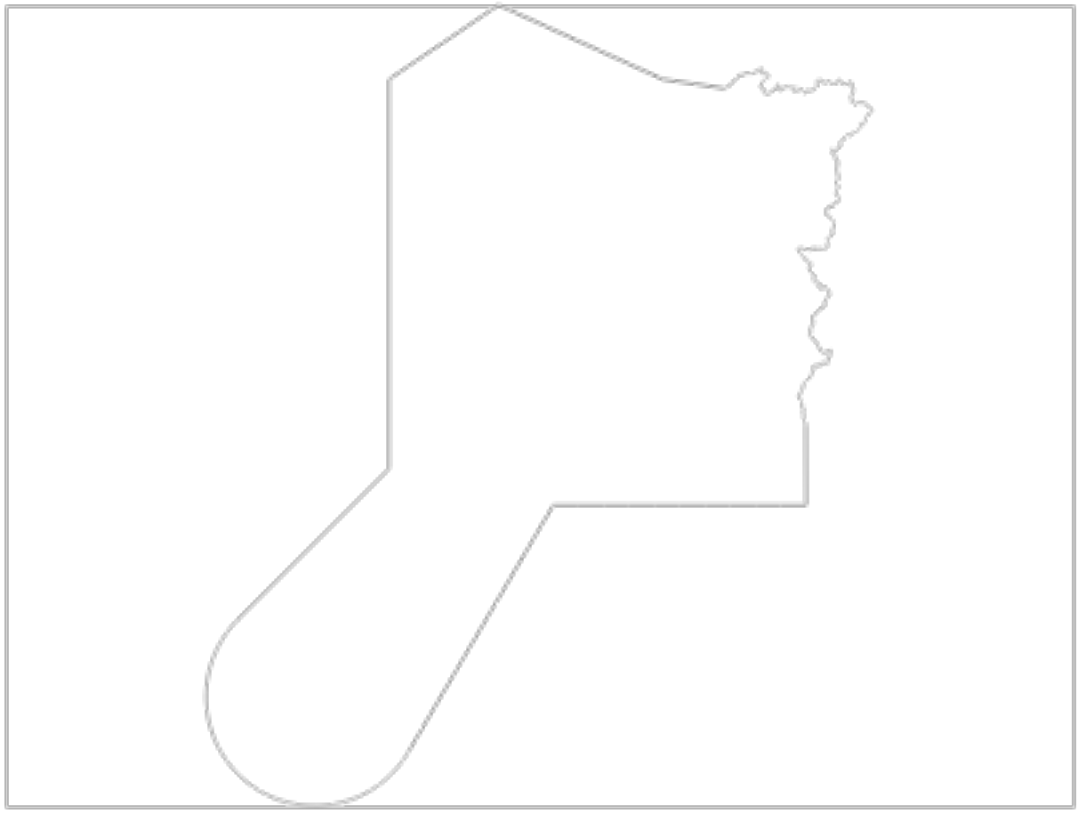
2.3 Sources of data

Basic navigational data are embedded in the library, together with parsing facilities for most common sources of information, with a main focus on Europe at the time being.

Airspaces are a key element of aviation: they are regulated by specific rules, whereby navigation is allowed to determined types of aircraft meeting strict requirements. Such volumes, assigned to air traffic controllers to ensure the safety of flights and proper separation between aircraft are most commonly described as a combination of extruded polygons. Flight Information Regions (FIR) are one of the basic form of airspaces.

A non official list of European FIRs, airports, navaids and airways is available in the traffic library (Details here).

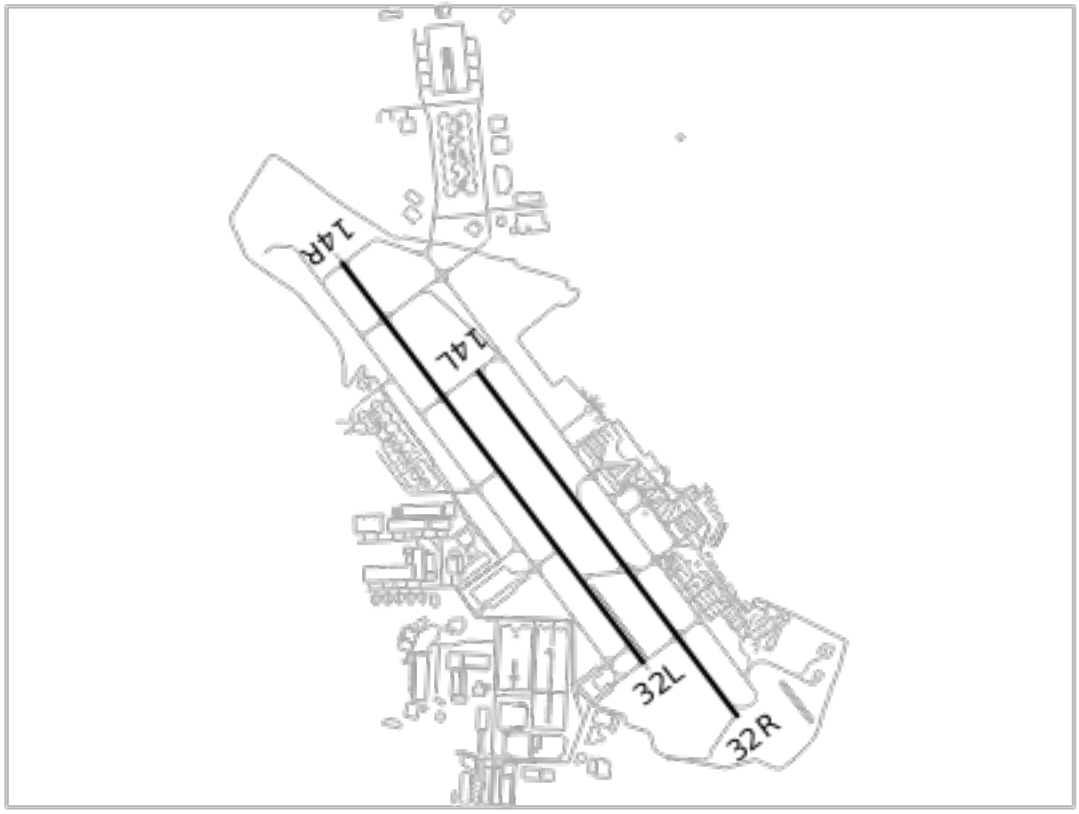
```
from traffic.data import eurofirs

# LISBOA FIR
eurofirs["LPPC"].geoencode()
```



```
from traffic.data import airports

airports["AMS"]
```

The details of airport representations are also available (fetched from OpenStreetMap) in their Matplotlib and Altair representation.

```
airports["LFBO"].geoencode(runways=True, labels=True)
```

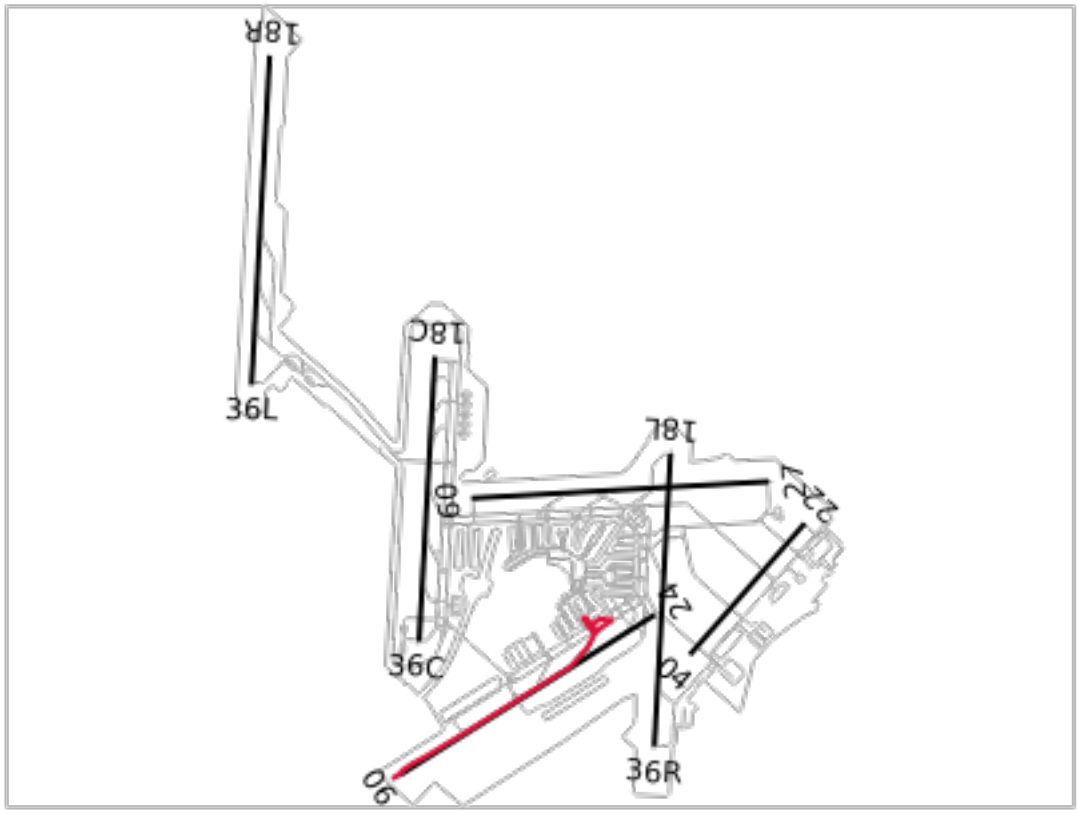


Intersections can be computed between trajectories and geometries (airports, airspaces). `Flight.intersects()` provides a fast boolean test; `Flight.clip()` trims the trajectory between the first point of entry in and last point of exit from the 2D footprint of the geometry.

```
belevingsvlucht.intersects(airports["EHAM"])
# True
```

Of course, all these methods can be chained.

```
(
    airports["EHAM"].geoencode(runways=True, labels=True)
    + belevingsvlucht.last(hours=1)
    .clip(airports["EHAM"])
    .geoencode()
    .mark_line(color="crimson")
)
```



2.4 A simple use case

The following use case showcases various preprocessing methods that can be chained to select all trajectories landing at Toulouse airport. We will need the coordinates of Toulouse Terminal Maneuvering Area (TMA) which is available in Eurocontrol AIRAC files.

You may not be entitled access to these data but the coordinates of Toulouse TMA are public, so we provide them in this library for the sake of this example.

If you have set the configuration for the AIRAC files (details here), you may uncomment the following cell.

```
# from traffic.data import nm_airspaces
# lfbo_tma = nm_airspaces["LFBOTMA"]
```

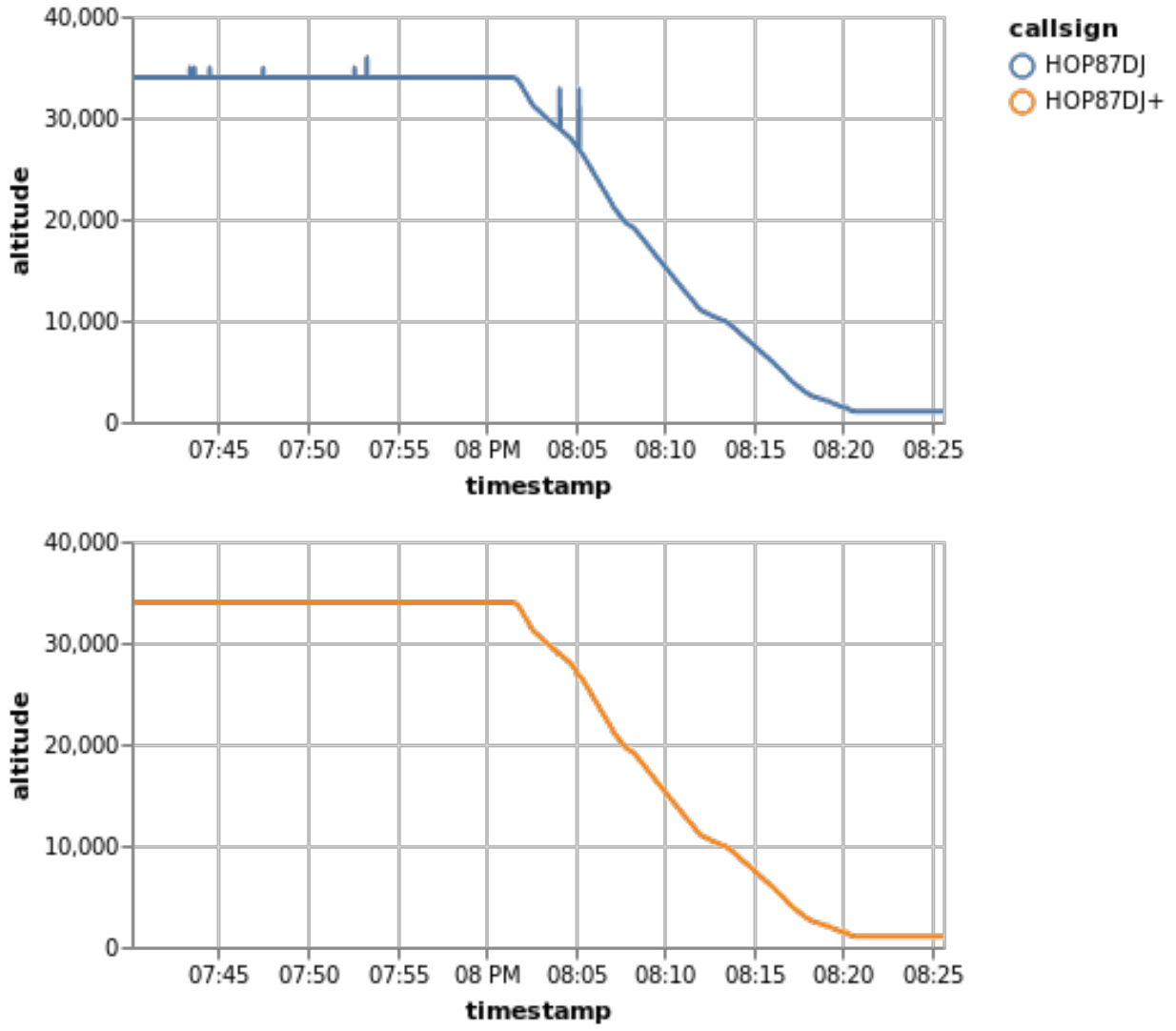
Since you may not be entitled access to these data and coordinates of Toulouse TMA are public, we provide them in this library for the sake of this example.

```
from traffic.data.samples import lfbo_tma

lfbo_tma
```

A first common necessary prepocessing concerns filtering of faulty values, esp. when data comes for a wide network of ADS-B sensors such as the OpenSky Network. A common pattern in such data is spikes in various signals, esp. altitude. Some filtering methods have been developped to take this faulty values out:

```
hop87dj = quickstart["HOP87DJ"]
# Set a different callsign and identify signals on the visualisation
filtered = hop87dj.filter().assign(callsign="HOP87DJ+")
```

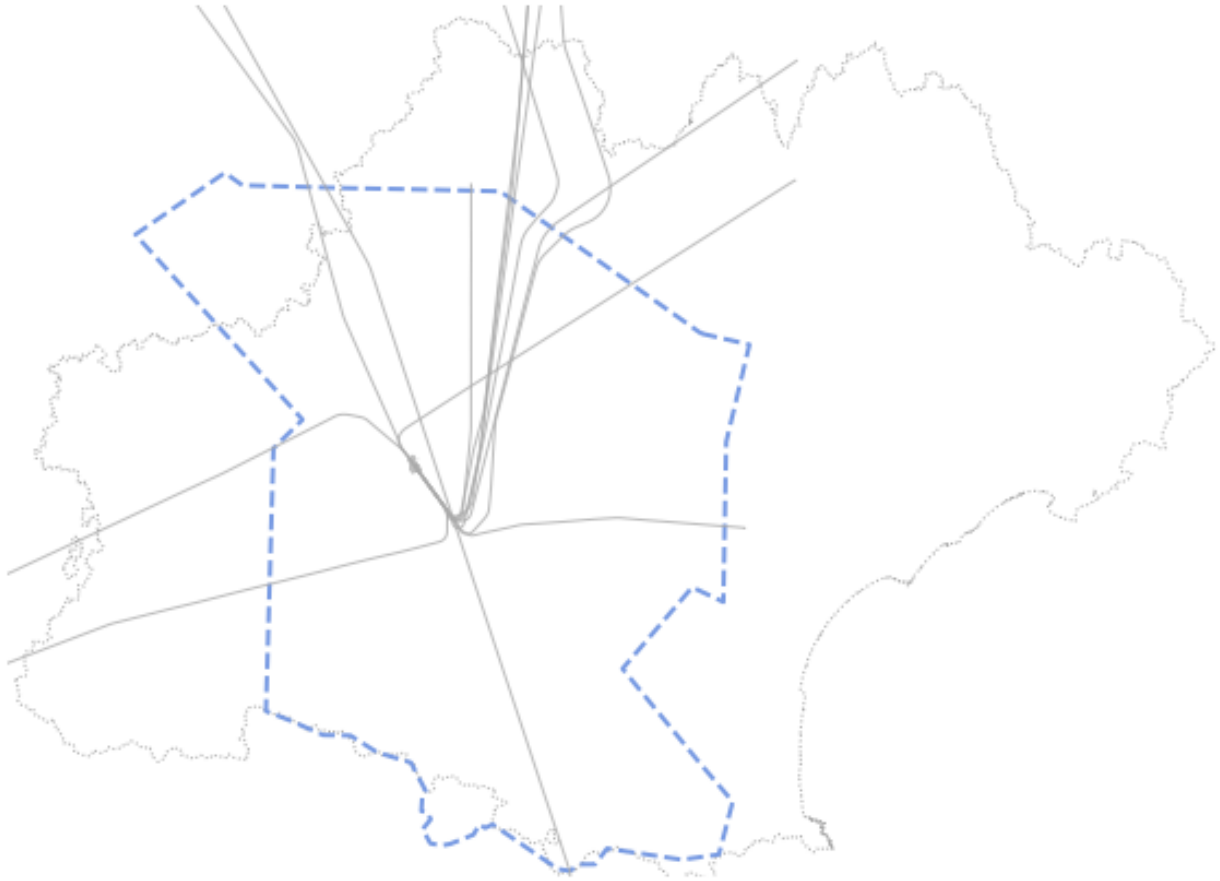
```
import altair as alt

# Let's define a common y-scale for both flights
scale = alt.Scale(domain=(0, 40000))

visualisations = [
    (flight.encode(alt.Y("altitude", scale=scale)).properties(height=180, width=360))
    for flight in [hop87dj, filtered]
]

alt.vconcat(*visualisations)
```



Let's select first trajectories intersecting Toulouse TMA, filter signals, then plot the results.

```
# A progressbar may be convenient...
landing_trajectories = []

for flight in quickstart:
    if flight.intersects(lfbo_tma):
        filtered = flight.filter()
        landing_trajectories.append(filtered)

t_tma = Traffic.from_flights(landing_trajectories)
t_tma
```

```
from traffic.drawing import location

with plt.style.context("traffic"):
    fig, ax = plt.subplots(subplot_kw=dict(projection=Lambert93()))
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

    # We may add contours from OpenStreetMap
    # (Occitanie is the name of the administrative region)
    location("Occitanie").plot(ax, linestyle="dotted")
    ax.set_extent("Occitanie")

    # Plot the airport, the TMA
    airports["LFBO"].plot(ax)
    lfbo_tma.plot(ax, linewidth=2, linestyle="dashed")

    # and the resulting traffic
    t_tma.plot(ax)
```



There is still one trajectory which does not seem to be coming to Toulouse airport. Also, we actually wanted to select landing trajectories. Let's only select trajectories coming below 10,000 ft and with an average vertical speed below 1,000 ft/min.

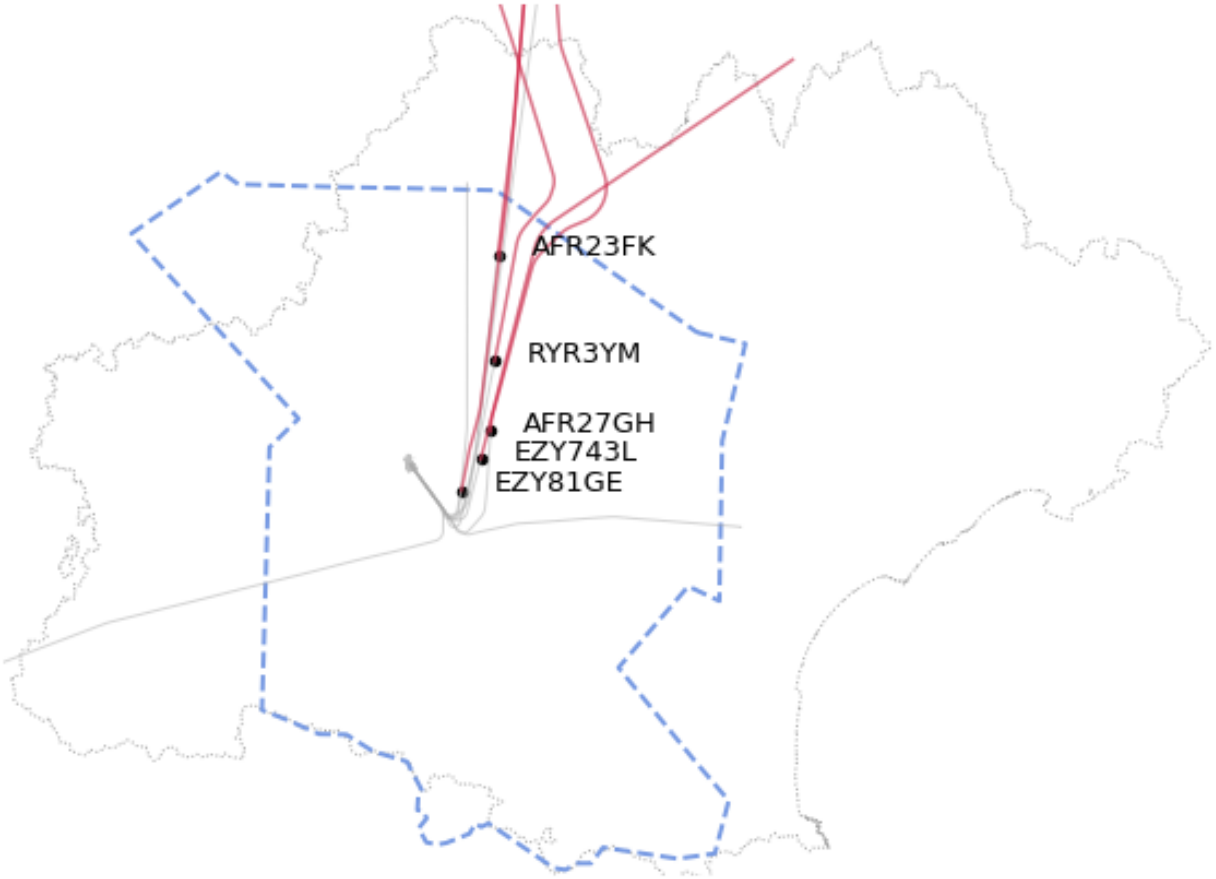
```
landing_trajectories = []

for flight in quickstart:
    if flight.intersects(lfbo_tma):
        filtered = flight.filter()
        if filtered.min("altitude") < 10_000:
            if filtered.mean("vertical_rate") < - 500:
                landing_trajectories.append(filtered)

t_tma = Traffic.from_flights(landing_trajectories)
t_tma
```

```
from traffic.drawing import location

with plt.style.context("traffic"):
    fig, ax = plt.subplots(subplot_kw=dict(projection=Lambert93()))
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

    # We may add contours from OpenStreetMap
    # (Occitanie is the name of the administrative region)
    location("Occitanie").plot(ax, linestyle="dotted")
```

(continues on next page)

(continued from previous page)

```
    ax.set_extent("Occitanie")

    # Plot the airport, the TMA
    airports["LFBO"].plot(ax)
    lfbo_tma.plot(ax, linewidth=2, linestyle="dashed")

    # and the resulting traffic
    t_tma.plot(ax, alpha=0.5)

    for flight in t_tma:
        flight_before = flight.before("2017-07-16 20:00")

        # Avoid unnecessary noise on the map
        if 1000 < flight_before.at().altitude < 20000:

            flight_before.plot(ax, alpha=0.5, color="crimson")
            flight_before.at().plot(ax, s=20, text_kw=dict(s=flight.callsign))
```



2.5 Lazy iteration

Basic operations on Flights define a little language which enables to express programmatically any kind of preprocessing.

The downside with programmatic preprocessing is that it may become unnecessarily complex and nested with loops

and conditions to express even basic treatments. As a reference, here is the final code we came to:

```
# unnecessary list
landing_trajectories = []

for flight in quickstart:
    # loop
    if flight.intersects(lfbo_tma):
        # first condition
        filtered = flight.filter()
        if filtered.min("altitude") < 10_000:
            # second condition
            if filtered.mean("vertical_rate") < 1_000:
                # third condition
                landing_trajectories.append(filtered)

t_tma = Traffic.from_flights(landing_trajectories)
```

As we define operations on single trajectories, we may also want to express operations, like filtering or intersections on collections of trajectories rather than single ones.

```
# Traffic.filter() would be
Traffic.from_flights(
    flight.filter() for flight in quickstart
)

# Traffic.intersects(airspace) would be
Traffic.from_flights(
    flight for flight in quickstart
    if flight.intersects(airspace)
)
```

Such implementation would be very inefficient because Python would constantly start a new iteration for every single operation that is chained. To avoid this, a mechanism of **lazy iteration** has been implemented:

- Most `Flight` methods returning a `Flight`, a boolean or `None` can be stacked on `Traffic` structures;
- When such a method is stacked, it is **not** evaluated, just pushed for later evaluation;
- The final `.eval()` call starts one single iteration and apply all stacked method to every `Flight` it can iterate on.
- If one of the methods returns `False` or `None`, the `Flight` is discarded;
- If one of the methods returns `True`, the `Flight` is passed as is not the next method.

```
# A custom grammar can be defined
# here we define conditions for detecting landing trajectories

def landing_trajectory(flight: "Flight") -> bool:
    return flight.min("altitude") < 10_000 and flight.mean("vertical_rate") < -500


t_tma = (
    quickstart
    # non intersecting flights are discarded
    .intersects(lfbo_tma)
    # intersecting flights are filtered
    .filter()
```

(continues on next page)

(continued from previous page)

```
    # filtered flights not matching the condition are discarded
    .filter_if(landing_trajectory)
    # final multiprocessed evaluation (4 cores) through one iteration
    .eval(max_workers=4)
)
t_tma
```

CHAPTER 3

Core structure

The *traffic* library is based on three main core classes for handling:

- aircraft trajectories are accessible through traffic.core.Flight;
- collections of aircraft trajectories are traffic.core.Traffic;
- airspaces and sectors are represented with traffic.core.Airspace.

Flight and Traffic are wrappers around pandas DataFrames with relevant efficiently implemented methods for trajectory analysis. Airspaces take advantage of shapely Geometries for geometrical analysis.

Contents

3.1 traffic.core.Flight

3.2 traffic.core.Traffic

3.3 traffic.core.Airspace

CHAPTER 4

Sources of data

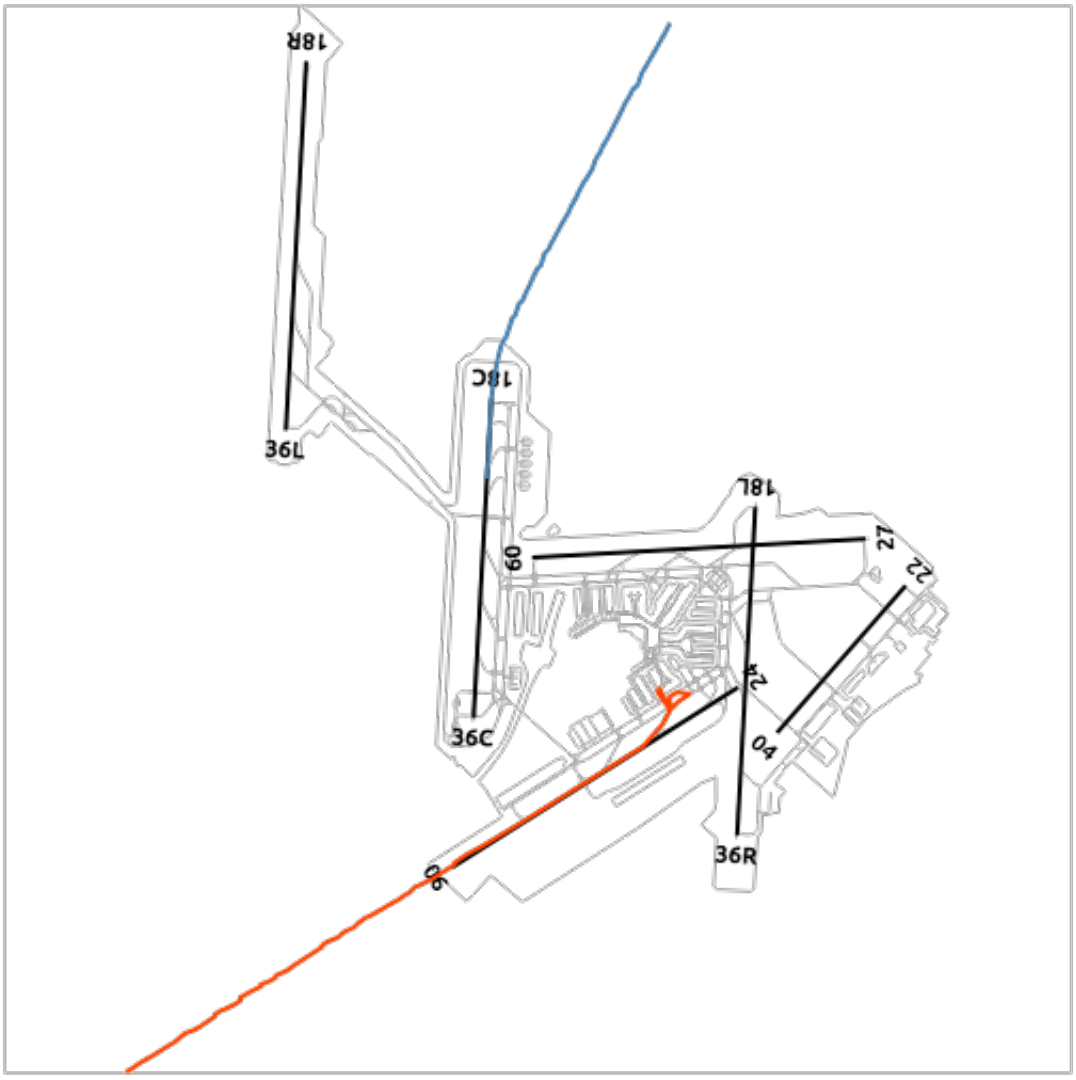
4.1 Airports and runways

Airports offer special display outputs in Jupyter Notebook based on their geographic footprint.

```
from traffic.data import airports
airports['EHAM']
```

It is also possible to benefit from the Altair geographical representation.

```
from traffic.data import airports
# see https://traffic-viz.github.io/samples.html if any issue on import
from traffic.data.samples import belevingsvlucht

(
    airports["EHAM"].geoencode(runways=True, labels=True)
    + belevingsvlucht.first(minutes=1).geoencode().mark_line(color="steelblue")
    + belevingsvlucht.last(minutes=6).geoencode().mark_line(color="orangered")
).configure_text(font="Ubuntu", fontWeight="bold")
```



4.2 Aircraft database

4.3 Navigational beacons

4.4 Airways, ATS routes

4.5 Sample trajectories

A bundle of sample trajectories covering various situations has been included in the library for reference, testing and for providing a baseline to compare the performance of various algorithms.

All sample trajectories are available in the *traffic.data.samples* module. The import automatically dispatch to Flight or Traffic according to the nature of the data.

Warning: The dynamic import of trajectories from the *traffic.data.samples* module is only available in Python versions above 3.7.

```
# Python >= 3.7
from traffic.data.samples.featured import belevingsvlucht
from traffic.data.samples import belevingsvlucht  # no ambiguity

# Python >= 3.6
from traffic.data.samples import featured, get_sample
belevingsvlucht = get_sample(featured, 'belevingsvlucht')
```

Note: A subset of the sample trajectories are presented on this page. Other parts of the documentation (e.g. calibration flights or trajectory clustering) may refer to other available sample trajectories.

4.5.1 Belevingsvlucht

https://www.belevingsvlucht.nl/

On May 30th 2018, test flights were conducted along future routes arriving and departing from Lelystad Airport in the Netherlands. The purpose of this flight operated by a Boeing 737 owned by Transavia was to assess noise exposure in neighbouring cities.

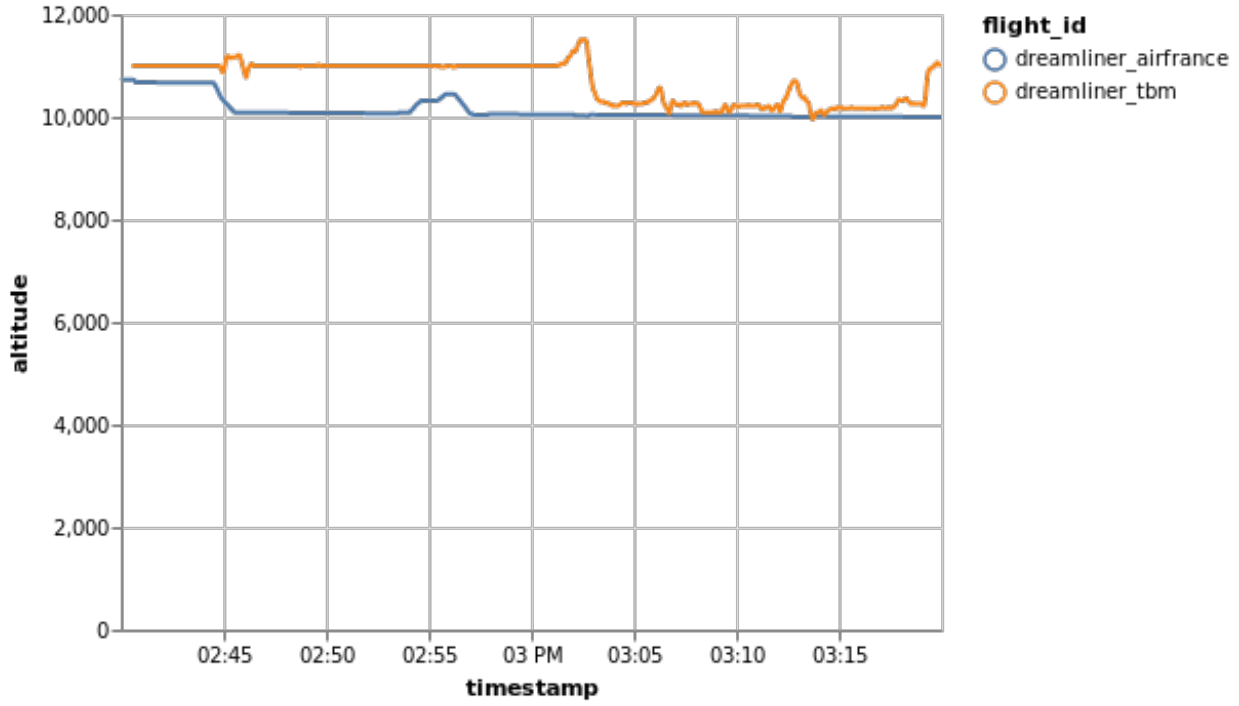
```
from traffic.data.samples import belevingsvlucht
belevingsvlucht
```

4.5.2 Dreamliner Air France

After getting their first Boeing 787 Dreamliner, Air France equipped a Socata TBM 900 and shot the following video in 8k. Both trajectories are (partly) available as sample flights.

```
from traffic.data.samples import dreamliner_airfrance
dreamliner_airfrance.between("2017-12-01 14:40", "2017-12-01 15:20")
```

```
import altair as alt

alt_chart = []
for flight in dreamliner_airfrance.between("2017-12-01 14:40", "2017-12-01 15:20"):
    alt_chart.append(flight.encode("altitude"))

alt.layer(*alt_chart)
```



4.5.3 Airbus tree

Before Christmas 2017, an Airbus pilot in Germany has delivered an early festive present by tracing the outline of an enormous Christmas tree during a test flight.

```
from traffic.data.samples import airbus_tree
airbus_tree
```

4.5.4 Aerial surveys

National Geographic Institutes sometimes conduct aerial surveys including photography and LIDAR measurements from aircraft.

Note that each submodule gives access to the `traffic` instance, but you can access each flight individually directly from the `traffic.data.samples` level.

```
from traffic.data.samples import aerialsurvey
aerialsurvey.traffic
```

```
from traffic.data.samples import fontainebleau
fontainebleau
```

4.6 Flight Information Regions (FIR)

FIR stands for Flight Information Region.

FIR are the largest regular division of airspace in use in the world today. Every portion of the atmosphere belongs to a specific FIR. Smaller countries' airspace is encompassed by a single FIR; larger countries' airspace is subdivided

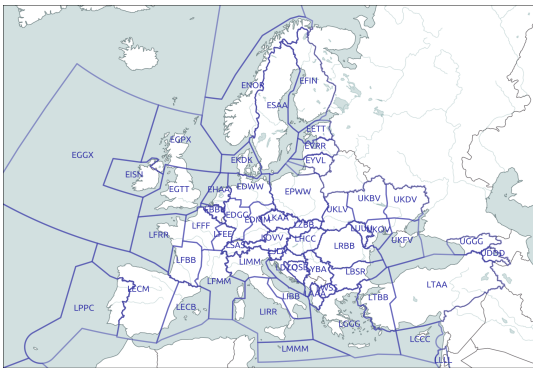
into a number of regional FIRs. Some FIRs encompass the territorial airspace of several countries. Oceanic airspace is divided into Oceanic Information Regions and delegated to a controlling authority bordering that region.

The division among authorities is done by international agreement through the International Civil Aviation Organization (ICAO). (source Wikipedia)

4.6.1 FIRs from the Eurocontrol area

FIRs of countries in the Eurocontrol area are available in the library.

```
from traffic.data import eurofirs

from traffic.drawing import TransverseMercator, countries, lakes, ocean, rivers

import matplotlib.pyplot as plt

fig, ax = plt.subplots(
    1, figsize=(15, 10),
     subplot_kw=dict(projection=TransverseMercator(10, 45))
)

ax.add_feature(countries(scale="50m"))
ax.add_feature(rivers(scale="50m"))
ax.add_feature(lakes(scale="50m"))
ax.add_feature(ocean(scale="50m"))

for name, fir in eurofirs.items():
    fir.plot(ax, edgecolor="#3a3aaa", lw=2, alpha=0.5)

    if name not in ["ENOB", "LPPO", "GCCC"]:
        fir.annotate(
            ax, s=name, ha="center", color="#3a3aaa"
        )

ax.set_extent((-20, 45, 30, 70))
```

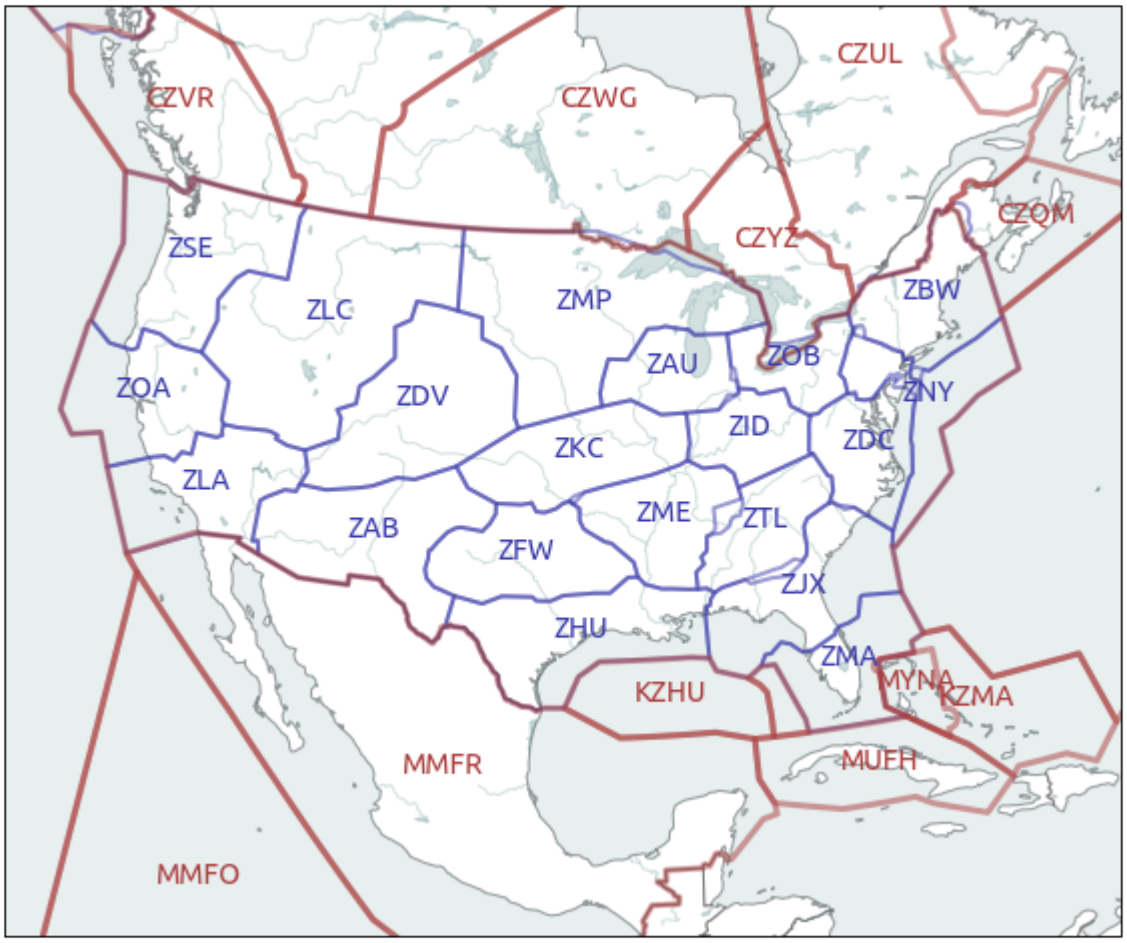


4.6.2 FIR and ARTCC from the FAA

The Federal Aviation Administration (FAA) publishes some data about their airspace in open data. The data is automatically downloaded the first time you try to access it.

Find more about this service here. On the following map, Air Route Traffic Control Centers (ARTCC) are displayed together with neighbouring FIRs.

```
from traffic.data import faa
from traffic.drawing import AlbersUSA, countries, lakes, ocean, rivers

import matplotlib.pyplot as plt

fig, ax = plt.subplots(
    figsize=(10, 10),
    subplot_kw=dict(projection=AlbersUSA())
)

ax.add_feature(countries(scale="50m"))
ax.add_feature(rivers(scale="50m"))
ax.add_feature(lakes(scale="50m"))
ax.add_feature(ocean(scale="50m", alpha=0.1))

# just a hack to push walls (i.e. projection limits)
AlbersUSA.xlimits = property(lambda _: (-3e6, 3e6))

for airspace in faa.airspace_boundary.values():

    if airspace.type == "ARTCC":
        airspace.plot(ax, edgecolor="#3a3aaa", lw=2, alpha=0.5)
        if airspace.designator != "ZAN":  # Anchorage
            airspace.annotate(
                ax,
                s=airspace.designator,
                color="#3a3aaa",
                ha="center",
                fontname="Ubuntu",
                fontsize=14,
            )

    if airspace.type == "FIR" and airspace.designator[0] in ["C", "M", "K"]:
        airspace.plot(ax, edgecolor="#aa3a3a", lw=3, alpha=0.5)
        if airspace.designator not in ["CZEG", "KZWY", "KZAK"]:
            airspace.annotate(
                ax,
                s=airspace.designator,
                color="#aa3a3a",
                ha="center",
                fontname="Ubuntu",
                fontsize=14,
            )
```



4.7 ADS-B data – OpenSky REST API

The first thing to do is to put your credentials in you configuration file. Add the following lines to the [global] section of your configuration file.

```
opensky_username =
opensky_password =
```

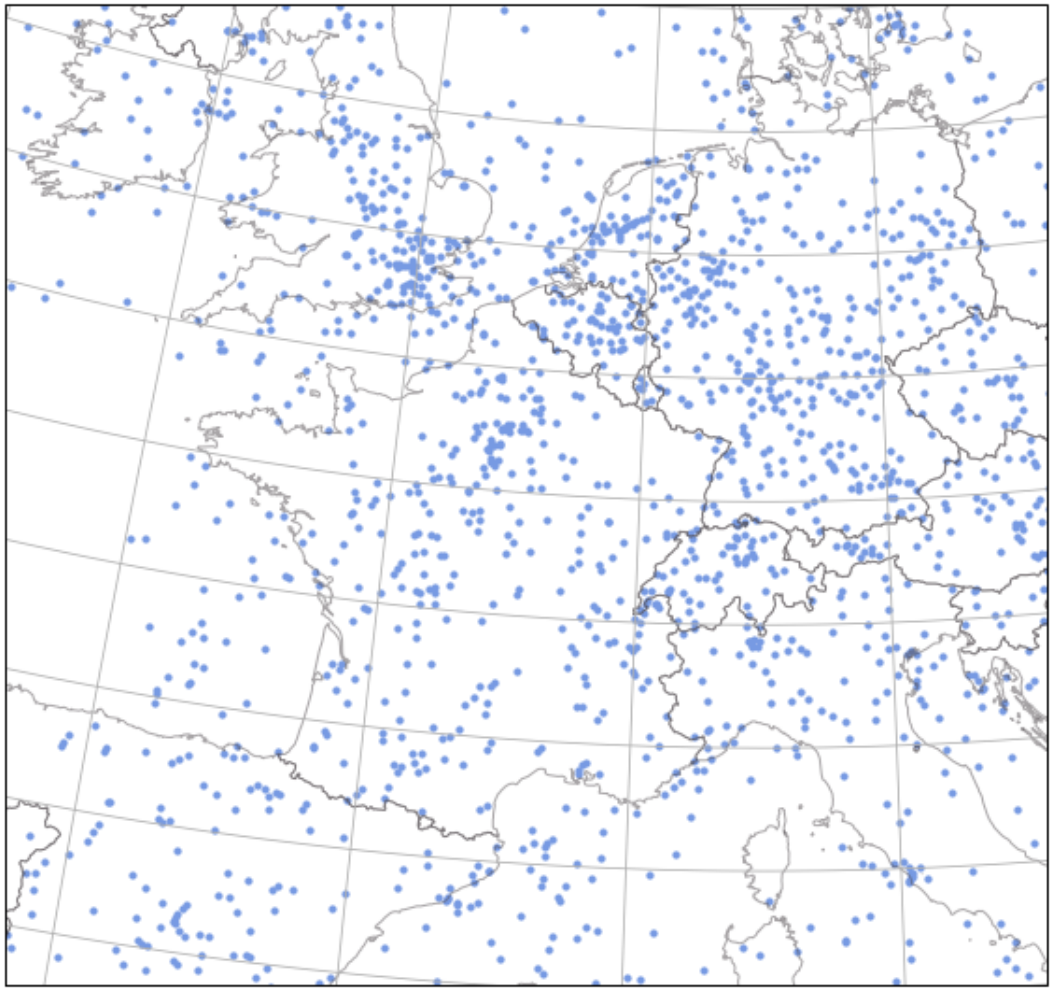
You can check the path to your configuration file here. The path is different according to OS versions so do not assume anything and check the contents of the variable.

```
>>> import traffic
>>> traffic.config_file
PosixPath('/home/xo/.config/traffic/traffic.conf')
```

Warning: Some functionalities of the REST API are currently unavailable due to issues on the server side. Documentation will be updated if the API changes when things are fixed.

The most basic usage for the OpenSky REST API is to get the instant position for all aircraft. This part actually does not require authentication.

```
%matplotlib inline
import matplotlib.pyplot as plt

from traffic.data import opensky
from traffic.drawing import EuroPP, countries

sv = opensky.api_states()

with plt.style.context('traffic'):
    fig, ax = plt.subplots(subplot_kw=dict(projection=EuroPP()))
    ax.add_feature(countries())
    ax.gridlines()
    ax.set_extent((-7, 15, 40, 55))

    sv.plot(ax, s=10)
```



You may access all callsigns in the state vector (or select few of them), then select the trajectory associated to a callsign (a `Flight`):

```
>>> import random
>>> random.sample(sv.callsigns, 6)
['RYR925Y', 'SKW5223', 'SWA1587', 'SWA2476', 'GTI8876', 'AAL2498']
```

```
flight = sv['AAL2498']
flight
```

```
# The same functionality is accessible based on the transponder code (icao24)
opensky.api_tracks(flight.icao24)
```

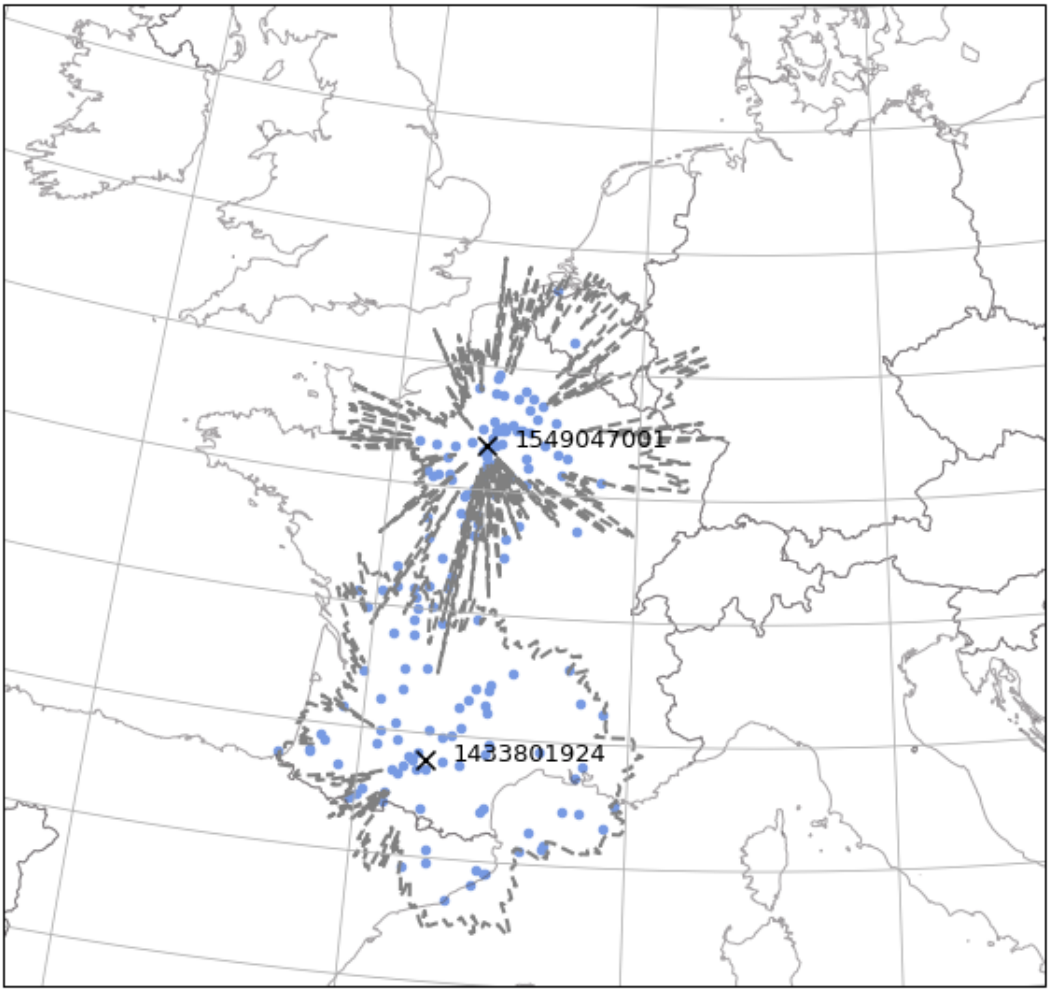
The API gives access to other functionalities, like an attempt to map a callsign to a route:

```
>>> opensky.api_routes('AFR291')
(RJBB/KIX    Osaka Kansai International Airport (Japan)
  34.427299 135.244003 altitude: 26,
 LFPG/CDG    Paris Charles de Gaulle Airport (France)
  49.012516 2.555752 altitude: 392)
```

You may get the serial numbers associated to your account and also plot the polygon of their range (by default on the current day).

```
>>> opensky.api_sensors
{'1433801924', '1549047001'}
```

```
with plt.style.context('traffic'):
    fig, ax = plt.subplots(subplot_kw=dict(projection=EuroPP()))
    ax.add_feature(countries())

    ax.gridlines()
    ax.set_extent((-7, 15, 40, 55))

    # get only the positions detected by your sensors
    sv = opensky.api_states(True)
    sv.plot(ax, s=20)

    for sensor in opensky.api_sensors:
        c = opensky.api_range(sensor)
        c.plot(ax, linewidth=2, edgecolor='grey', linestyle='dashed')
        c.point.plot(ax, marker='x', text_kw=dict(s=c.point.name))
```



The API also offers an attempt to map an airport with aircraft departing or arriving at an airport:

```
opensky.api_departure('LFBO', '2018-10-25 11:11', '2018-10-25 13:42')
```

4.8 ADS-B data – OpenSky Impala shell

For more advanced request or a dig in further history, you may be eligible for an direct access to the history database through their Impala shell.

Provided functions are here to help:

- build appropriate and efficient requests without any SQL knowledge;
- split requests efficiently and store intermediary results in cache files;
- parse results with pandas and wrap results in appropriate data structures.

The first thing to do is to put your credentials in you configuration file. Edit the following lines to your configuration file.

```
[opensky]
username =
password =
```

You can check the path to your configuration file here. The path is different according to OS versions so do not assume anything and check the contents of the variable.

```
>>> import traffic
>>> traffic.config_file
PosixPath('/home/xo/.config/traffic/traffic.conf')
```

4.8.1 Historical traffic data

4.8.2 Examples of requests

- based on callsign:

```
flight = opensky.history(
    "2017-02-05",
    # stop is implicit, i.e. stop="2017-02-06"
    callsign="EZY158T",
    return_flight=True
)
```

- based on bounding box:

```
# two hours of traffic over LFBB FIR
t_lfbb = opensky.history(
    "2018-10-01 11:00",
    "2018-10-01 13:00",
    bounds=eurofirs['LFBB']
)
```

- based on airports and callsigns (with wildcard):

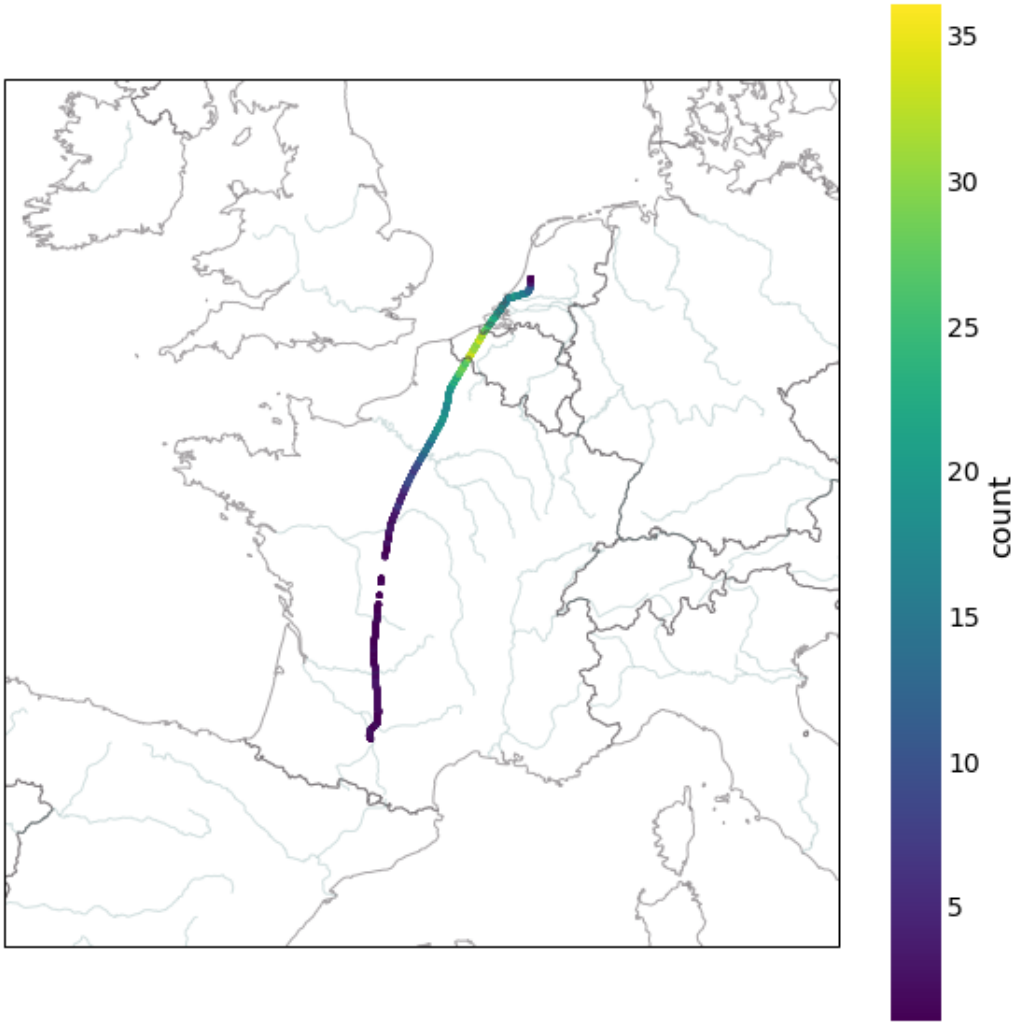
```
# Airbus test flights from and to Toulouse airport
t_aib = opensky.history(
    "2019-11-01 09:00",
    "2019-11-01 12:00",
    departure_airport="LFBO",
    arrival_airport="LFBO",
    callsign="AIB%",
)
```

- based on (own?) receiver's identifier:

```
t_sensor = opensky.history(
    "2019-11-11 10:00",
    "2019-11-11 12:00",
    serials=1433801924,
)
```

- with information about coverage:

```
from traffic.drawing import EuroPP, PlateCarree, countries, rivers

flight = opensky.history(
    "2018-06-11 15:00",
    "2018-06-11 17:00",
    callsign='KLM1308',
    count=True,
    return_flight=True
)

with plt.style.context('traffic'):
    fig, ax = plt.subplots(
        subplot_kw=dict(projection=EuroPP())
    )
    ax.add_feature(countries())
    ax.add_feature(rivers())
    ax.set_extent((-7, 13, 40, 55))

    # no specific method for that in traffic
    # but switch back to pandas DataFrame for manual plot
    flight.data.plot.scatter(
            ax=ax, x='longitude', y='latitude', c='count',
            transform=PlateCarree(), s=5, cmap='viridis'
    )
```



4.8.3 Extended Mode-S (EHS)

EHS messages are not automatically decoded for your on the OpenSky Database but you may access them and decode them from your computer.

Warning: `Flight.query_ehs()` messages also takes a dataframe argument to avoid making possibly numerous requests to the Impala database.

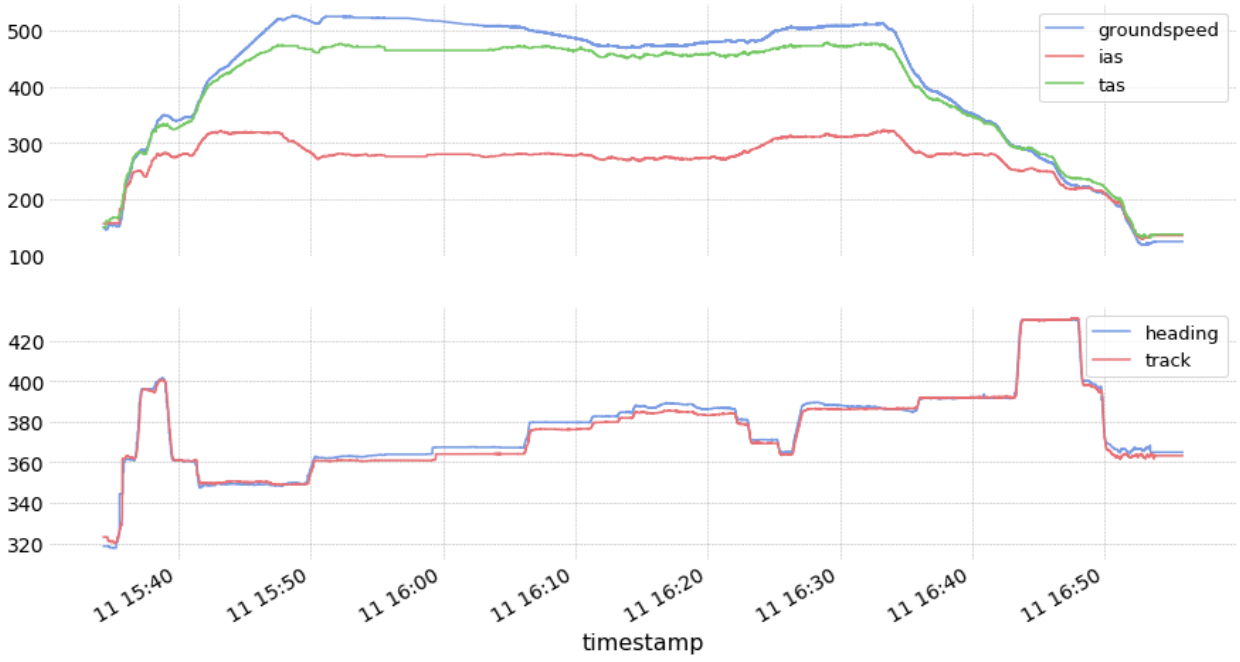
Consider using *opensky.extended()* and request all necessary data, then pass the resulting dataframe as an argument.

```
ehs_flight = (
    flight
    # this triggers a new specific call to OpenSky Impala
    .query_ehs()
    # avoid big gaps in angle
    .unwrap()
    # cascade of median filters
    .filter()
```

(continues on next page)

(continued from previous page)

```
    .filter(groundspeed=53, tas=53, ias=53, heading=53, track=53)
)

with plt.style.context('traffic'):

    fig, (ax1, ax2) = plt.subplots(
        2, 1, sharex=True, figsize=(15, 8)
    )

    ehs_flight.plot_time(ax1, ['groundspeed', 'ias', 'tas'])
    ehs_flight.plot_time(ax2, ['heading', 'track'])

    ax1.legend()
    ax2.legend()
```



4.8.4 Examples of requests

- based on transponder identifier (icao24):

```
from traffic.data.samples import belevingsvlucht

df = opensky.extended(
    belevingsvlucht.start,
    belevingsvlucht.stop,
    icao24=belevingsvlucht.icao24
)

enriched = belevingsvlucht.query_ehs(df)
```

- based on geographical bounds:

```
from traffic.data import eurofirs
from traffic.data.samples import switzerland

df = opensky.extended(
    switzerland.start_time,
    switzerland.end_time,
    bounds=eurofirs['LSAS']
)

enriched_ch = (
    switzerland
    .filter()
    .query_ehs(df)
    .resample('1s')
    .eval(desc='', max_workers=4)
)
```

- based on airports, together with traffic:

```
schiphol = opensky.history(
    "2019-11-11 12:00",
    "2019-11-11 14:00",
    airport="EHAM"
)

df = opensky.extended(
    "2019-11-11 12:00",
    "2019-11-11 14:00",
    airport="EHAM"
)

enriched_eham = (
    schiphol
    .filter()
    .query_ehs(df)
    .resample('1s')
    .eval(desc='', max_workers=4)
)
```

4.8.5 Flight list by airport

4.9 Airspaces from Eurocontrol NM

The first thing to do is to put the path to a directory containing your files from the Eurocontrol NM in your configuration file.

You will need to put the three following files from the same AIRAC cycle in the same directory:

```
/home/xo/Documents/data/AIRAC_1703
├── Sectors_1703_Collapse.spc
├── Sectors_1703_Sectors.are
└── Sectors_1703_Sectors.sls
```

Identify the path to your configuration file here:

```
>>> import traffic
>>> traffic.config_file
PosixPath('/home/xo/.config/traffic/traffic.conf')
```

Then edit the following line accordingly:

```
[global]
nm_path = /home/xo/Documents/data/AIRAC_1703
```

4.9.1 Basic usage

The basic way to access an airspace is by its name:

```
from traffic.data import nm_airspaces
nm_airspaces['LFBBFIR']
```

Most airspaces are a composition of elementary airspaces. Their union is computed and yields a list of polygons associated with minimum and maximum flight levels.

```
nm_airspaces['LFBBBDX']
```

Some basic looping can help finding the airspaces your need. Two methods are provided:

- the parse method yields only metadata about the airspace (basically name and type);
- the search method computes the actual shape of the airspace to you get for example the area of its 2D projection. Here, we use it to find the biggest control sector with a name starting with LFBB.

```
# get all types of airspaces provided (you only need metadata)
>>> set(a.type for a in nm_airspaces.parse('.*'))
{'AREA', 'AUA', 'AUAG', 'CLUS', 'CRAS', 'CRSA', 'CS', 'ERAS', 'ERSA', 'ES', 'FIR',
↪'NAS', 'REG'}
```

```
# Find the biggest CS in Bordeaux ACC
from operator import attrgetter
max(nm_airspaces.search('LFBB.*/CS'), key=attrgetter('area'))
```

4.9.2 Use cases

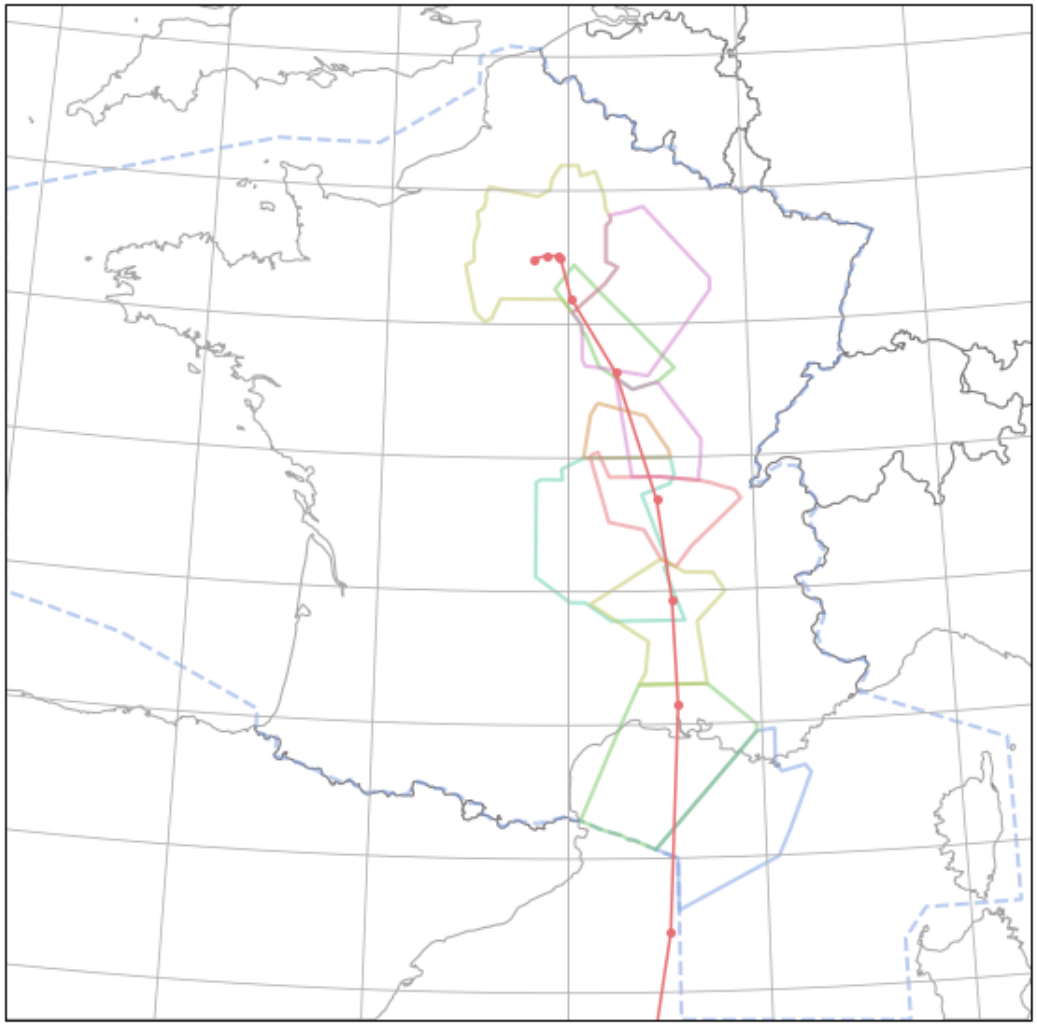
The provided infrastructure lets you find all elementary sectors crossed by a trajectory (here from the so6 file)

```
from traffic.data import SO6
so6 = SO6.from_file('data/sample_m3.so6.7z')

with plt.style.context('traffic'):
    fig = plt.figure()
    ax = plt.axes(projection=Lambert93())

    ax.add_feature(countries())
    ax.gridlines()
    ax.set_extent(nm_airspaces['LFFFUIR'])

    nm_airspaces['LFFFUIR'].plot(ax, lw=2, alpha=.5, linestyle='dashed')
    so6['DAH1008'].plot(ax, marker='.')
```

(continues on next page)

(continued from previous page)

```
    # display elementary sectors (ES) crossed by the trajectory
    for airspace in nm_airspaces.search("LF.*/ES"):
        if so6['DAH1008'].intersects(airspace):
            airspace.plot(ax, alpha=.5, lw=2)
```



Another use case could be to plot all flights going through an airspace at noon.

```
# callsigns at noon inside LFBBBDX
bdx_noon = (
    so6.at("2018-01-01 12:00")
    .inside_bbox(nm_airspaces['LFBBBDX'])
    .intersects(nm_airspaces['LFBBBDX'])
)

# full so6 limited to flights hereabove
so6_bdx_noon = so6.select(bdx_noon)
```

```
from traffic.drawing import EuroPP, countries
```

(continues on next page)

(continued from previous page)

```
with plt.style.context('traffic'):
    fig = plt.figure()
    ax = plt.axes(projection=EuroPP())

    ax.add_feature(countries())
    ax.gridlines()
    ax.set_extent((-10, 15, 35, 55))

    nm_airspaces['LFBBBDX'].plot(ax, lw=2, alpha=.5)

    for _, flight in so6_bdx_noon:
        flight.plot(ax, color='#aa3a3a', lw=.4, alpha=.5)
```

4.10 DDR files from Eurocontrol NM

Tip: Advanced use cases are presented in the Airspaces from Eurocontrol NM section.

4.11 Web services from Eurocontrol NM

The NM B2B web services is an interface provided by the EUROCONTROL Network Manager (NM) for system-to-system access to its services and data, allowing users to retrieve and use the information in their own systems.

We provide a basic API for some NM web services. They have been implemented on a per need basis: not everything is provided, but functionalities may be added in the future.

CHAPTER 5

Algorithms

5.1 Closest point of approach

5.2 Trajectory clustering

An API for trajectory clustering is provided in the Traffic class. Regular clustering methods from scikit-learn can be passed as parameters, or any object implementing the fit(), predict() and fit_predict() methods (see ClusterMixin.)

Data is prepared to match the input format of these methods, preprocessed with transformers (see TransformerMixin), e.g. MinMaxScaler() or StandardScaler(), and the result of the clustering is added to the Traffic DataFrame as a new `cluster` feature.

The following illustrates the usage of the API on a sample dataset of traffic over Switzerland. The dataset contains flight trajectories over Switzerland on August 1st 2018, cleaned and resampled to one point every five seconds. The data is available as a basic import:

```
# see https://traffic-viz.github.io/samples.html if any issue on import
from traffic.data.samples import switzerland
```

The full dataset of trajectories can be displayed on a basic Switzerland map.

```
import matplotlib.pyplot as plt

from traffic.core.projection import CH1903
from traffic.drawing import countries

with plt.style.context("traffic"):
    ax = plt.axes(projection=CH1903())
    ax.add_feature(countries())
    switzerland.plot(ax, alpha=0.1)
```

It is often relevant to use the track angle when doing clustering on trajectories as it helps separating close trajectory flows heading in opposite directions. However a track angle may be problematic when crossing the 360°/0° (or -180°/180°) line.

Flight.unwrap() is probably the most relevant method to apply:

```
t_unwrapped = switzerland.assign_id().unwrap().eval(max_workers=4)
```

The following snippet projects trajectories to a local projection, resample each trajectory to 15 sample points, apply scikit-learn's StandardScaler() before calling a DBSCAN.fit_predict() method.

If a fitted DBSCAN instance is passed in parameter, the predict() method can be called on new data.

```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
from traffic.core.projection import CH1903

t_dbscan = t_unwrapped.clustering(
    nb_samples=15,
    projection=CH1903(),
    features=["x", "y", "track_unwrapped"],
    clustering=DBSCAN(eps=0.5, min_samples=10),
    transform=StandardScaler(),
).fit_predict()
```

```
>>> dict(t_dbscan.groupby(["cluster"]).agg({"flight_id": "nunique"}).flight_id)

{-1: 685, 0: 62, 1: 30, 2: 23, 3: 17, 4: 14, 5: 25, 6: 24, 7: 70, 8: 49, 9: 53,
 10: 15, 11: 24, 12: 30, 13: 19, 14: 23, 15: 25, 16: 19, 17: 16, 18: 10, 19: 11}
```

The above distribution results from the DBSCAN prediction. We can compare the results with a Gaussian Mixture

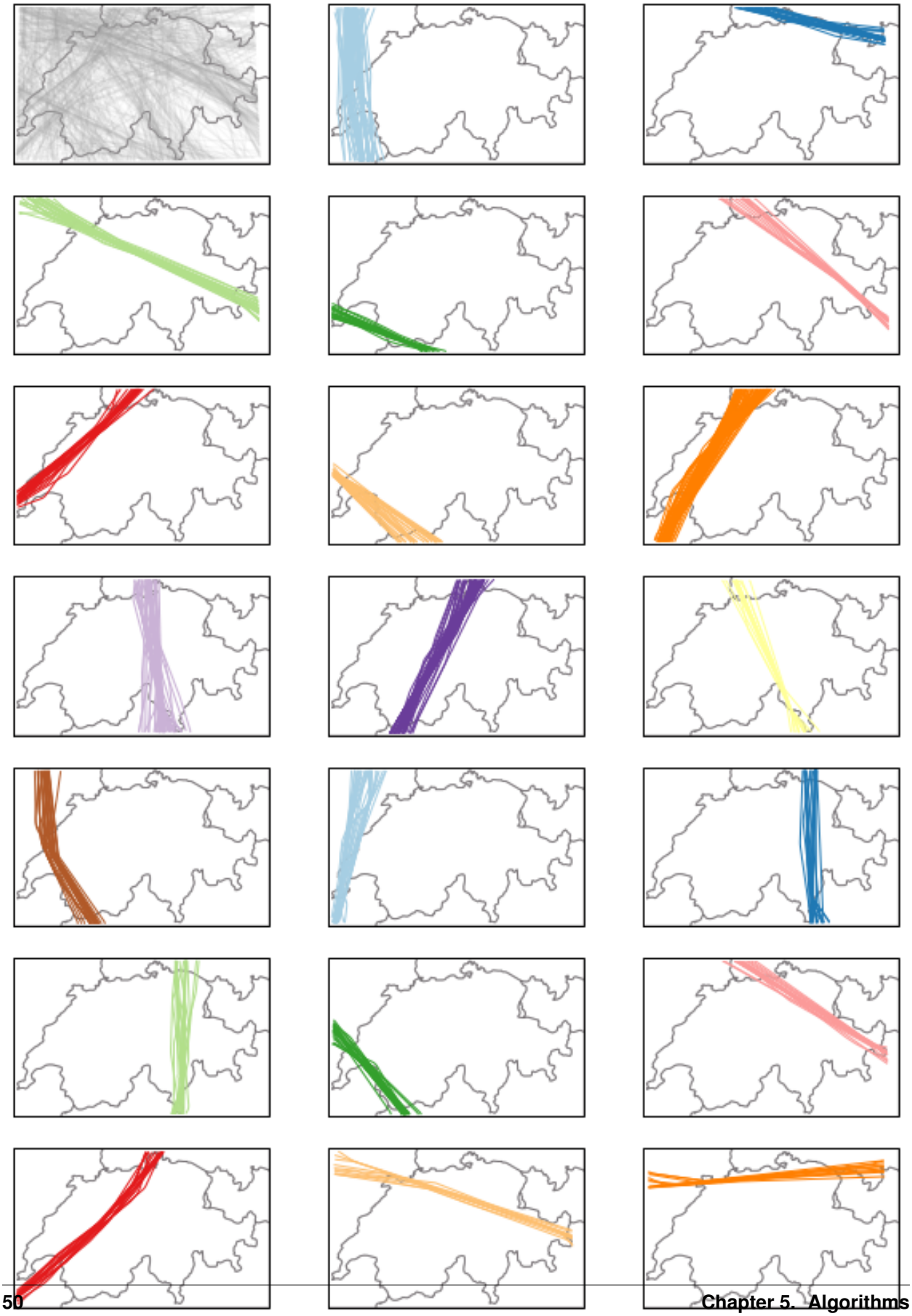
model for example. This model must be initialized with a number of components. We chose here 19 components to compare them with the result of the DBSCAN algorithm.

```
from sklearn.mixture import GaussianMixture

t_gmm = t_unwrapped.clustering(
    nb_samples=15,
    projection=CH1903(),
    features=["x", "y", "track_unwrapped"],
    clustering=GaussianMixture(n_components=19),
    transform=StandardScaler(),
).fit_predict()
```

```
>>> dict(t_gmm.groupby(["cluster"]).agg({"flight_id": "nunique"}).flight_id)

{0: 94, 1: 76, 2: 46, 3: 145, 4: 47, 5: 89, 6: 76, 7: 50, 8: 143, 9: 57,
 10: 31, 11: 108, 12: 35, 13: 75, 14: 35, 15: 55, 16: 12, 17: 13, 18: 57}
```

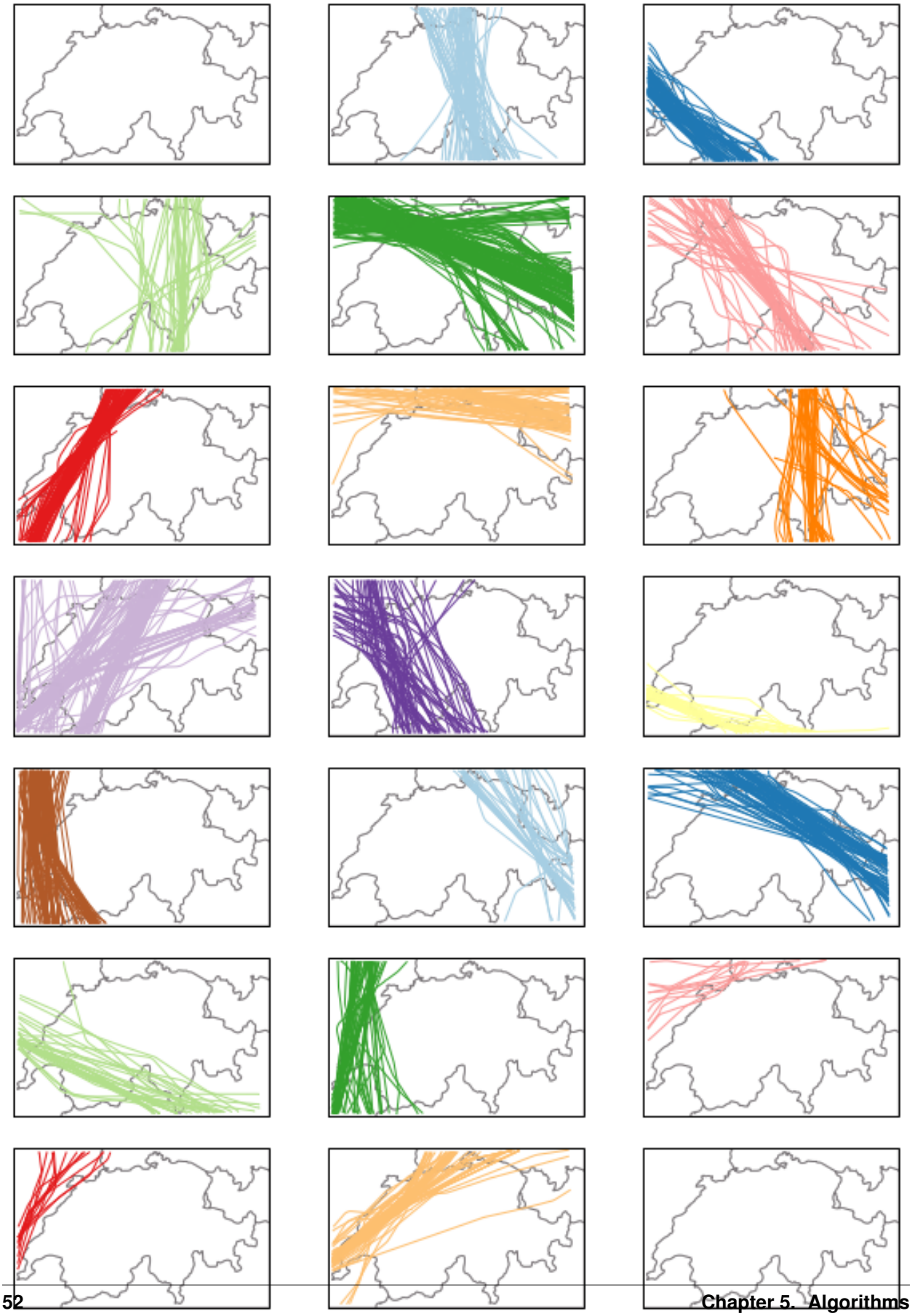
The following snippets visualises each trajectory cluster with a given color. Many outliers appear in shaded grey in the first quartet.

```
from itertools import islice, cycle
from traffic.drawing import countries

n_clusters = 1 + t_dbscan.data.cluster.max()

# -- dealing with colours --

color_cycle = cycle(
    "#a6cee3 #1f78b4 #b2df8a #33a02c #fb9a99 #e31a1c "
    "#fdbf6f #ff7f00 #cab2d6 #6a3d9a #ffff99 #b15928".split()
)
colors = list(islice(color_cycle, n_clusters))
colors.append("#aaaaaa")  # color for outliers, if any

# -- dealing with the grid --

nb_cols = 3
nb_lines = (1 + n_clusters) // nb_cols + (((1 + n_clusters) % nb_cols) > 0)

with plt.style.context("traffic"):

    fig, ax = plt.subplots(
        nb_lines, nb_cols, figsize=(10, 15), subplot_kw=dict(projection=CH1903())
    )

    for cluster in range(-1, n_clusters):
        ax_ = ax[(cluster + 1) // nb_cols][(cluster + 1) % nb_cols]
        ax_.add_feature(countries())

        t_dbscan.query(f"cluster == {cluster}").plot(
            ax_, color=colors[cluster], alpha=0.1 if cluster == -1 else 1
        )
        ax_.set_global()
```



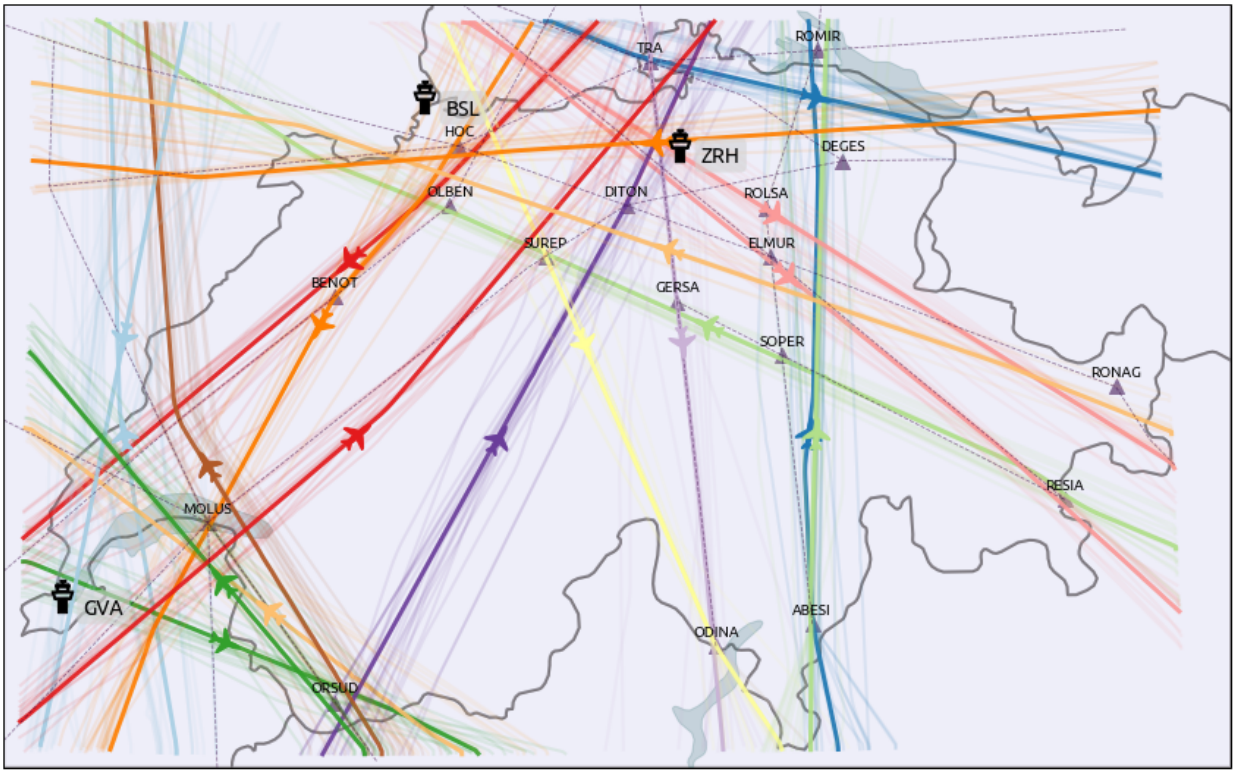
Gaussian Mixtures do not yield any outlier. The following clustering is balanced differently.

```
with plt.style.context("traffic"):

    fig, ax = plt.subplots(
        nb_lines, nb_cols, figsize=(10, 15), subplot_kw=dict(projection=CH1903())
    )

    for cluster in range(-1, n_clusters):
        ax_ = ax[(cluster + 1) // nb_cols][(cluster + 1) % nb_cols]
        ax_.add_feature(countries())

        t_gmm.query(f"cluster == {cluster}").plot(
            ax_, color=colors[cluster], alpha=0.1 if cluster == -1 else 1
        )
        ax_.set_global()
```



The following map demonstrates how to use the Traffic.centroid() method, computed with the same parameters as the clustering.

```
from random import sample

from traffic.data import airports, airways, navaids
from traffic.drawing import CH1903, countries, lakes
from traffic.drawing.markers import rotate_marker, atc_tower, aircraft

with plt.style.context("traffic"):
    fig, ax = plt.subplots(1, figsize=(15, 10), subplot_kw=dict(projection=CH1903()))
    ax.add_feature(countries(facecolor="#dedef4", linewidth=2))
    ax.add_feature(lakes())

    for cluster in range(n_clusters):

        current_cluster = t_dbscan.query(f"cluster == {cluster}")
        centroid = current_cluster.centroid(15, projection=CH1903())
        centroid.plot(ax, color=colors[cluster], alpha=0.9, linewidth=3)
        centroid_mark = centroid.at_ratio(0.45)

        centroid_mark.plot(
            ax,
            color=colors[cluster],
            marker=rotate_marker(aircraft, centroid_mark.track),
            s=500,
            text_kw=dict(s=""),  # no text associated
        )
        sample_size = min(20, len(current_cluster))

        for flight_id in sample(current_cluster.flight_ids, sample_size):
            current_cluster[flight_id].plot(
                ax, color=colors[cluster], alpha=0.1, linewidth=2
            )

    swiss_airways = airways.extent("Switzerland")
    for (
        name
    ) in "UL613 UL856 UM729 UN491 UN850 UN851 UN853 UN869 UN871 UQ341 Z50".split():
        swiss_airways[name].plot(ax, color="#34013f")

    for name in "BSL GVA ZRH".split():
        bbox = dict(
            facecolor="lightgray", edgecolor="none", alpha=0.6, boxstyle="round"
        )
        airports[name].point.plot(ax, marker=atc_tower, s=500, zorder=5)

    swiss_navaids = navaids.extent("Switzerland")
    for name in (
        "ABESI BENOT DEGES DITON ELMUR GERSA HOC MOLUS ODINA OLBEN "
        "ORSUD RESIA ROLSA ROMIR RONAG SOPER SUREP TRA".split()
    ):
        swiss_navaids[name].plot(ax, marker="^", color="#34013f")

    ax.set_global()
```



The result may be compared to the Blick newspaper great visualisation by Simon Huwiler and Priska Wallimann here. (github repository)

CHAPTER 6

Exporting and storing data

The traffic library is based on the pandas library for representing and manipulating trajectories and on the shapely library for manipulating geographical information like airways, beacons, airports and airspaces.

Various parsers are provided (feel free to file a PR if you use other tools with different data sources) but after applying different operations to imported data, you may want to export data for storing, sharing and loading again later.

6.1 Traffic and Flight structures

The question applies to pandas dataframes as well, with many opinions available on the net. In general, the question boils down to whether you want to distribute the data, how fast you need to access it and how long you need to keep it.

- CSV (comma separated values) is a pretty standard and widely acknowledged format (modulo the definition of the separator). It is easy to parse but it can be slow when data gets large. Also it doesn't contain information about types so you need to check dtypes and transform them manually if need be. A good rule of thumb could be to parse CSV data only once and to use another format for storing it for future use.
- JSON (JavaScript Object Notation) is another lightweight text notation, human readable, also slow to parse. The added value compared to CSV is that you can distinguish boolean, numerical and string values.
- Pickle is the standard format for Python serialisation. The binary representation of data is dumped as is in a file, no question asked. It is fast to read and write and you are sure to recover your data after you restart your Python interpreter/kernel. The downside is that the serialisation format may change with Python and pandas versions so it is not a good format for sharing and storing data in the time.
- HDF (Hierarchical Data Format) is a cross platform and cross language standard format for storing very large amounts of data. You may need extra dependencies to read and write from this format.
- Apache Parquet is another columnar cross platform and cross language standard storage format. Its implementation inside pandas is very fast for both read and write operations and the resulting files are rather compact. Types are respected but all Python structures (like sets, lists and dictionaries) may not be directly exportable. You may need extra dependencies to read and write from this format.

The Flight and Traffic structures implement the following methods:

class traffic.core.mixins.**DataFrameMixin**

DataFrameMixin aggregates a pandas DataFrame and provides the same representation methods.

classmethod from_file(*filename: Union[pathlib.Path, str], **kwargs*) → Optional[T]

Read data from various formats.

This class method dispatches the loading of data in various format to the proper `pandas.read_*` method based on the extension of the filename.

- .pkl and .pkl.gz dispatch to `pandas.read_pickle`;
- .parquet and .parquet.gz dispatch to `pandas.read_parquet`;
- .json and .json.gz dispatch to `pandas.read_json`;
- .csv and .csv.gz dispatch to `pandas.read_csv`;
- .h5 dispatch to `pandas.read_hdf`.

Other extensions return `None`. Specific arguments may be passed to the underlying `pandas.read_*` method with the kwargs argument.

Example usage:

```
>>> t = Traffic.from_file("data/sample_opensky.pkl")
```

to_csv(*filename: Union[str, pathlib.Path], *args, **kwargs*) → None

Exports to CSV format.

Options can be passed to `pandas.to_csv` as args and kwargs arguments.

Read more about export formats in the Exporting and Storing data section

to_hdf(*filename: Union[str, pathlib.Path], *args, **kwargs*) → None

Exports to HDF format.

Options can be passed to `pandas.to_hdf` as args and kwargs arguments.

Read more about export formats in the Exporting and Storing data section

to_json(*filename: Union[str, pathlib.Path], *args, **kwargs*) → None

Exports to JSON format.

Options can be passed to `pandas.to_json` as args and kwargs arguments.

Read more about export formats in the Exporting and Storing data section

to_parquet(*filename: Union[str, pathlib.Path], *args, **kwargs*) → None

Exports to parquet format.

Options can be passed to `pandas.to_parquet` as args and kwargs arguments.

Read more about export formats in the Exporting and Storing data section

to_pickle(*filename: Union[str, pathlib.Path], *args, **kwargs*) → None

Exports to pickle format.

Options can be passed to `pandas.to_pickle` as args and kwargs arguments.

Read more about export formats in the Exporting and Storing data section

CHAPTER 7

Command line interface

CHAPTER 8

Graphical user interface

The traffic library comes with a Qt Graphical user interface (GUI) designed to decode and explore historical and live data.

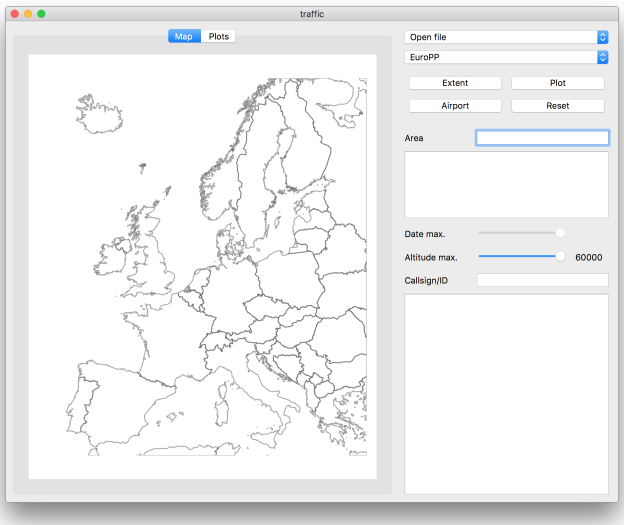
The GUI can be installed as a plugin package and is accessible through the following command (clickable application icons are doable and will probably be automatically generated in future versions)

```
pip install traffic_qtgui  # first install the plugin
traffic gui
```

8.1 Data exploration

The GUI consists of two panes:

- the **display** pane on the left-hand side, with a *map* and a *plots* tab;
- the **command** pane on the right-hand side, with selection and filtering buttons.



By default, the tool opens with a EuroPP projection. Other projections like Mercator are also available by default: you can customize more projections in the configuration file, in the `[projections]` section:

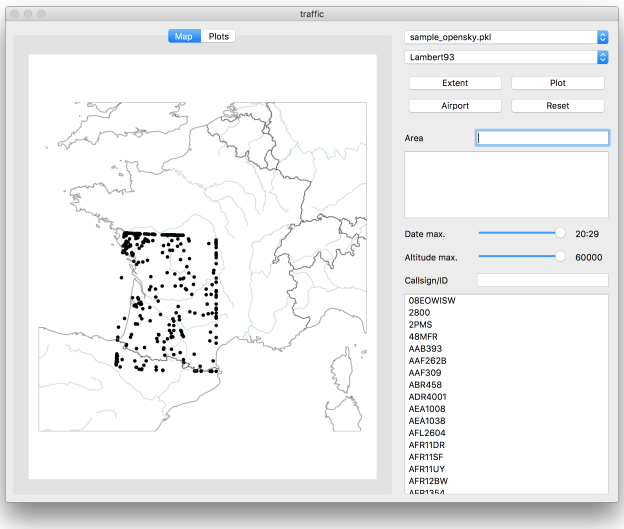
```
[projections]
default = EuroPP
extra = Lambert93; Amersfoort; GaussKruger
```

Note: Available projections are default cartopy projections, completed by additional common European projections in the cartotools dependency module (here Lambert 93 is the official projection in France, Amersfoort in the Netherlands and Gauss-Krüger in Germany)

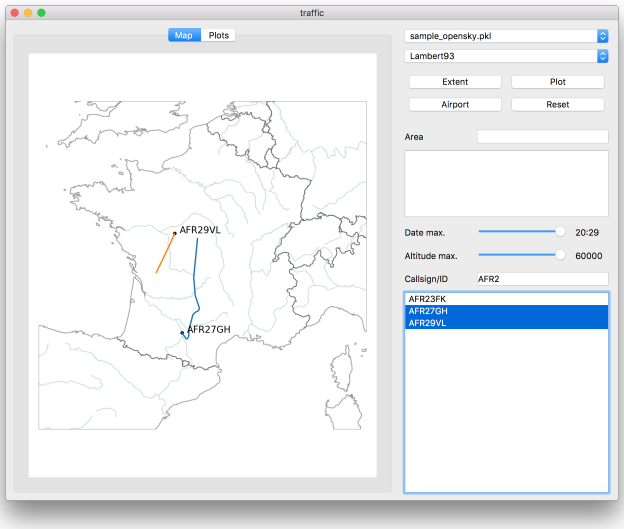
Tip: You can implement more projections as plugins.

You can either pan and zoom the map. Zoom is operated by the mouse or trackpad scrool. Note that on MacOS, the trackpad scroll requires clicking.

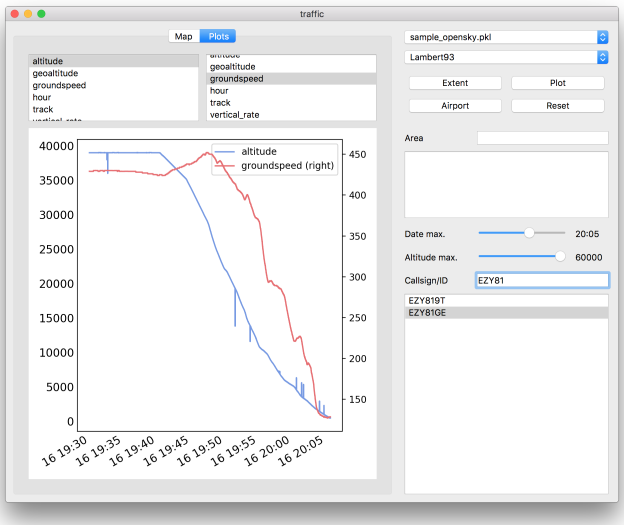
In order to explore data, click on *Open file* and select a .pkl file (like `sample_opensky.pkl` in the `data/` directory) By default, a scatter of all last recorded points is displayed.



- You may select callsigns in order to plot trajectories.
- Date and altitude sliders operate filters on the full pandas DataFrame.



In the *Plots* tab, you may select one callsign with different signals (e.g. *altitude* on the left-hand side and *ground speed* on the righ-hand side) **or (exclusive)** several callsigns with one signal (e.g. *altitude*).

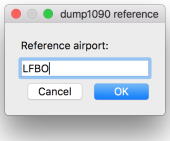


8.2 Data recording

The application does not process raw signals from any 1090 MHz antenna. It relies on other tools and listen to a standard format of raw data broadcasted on specific ports. Specifically, you may run an instance of dump1090 be running with appropriate options:

```
dump1090 --interactive --net
```

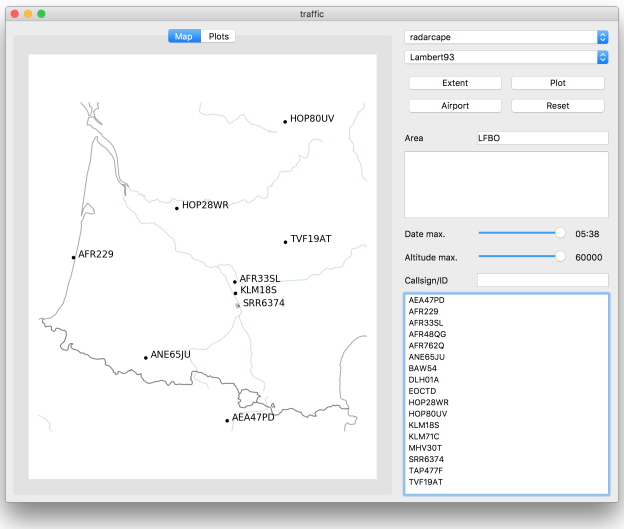
Then, the second option in the *Open file* dropdown menu is *dump1090*.
You should be asked for a reference airport:



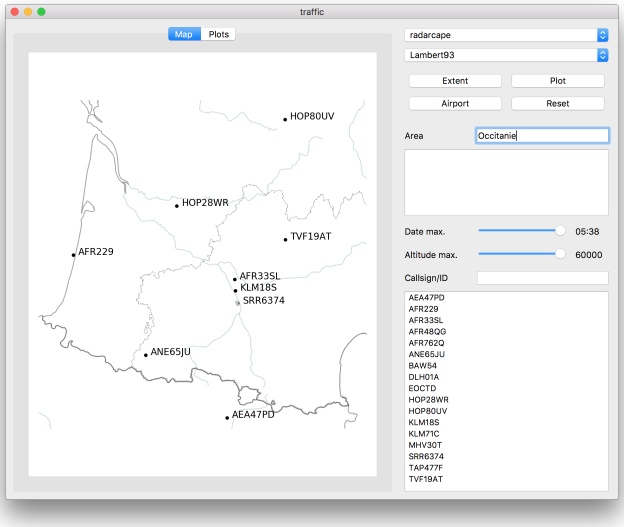
Fill in the ICAO (LFPG, PHNL, RJBB, etc.) or IATA (SFO, AMS, HKG, etc.) code of the airport for reference coordinates of the antenna. The associated latitude and longitude coordinates are useful to process ground messages. If several airports are in your neighbourhood, choose any of them (the closest one?)

Tip: If you use a different decoding device like radarcape, data is broadcasted on a different port (usually 10003). You may add the corresponding address (with airport) in your configuration file: this will add an option in the dropdown menu.

```
[decoders]
radarcape = xxx.xxx.xxx.xxx:10003/LFBO
```

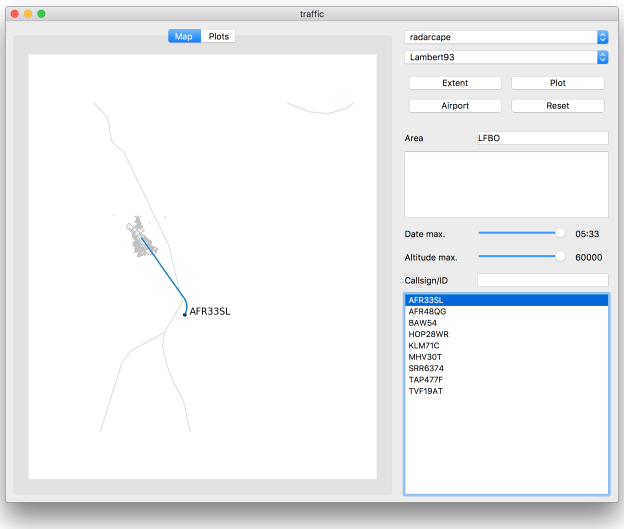


For more details on your map, you may enter a name in the *Area* field and click *Plot*. The corresponding boundaries will be downloaded from OpenStreetMap servers and added to the map.

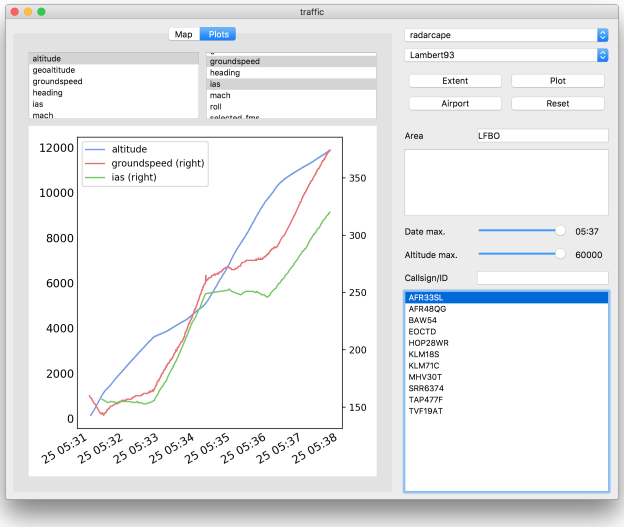


This also work with airports: enter the ICAO or IATA code in the *Area* field and click *Airport*. Data is downloaded (and cached) from OpenStreetMap servers.

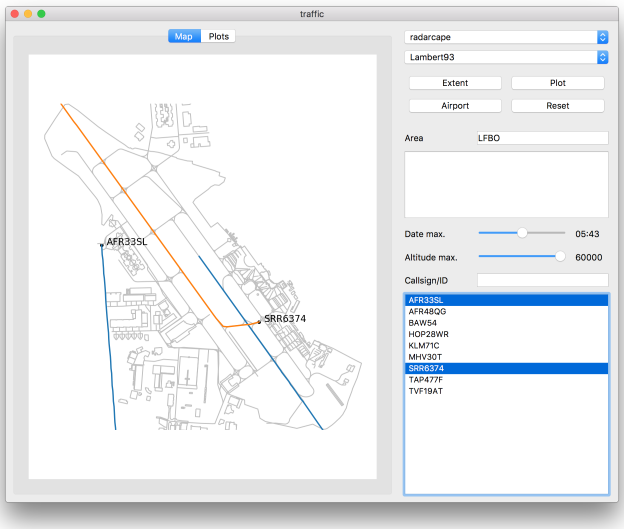
You can then select a callsign and follow its trajectory:



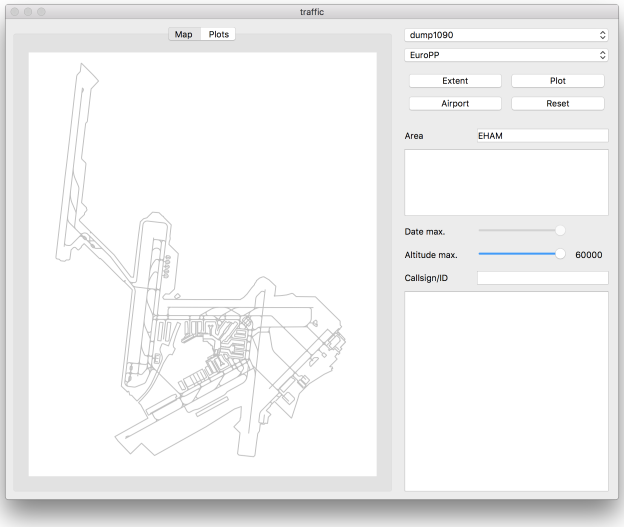
The second tab lets you plot other details of the trajectory for different signals: *altitude* shows the climbing profile. *Indicated Airspeed* (IAS) is plotted here so as to reflect the 250 knots limit under 10,000 ft.



You can automatically zoom to a geographical location by clicking *Extent* and observe ground movements on the airfield:



If you are closer to bigger airport with a good reception (from the rooftop viewing area), you may have fun looking at aircraft ground movements from your application.



When it decodes data, the GUI also writes a CSV text file in your home directory. The first column in the file is a GPS timestamp (nano-second precision) and the second column the raw message.

```
$ head ADSB_EHS_RAW_20190225_dump1090.csv

1551072485.607867,1a331339643b50b639903907fbc154da992c5cff2dbf13
1551072485.609867,1a3313396459d5173a903907fbc254000016c354424d11
1551072485.614405,1a331339649f1359418d45ce4699907294c83c1b5b7b99
1551072485.647058,1a33133966915136258dab120b58d302fde3ffc1da0aab
1551072485.725486,1a3313396b3e07ea38903907dbc23400000536e52a12b6
```

After you close the GUI, you can decode the .pkl file for an offline exploration of the data:

```
traffic decode ADSB_EHS_RAW_20190225_dump1090.csv LFBO
```

CHAPTER 9

Plugins

Plugins are pieces of software which are designed to extend the basic functionalities of the traffic library. Plugins can be implemented through a registration mechanism and selectively activated in the configuration file.

Some plugins are provided by the traffic library with visualisation facilities for Leaflet, Kepler.gl and CesiumJS.

9.1 Plugin activation

You may activate plugins in the configuration file:

```
>>> import traffic
>>> traffic.config_file
PosixPath('/home/xo/.config/traffic/traffic.conf')
```

Then edit the following line according to the plugins you want to activate:

```
[plugins]
enabled_plugins = Leaflet, Kepler, CesiumJS
```

9.2 Available plugins

9.2.1 Leaflet

Leaflet offers a Python widget for Jupyter Lab. Flights and Airspaces can easily be plotted into such widgets. The Traffic extracted above can be conveniently explored in the following widget.

Just for fun, you can zoom up to the airport level and check the runway used for landing.

Warning: The plugin must be activated in your configuration file.

```
from ipyleaflet import Map, basemaps
from ipywidgets import Layout

map_ = Map(
    center=(43.5, 1.5),
    zoom=7,
    basemap=basemaps.Stamen.Terrain,
    layout=Layout(width="100%", max_width="800px", height="500px"),
)

map_.add_layer(nm_airspaces["LFBOTMA"])
for flight in demo:
    map_.add_layer(flight, color="#990000", weight=2)

map_
```

9.2.2 Kepler.gl

Kepler.gl is a data-agnostic, high-performance web-based application for visual exploration of large-scale geolocation data sets. It is built on top of Mapbox GL and deck.gl, and can render millions of points representing thousands of trips and perform spatial aggregations on the fly. A Jupyter binding is available here: Flights, Traffic, Airspaces, and other data can easily be plotted into the widget.

> **Warning:** The plugin must be activated in your configuration file.

The following example displays the same example as with Leaflet, together with extra data: German airports, VORs in Ireland and French FIRs.

```
from traffic.data import airports, navaids, eurofirs

from keplergl import KeplerGl

map_ = KeplerGl(height=500)

# add a Flight or a Traffic
map_.add_data(belevingsvlucht, name="Belevingsvlucht")
map_.add_data(demo, name="Quickstart trajectories")

# also other sources of data
map_.add_data(  # airports (a subset)
    airports.query('country == "Germany"'),
    name="German airports"
)
map_.add_data(  # navaids (a subset)
    navaids.extent("Ireland").query("type =='VOR'"),
    name="Irish VORs"
)
map_.add_data(lfbo_tma, "Toulouse airport TMA")

# you can write your own generator to send data to the widget
map_.add_data(
    (
        fir for name, fir in eurofirs.items()
```

(continues on next page)

(continued from previous page)

```
        if name.startswith("LF")
    ),
    "French FIRs"
)

map_
```

9.2.3 CesiumJS

CesiumJS is a great tool for displaying and animating geospatial 3D data. The library provides an export of a Traffic structure to a czml file. A copy of this file is available in the `data/` directory. You may drag and drop it on the http://cesiumjs.org/ page after you open it on your browser.

Warning: The plugin must be activated in your configuration file.

```
demo.to_czml('data/sample_cesium.czml')
```

9.2.4 BlueSky

The examples are provided using the data produced in the Quickstart page.

```
from traffic.data.samples import quickstart, lfbo_tma

def landing_trajectory(flight: "Flight") -> bool:
    return (
        flight.min("altitude") < 10_000 and
        flight.mean("vertical_rate") < -500
    )

demo = (
    quickstart
    # non intersecting flights are discarded
    .intersects(lfbo_tma)
    # intersecting flights are filtered
    .filter()
    # filtered flights not matching the condition are discarded
    .filter_if(landing_trajectory)
    # stay below 25000ft
    .query('altitude < 25000')
    # final multiprocessed evaluation (4 cores) through one iteration
    .eval(max_workers=4)
)
```

9.3 Plugin registration

You may write your own plugins to monkey-patch the library and registrate them using entry points in your setup.py configuration.

Write your code in a `my_traffic_plugin/plugin.py` and use the following sample setup.py to registrate your plugin. Then edit the configuration file and add *MyPlugin* to the list of enabled plugins.

You may then `python setup.py install` your plugin. Put the monkey-patching in an `_onload()` function that will be called iff the plugin is enabled. (Check an example on the github repository)

```
from setuptools import setup

setup(
    name="my_traffic_plugin",
    version="0.1",
    author="Antoine",
    author_email="saintex@aeropostale.fr",
    license="MIT",
    description="My first traffic plugin",
    entry_points={
        "traffic.plugins": [
            "MyPlugin = my_traffic_plugin.plugin"
        ]
    },
    install_requires=["traffic"]
)
```

CHAPTER 10

Scenarios and use cases

10.1 Is this a plane?

On my way to the Opensky Network workshop 2018, I could catch a picture of this aircraft which looked really close. I wondered how close it came to us considering that the usual separation rule between aircraft is of 5 nautical miles (horizontal) and 1000ft (vertical).

First let's get information about the flight I was on.

```
from traffic.data import opensky

flight = opensky.history(
    "2018-11-15 06:00",  # UTC
    "2018-11-15 08:00",
    callsign='DLH07F' # of course, you need to know the callsign of your flight
)
```

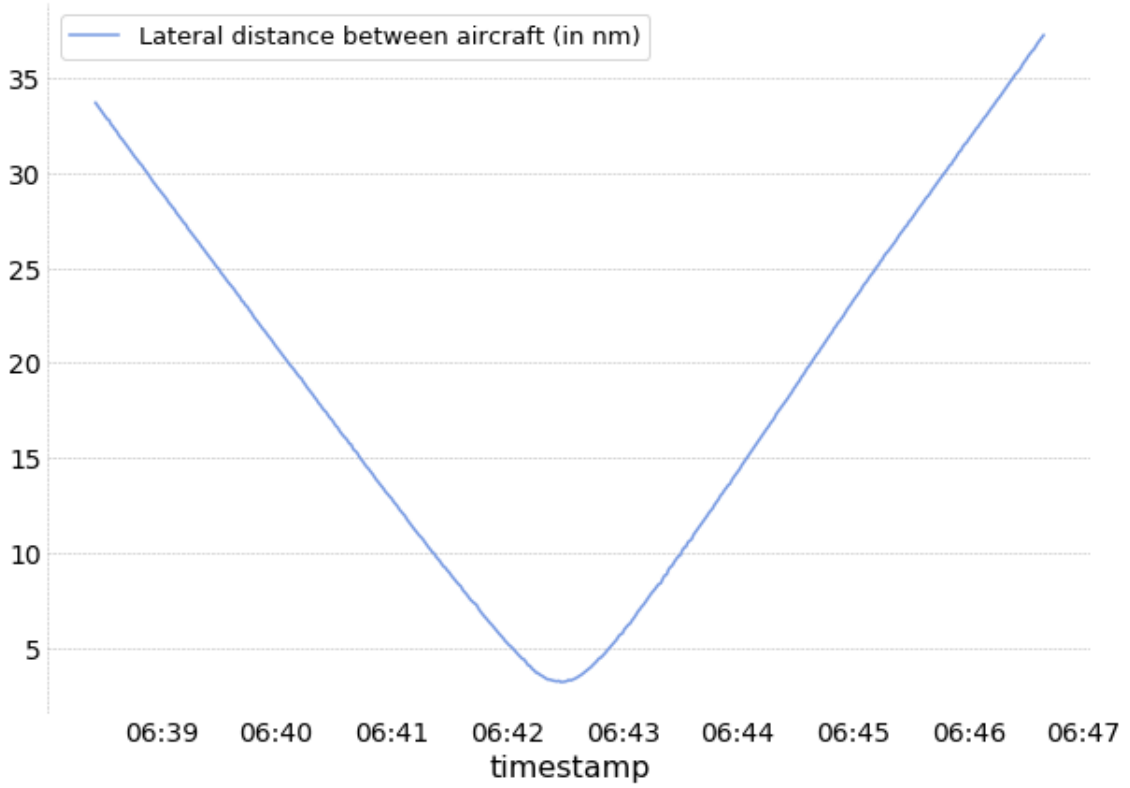
I can check the shape of the trajectory with a good overview of Frankfurt approach up to the North.

The picture is timestamped at 06:42 UTC. Let's have a look of what came around.

```
from datetime import timedelta

p = flight.at("2018-11-15 06:42")

around = opensky.history(
    p.name - timedelta(minutes=5),
    p.name + timedelta(minutes=5),
    bounds=(
        p.longitude - 0.5, p.latitude - 0.5,
        p.longitude + 0.5, p.latitude + 0.5
    )
)
around
```

It seems we get a lot of messages of this AFR1084 flight (Paris to Tunis). Let's plot their lateral distance vs. time.

```
%matplotlib inline
import matplotlib.pyplot as plt

with plt.style.context('traffic'):

    fig, ax = plt.subplots(figsize=(10, 7))

    flight.distance(around['AFR1084']).plot(
        ax=ax, x='timestamp', y='d_horz',
        label="Lateral distance between aircraft (in nm)"
    )
```



Wow, less than 5nm. We came really close! We can now plot a map, with information confirming we were properly separated (1000ft). That is only 300m of altitude difference: it really felt we were flying the same altitude though.

```
from traffic.drawing import Lambert93, countries, rivers, location
from traffic.drawing.markers import rotate_marker, aircraft

with plt.style.context("traffic"):

    fig, ax = plt.subplots(subplot_kw=dict(projection=Lambert93()))

    ax.add_feature(countries())
    ax.add_feature(rivers(linewidth=2.5))
    ax.set_extent((3.9, 6, 44.5, 46))

    location("Lyon").point.plot(
        ax, s=50, marker="*", text_kw=dict(s="Lyon")
```

(continues on next page)

(continued from previous page)

```
    )
    location("Valence, France").point.plot(
        ax, s=50, marker="*", text_kw=dict(s="Valence")
    )
    location("Saint-Étienne").point.plot(
        ax, s=50, marker="*", text_kw=dict(s="Saint-Étienne")
    )

    for i, cs in enumerate(["DLH07F", "AFR1084"]):
        x, *_ = (
            around[cs].before("2018-11-15 06:42").last(minutes=5).plot(ax,␣
↪linewidth=2)
        )

        p = around[cs].at("2018-11-15 06:42")
        p.plot(
            ax,
            s=300,
            marker=rotate_marker(aircraft, p.track),
            color=x.get_color(),
            text_kw=dict(
                s=f"{cs}\nFL{p.altitude/100:.0f}",
                verticalalignment="top" if i % 2 == 0 else "bottom",
                color=x.get_color(),
                fontweight="bold",
            ),
        )

    ax.outline_patch.set_visible(False)
```

10.2 A Mode S based wind field

ADS-B data broadcasted by aircraft contains information about groundspeed and true track angle of its trajectory. When proper requests are sent by a Secondary surveillance radar, aircraft also send more information with true airspeed and heading angle in specific BDS5,0 and BDS6,0 messages (see Junzi Sun's website)

Groundspeeds and true track angles are derived from the GNSS positions whereas true airspeed is computed with traditional onboard instruments like Pitot tubes.

Here, aircraft behave as a distributed network of moving sensors and researchers[1][2] have been recommending methods

[1] Hurter, C., R. Alligier, D. Gianazza, S. Puechmorel, G. Andrienko, and N. Andrienko.
« Wind Parameters Extraction from Aircraft Trajectories ». Computers, Environment and Urban Systems 47 (2014): 28-43.
https://doi.org/10.1016/j.compenvurbsys.2014.01.005.

[2] Sun, Junzi, Huy Vu, Joost Ellerbroek, and Jacco Hoekstra.
« Ground-Based Wind Field Construction from Mode-S and ADS-B Data with a Novel Gas Particle Model », 2017, 9.

to derive wind fields from Mode S data. The following method is a very basic approach to compute wind.

Please note I am not a meteorology specialist, so feel free to improve this page where it should.

```
from traffic.core import Traffic

t = Traffic.from_file("<fill here>")

t_extended = (
    traffic
    # download and decode EHS messages (DF 20/21)
    .query_ehs()
    # resample/interpolate one sample per second
    .resample("1s")
    # median filters
    .filter(altitude=23, track=53, heading=53, groundspeed=53, TAS=53)
    # wind triangle computation
    .compute_wind()
    # median filter
    .filter(wind_u=53, wind_v=53)
    # resample one sample per minute
    .resample("1T")
    # do not use multiprocessing to avoid denial of service
    .eval(desc="preprocessing")
)

# t_extended.to_pickle("wind_backup.pkl")
```

The result of this computation is a set of trajectories: each aircraft yields one point per minute with a 4D-position (timestamp, latitude, longitude, altitude) and a wind vector decomposed along a zonal speed (*wind_u*) and a meridional speed (*wind_v*).

We then use ipyleaflet to display the wind field. The *Velocity* widget requires two 2D matrices of zonal and meridional components of the wind. The following method rounds lat/lon coordinates to the closest integer and average wind in each resulting cell.

```
import xarray as xr

def compute_grid(traffic: Traffic) -> xr.Dataset:

    avg = (
        traffic
        # remove NaN values, just in case
        .query("wind_u == wind_u")
        # prepare coordinates for the 4d-grid, also remove NaN in wind
        .assign(
            # round coordinates to the closest .33 latitude/longitude
            lat_=lambda df: (3 * df.latitude.round(0)) / 3,
            lon_=lambda df: (3 * df.longitude.round(0)) / 3,
            # This basic version averages on all altitudes/timeranges
            # but it is easy to use the following fields to display
            # wind fields in particular time ranges and altitude levels.
            alt_=lambda df: (3e-3 * df.altitude).round(0) / 3e-3,
            hour=lambda df: df.timestamp.dt.round("h"),
        )
        # compute the average wind
        .data[["wind_u", "wind_v", "lat_", "lon_"]]
        .groupby(["lat_", "lon_"])
```

(continues on next page)

(continued from previous page)

```
        .mean()
    )

    # Unstack then fill the holes where possible (2D interpolation)
    u = avg[["wind_u"]].unstack().interpolate().values
    v = avg[["wind_v"]].unstack().interpolate().values

    return xr.Dataset(
        data_vars={
            "u_wind": xr.DataArray(u, coords=avg.index.levels),
            "v_wind": xr.DataArray(v, coords=avg.index.levels),
        }
    )
```

The following is a basic rendering delegated to ipyleaflet library.

```
from ipyleaflet import Map, Velocity, basemaps

# t_extended = Traffic.from_file("wind_backup.pkl")

map_ = Map(
    center=(52, 15),
    zoom=4,
    interpolation="nearest",
    basemap=basemaps.CartoDB.DarkMatter,
)

wind = Velocity(
    data=compute_grid(t_extended),
    zonal_speed="u_wind",
    meridional_speed="v_wind",
    latitude_dimension="lat_",
    longitude_dimension="lon_",
    velocity_scale=0.002,
    max_velocity=150,
)

map_.add_layer(wind)

map_
```

The example above is wind averaged between 25°W and 55°E and between 32°N and 65°N, from FL200 and above on February 23th 2019, between 14:00 and 16:30 UTC.

10.3 Calibration flights

What is this plane doing?

This was my first reaction after hitting on the following trajectory during an analysis of approaches at Toulouse airport. After exchanges with ATC people, I learned that these trajectories are flown by small aircraft working at calibrating landing assistance systems, including ILS and VOR.

These trajectories mostly consist of many low passes over an airport and large circles or arcs of circle. A small sample of such trajectories is included in traffic.data.samples, and the following snippet of code will let you explore those before an attempt of explanation.

You may click on trajectories for more information.

10.3.1 A short introduction to radio-navigation

A number of navigational systems emerged in the second half of the 20th century, mainly based on VHF communications. Some of them were designed for surveillance, like Secondary surveillance radars; other systems also came to assist pilot in navigation and landing.

VOR (VHF Omnidirectional Range) ground stations send an omnidirectional master signal (on a predefined frequency determined for each station) and a second highly directional signal. Aircraft measure the phase difference between the two signals, which corresponds to the bearing from the station to the aircraft.[1]

Historically, airways were laid out between VORs, which stand at the intersections between those routes. Advances in GNSS (understand GPS) make these stations less necessary.

VOR stations often host a DME (Distance Measuring Equipment). The principle is similar to radar ranging, except the roles of the aircraft and of the ground station are reversed. The aircraft sends a signal to the DME; the DME repeats the same signal 50 μs after reception. When the aircraft receives a copy of the sent messages, it measures the time of travel to the DME, subtracts 50 μs and divides the results by 2: speed of light gives an estimation of the distance between the aircraft to the ground station.

ILS (Instrument Landing Systems) consists of two guidance systems: a lateral one (the LOC, for *localizer*) and a vertical one (the GS, for *glide slope*, also *glide path*). The localizer usually consists of several pairs of directional antennas placed beyond the departure end of the runway.[2]

Local authorities define very strict thresholds for accuracy: internal monitoring shall switch off the system if the accuracy of the signal is not appropriate. All radio-navigation beacons (including VOR, DME and ILS) are checked periodically by specially equipped aircraft. In particular, the VOR test consists of flying around the beacon in circles at defined distances and along several radials.

10.3.2 A basic analysis of VOR calibration trajectories

We can have a look at the first trajectory in the calibration dataset. The aircraft takes off from Ajaccio airport before flying concentric circles and radials. There must be a VOR around, we can search in the navaid database:

```
# see https://traffic-viz.github.io/samples.html if any issue on import
from traffic.data.samples.calibration import ajaccio
from traffic.data import navaids

navaids.extent(ajaccio).query('type == "VOR"')
```

Next step is to compute for each point the distance and bearing from the VOR to each point of the trajectory. The parts of the trajectory that are of interest are the ones with little to no variation in the distance (circles) and in the bearing (radials) to the VOR.

```
vor = navaids.extent(ajaccio)['AJO']

ajaccio = (
    ajaccio.distance(vor)  # add a distance column (in nm) w.r.t the VOR
    .bearing(vor)  # add a bearing column w.r.t the VOR
    .assign(
        distance_diff=lambda df: df.distance.diff().abs(),  # large circles
        bearing_diff=lambda df: df.bearing.diff().abs(),  # long radials
```

(continues on next page)

[1] Decoding VOR signals can be a fun exercice for the amateur software radio developper. (link)
[2] Check for them next time you drive around an airport!

(continued from previous page)

```
    )
)
```

We can write a simple .query() followed by a .split() method to select all segments with a constant bearing with respect to the selected VOR.

```
for segment in ajaccio.query('bearing_diff < .01').split('1T'):
    if segment.longer_than('5 minutes'):
        print(segment.duration)

# 0 days 00:05:05
# 0 days 00:05:10
# 0 days 00:17:20
# 0 days 00:22:35
# 0 days 00:05:20
# 0 days 00:09:40
# 0 days 00:08:15
```

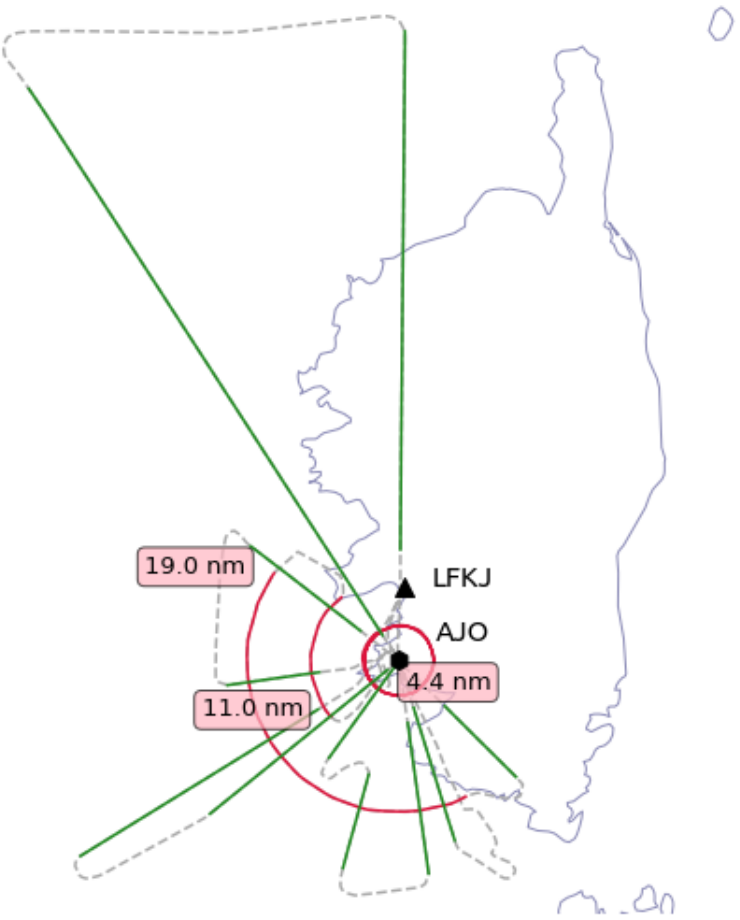
We have all we need to enhance the interesting parts of the trajectory now:

```
%matplotlib inline

import matplotlib.pyplot as plt
import pandas as pd

from traffic.drawing import Lambert93, countries
from traffic.data import airports

point_params = dict(zorder=5, text_kw=dict(fontname="Ubuntu", fontsize=15))
box_params = dict(boxstyle="round", facecolor="lightpink", alpha=.7, zorder=5)

with plt.style.context("traffic"):

    fig, ax = plt.subplots(subplot_kw=dict(projection=Lambert93()))

    ax.add_feature(countries(edgecolor="midnightblue"))

    airports["LFKJ"].point.plot(ax, marker="^", **point_params)
    shift_vor = dict(units="dots", x=20, y=10)
    vor.plot(ax, marker="h", shift=shift_vor, **point_params)

    # background with the full trajectory
    ajaccio.plot(ax, color="#aaaaaa", linestyle="--")

    # plot large circles in red
    for segment in ajaccio.query("distance_diff < .02").split("1 minute"):
        # only print the segment if it is long enough
        if segment.longer_than("3 minutes"):
            segment.plot(ax, color="crimson")
            distance_vor = segment.data.distance.mean()

            # an annotation with the radius of the circle
            segment.at().plot(
                ax, alpha=0,  # We don't need the point, only the text
                text_kw=dict(s=f"{distance_vor:.1f} nm", bbox=box_params)
            )
```

(continues on next page)

(continued from previous page)

```
    for segment in ajaccio.query("bearing_diff < .01").split("1 minute"):
        # only print the segment if it is long enough
        if segment.longer_than("3 minutes"):
            segment.plot(ax, color="forestgreen")

    ax.set_extent((7.6, 9.9, 41.3, 43.3))
    ax.outline_patch.set_visible(False)
    ax.background_patch.set_visible(False)
```



The following map displays the result of a similar processing on the other VOR calibration trajectories from the sample dataset.[3]

10.3.3 Equipped aircraft for beacon calibration

This list only contains the equipped aircraft for the calibration in the sample dataset. Apart from F-HNAV, registration numbers were found on social networks. Two of the aircraft registrations were not in the provided database at the time

[3] Time, distance and bearing thresholds may need further ajustments for a proper picture. Note the kiruna14 seems to circle around a position that is not referenced in the database. Any help or insight welcome!

of the writing, so we added them manually.

```
from traffic.data.samples.calibration import traffic as calibration
from traffic.data import aircraft

# aircraft not in junzis database
other_aircraft = {"4076f1": "G-TACN (DA62)", "750093": "9M-FCL (LJ60)"}

(
    calibration.groupby(["flight_id"], as_index=False)
    .agg({"timestamp": "min", "icao24": "first"})
    .assign(
        registration=lambda df: df.icao24.apply(
            lambda x: f"{aircraft[x].regid.item()} ({aircraft[x].mdl.item()})"
            if aircraft[x].shape[0] > 0
            else other_aircraft.get(x, None)
        ),
        flight_id=lambda df: df.agg(
            # not the most efficient way but quite readable
            lambda x: f"{x.flight_id} ({x.timestamp:%Y-%M-%d})", axis=1
        )
    )
    .sort_values(["registration", "timestamp"])
    .groupby(["registration", "icao24"])
    .apply(lambda df: ", ".join(df.flight_id))
    .pipe(lambda series: pd.DataFrame({"flights": series}))
)
```

10.4 Cross-wind landing and geomagnetic declination

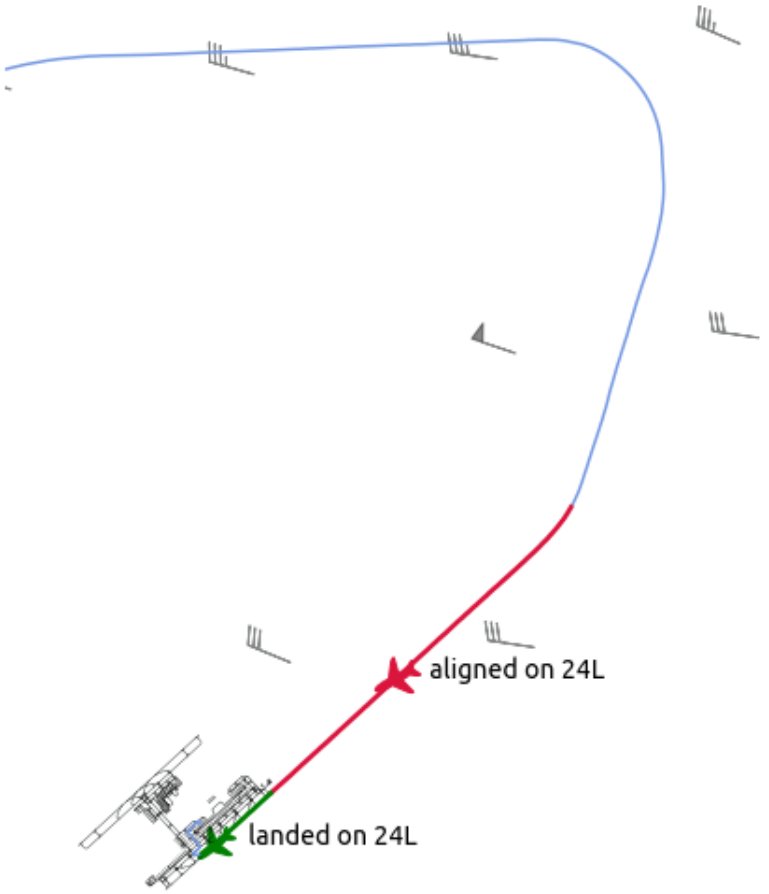
Warning: The following code snippets including experimental features which are not available in the library at the time of the writing, but may appear in a near future.

The following situation refers to a flight I boarded, bound for Kansai airport, Japan (RJBB). The flight encountered reasonable cross wind during final approach. The regular behaviour then is to perform a crab approach, then to remove the crab angle during the touchdown.

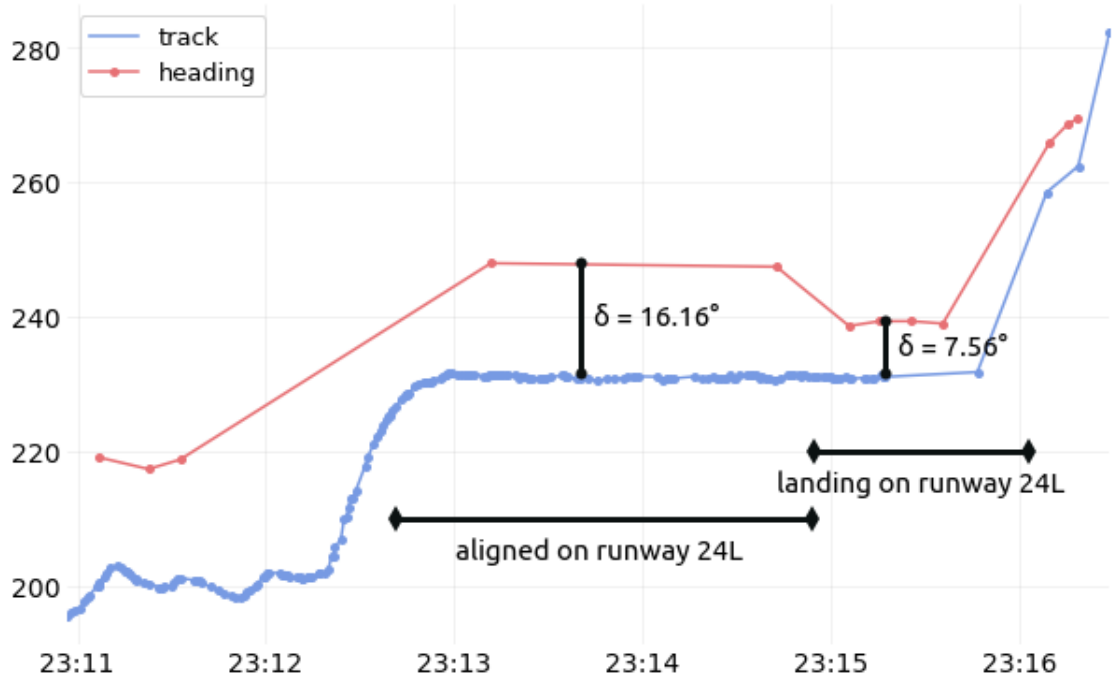
Let's have a look at the data. The following simply plots a map of the last minutes of flight, together with a reconstructed windfield. (code below)

Together with timestamps corresponding to the catching of the ILS signal and of the landing, we can plot both track and heading angles of the aircraft. Both angles, together with ground and true air speed measures help construct windfields in general.

Note: See Flight.compute_wind() and Flight.plot_wind().



Here, since we have very few measurements points for the heading angle, we did not resample the trajectories prior to treatments. Let's observe the granular data as decoded onboard (code below)



There are, for sure, few points requested with heading angles. Yet we can clearly see that:

- the differences between the two angles (about 16°) when the aircraft is aligned on the ILS reflects the crab approach;
- the moment when the aircraft removes the crab angle appears clearly after landing, yet it is not going down to zero (about 7.5°).

Warning: The explanation lies in a subtle nuance in the Mode S specifications:

- ADS-B broadcasts a **true track angle** (i.e. with respect to the **geographical North**)
- BDS 5,0 replies with a **magnetic heading** (i.e. with respect to the **magnetic North**)

Looking at the World Magnetic Model for the declination map, the resulting 7.5° looks conformal to the model!

```
# Select the first segment aligned on any ILS for RJBB
ils = next(flight.aligned_on_ils("RJBB"))
# Select the most reasonable runway used for landing or take-off.
rwy = flight.on_runway("RJBB")
```

```
from traffic.core.projection import Mercator

from traffic.data import airports, navaids

from traffic.drawing import countries, location
from traffic.drawing.markers import rotate_marker, aircraft


with plt.style.context("traffic"):

    fig, ax = plt.subplots(subplot_kw=dict(projection=Mercator()))
```

(continues on next page)

(continued from previous page)

```
    airports["RJBB"].plot(ax, edgecolor="#0e1111", linewidth=0.4, alpha=0.9)

    flight.plot(ax, lw=1.5)

    # The segment aligned on ILS in one color
    x, *_ = ils.plot(ax, lw=2.5, color="crimson")
    ils.at_ratio(0.6).plot(
        ax,
        s=700,
        color=x.get_color(),
        marker=rotate_marker(aircraft, ils.at_ratio(0.6).heading),
        text_kw=dict(s=" aligned on 24L", fontname="Ubuntu", fontsize=14),
    )

    # The segment when the aircraft is on the runway in another color
    x, *_ = rwy.plot(ax, lw=2.5, color="green")
    rwy.at_ratio(0.6).plot(
        ax,
        s=500,
        marker=rotate_marker(aircraft, rwy.at_ratio(0.6).heading - 7.56),
        color=x.get_color(),
        text_kw=dict(s=" landed on 24L", fontname="Ubuntu", fontsize=14),
    )

    (
        flight  # .assign(heading=lambda df: df.heading - 7.56)
        .resample("1s")
        .compute_wind()
        .query("altitude > 500")
        .plot_wind(
            ax, resolution=dict(latitude=15, longitude=15), alpha=0.5, color="#0e1111
↪",
        )
    )

    ax.set_extent((135.2, 135.4, 34.4, 34.62))
    ax.outline_patch.set_visible(False)
```

```
from matplotlib.dates import DateFormatter

center_15 = dict(fontname="Ubuntu", fontsize=15, horizontalalignment="center")
marker_style = dict(color="#0b1111", linewidth=3, marker="o", markersize=6)

with plt.style.context("traffic"):

    fig, ax = plt.subplots(figsize=(10, 7))

    flight.plot_time(ax, ["track", "heading"], marker=".")

    # Annotate the different phases of landing
    ax.plot([ils.start, ils.stop], [210, 210], color="#0b1111", linewidth=3, marker="d
↪")
    ax.text(ils.start + ils.duration / 2, 204, "aligned on runway 24L", **center_15)

    ax.plot([rwy.start, rwy.stop], [220, 220], color="#0b1111", linewidth=3, marker="d
↪")
```

(continues on next page)

(continued from previous page)

```
    ax.text(rwy.start + rwy.duration / 2, 214, "landing on runway 24L", **center_15)

    # Annotate the differences
    ax.plot(
        [ils.start + ils.duration / 2, ils.start + ils.duration / 2,],
        [ils.max("heading"), ils.max("track")], **marker_style
    )

    ax.plot(
        [rwy.start + rwy.duration / 3, rwy.start + rwy.duration / 3,],
        [rwy.max("heading"), rwy.max("track")], **marker_style
    )

    ax.text(
        ils.start + ils.duration / 2,
        (ils.max("heading") + ils.max("track")) / 2,
        f"  δ = {ils.max('heading') - ils.max('track'):.2f}°",
        horizontalalignment="left", **center_15
    )

    ax.text(
        rwy.start + rwy.duration / 3,
        (rwy.max("heading") + rwy.max("track")) / 2,
        f"  δ = {rwy.max('heading') - rwy.max('track'):.2f}°",
        horizontalalignment="left", **center_15
    )

    # Simplify the formatting for dates on the x-axis
    ax.xaxis.set_major_formatter(DateFormatter("%H:%M"))
    fig.autofmt_xdate(rotation=0, ha="center")
```

CHAPTER 11

Publications

If you find this project useful for your research and use it in an academic work, you may cite it as:

```
@article{olive2019traffic,
    author={Xavier {Olive}},
    journal={Journal of Open Source Software},
    title={traffic, a toolbox for processing and analysing air traffic data},
    year={2019},
    volume={4},
    pages={1518},
    doi={10.21105/joss.01518},
    issn={2475-9066},
}
```

The following list contains publications from research using the traffic library:

- M. Schultz, X. Olive, J. Rosenow, H. Fricke, S. Alam.
  Analysis of airport ground operations based on ADS-B data. *Proceedings of the 1st conference on Artificial Intelligence and Data Analytics in Air Transportation* (AIDA-AT), 2020
- M. Schultz, J. Rosenow and X. Olive.
  A-CDM Lite: situation awareness and decision making for small airports based on ADS-B data.
  Proceedings of the 9th SESAR Innovation Days, 2019.
  (paper)
- X. Olive and L. Basora.
  Air Traffic Data Processing using Python: Trajectory Clustering.
  Proceedings of the 7th OpenSky Workshop, 2019.
  (paper) (notebook) https://doi.org/10.29007/sf1f
- M. Schäfer, X. Olive, M. Strohmeier, M. Smith, I. Martinovic, V. Lenders.
  OpenSky Report 2019: Analysing TCAS in the Real World using Big Data.
  Proceedings of the 38th Digital Avionics Systems Conference (DASC), 2019

(paper)

- X. Olive and L. Basora
  Identifying Anomalies in past en-route Trajectories with Clustering and Anomaly Detection Methods.
  Proceedings of the 13th Air Traffic Management R&D Seminar, 2019
  (paper) (notebook)
- X. Olive, J. Grignard, T. Dubot and J. Saint-Lot.
  Detecting Controllers' Actions in Past Mode S Data by Autoencoder-Based Anomaly Detection.
  Proceedings of the 8th SESAR Innovation Days, 2018
  (paper) (notebook)
- X. Olive and P. Bieber.
  Quantitative Assessments of Runway Excursion Precursors using Mode S Data. *Proceedings of the 8th International Conference on Research in Air Transportation*, 2018 (Best paper award)
  (paper) (notebook)
- X. Olive and J. Morio.
  Trajectory clustering of air traffic flows around airports.
  Aerospace Science and Technology 84, 2019, pp. 776–781.
  https://doi.org/10.1016/j.ast.2018.11.031

Links

11.1 Identifying Anomalies in past en-route Trajectories with Clustering and Anomaly Detection Methods

Xavier Olive and Luis Basora

This notebook comes with the paper published at ATM Seminar 2019.
Details are presented in the paper. The following code is provided for reproducibility concerns.

11.1.1 Data preparation

```
import numpy as np
from traffic.core import Traffic

t = (
    # trajectories during the opening hours of the sector
    Traffic.from_file("data/LFBBPT_flights_2017.pkl")
    .clean_invalid()
    # first cleaning and interpolation of positions when points are missing
    .resample("1s")
    # come back to 50 samples per trajectory
    .resample(50)
    # trajectory clustering needs the log of the altitude
    .assign(
        log_altitude=lambda x: x.altitude.apply(lambda x: x if x == 0 else np.
↪log10(x))
    )
    # lazy evaluation multiprocessed on 12 cores
```

(continues on next page)

(continued from previous page)

```
    .eval(max_workers=12)
)
```

The data has been downloaded from the OpenSky Network Impala database.

First a clustering is applied to the dataset. The implementation of the specific clustering described in the paper is available on the following github repository

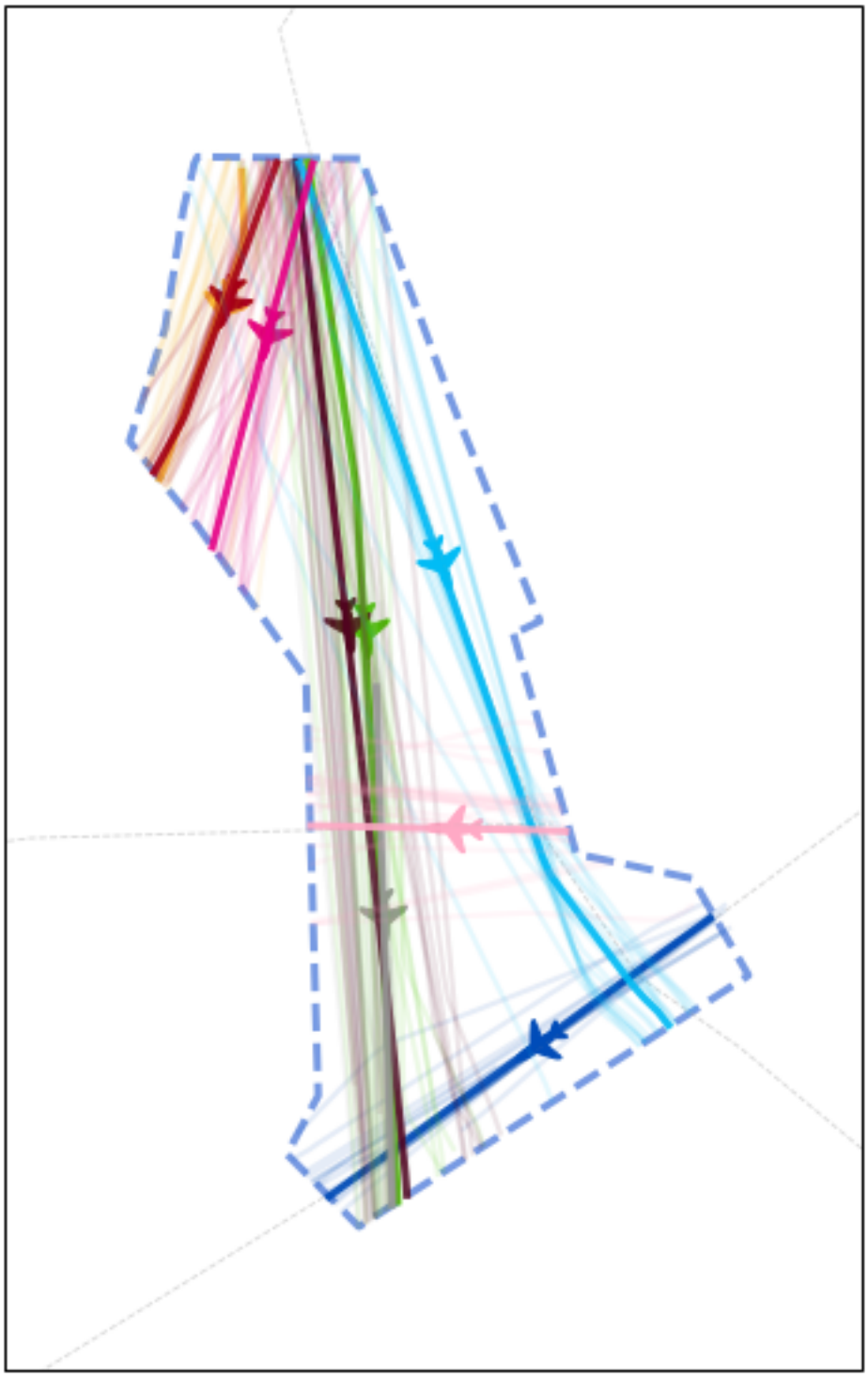
```
from traffic.core.projection import Lambert93

# pip install git+https://github.com/lbasora/sectflow
from sectflow.clustering import TrajClust

features = ["x", "y", "latitude", "longitude", "altitude", "log_altitude"]
clustering = TrajClust(features)

# use the clustering API from traffic
t_cluster = t.clustering(
    nb_samples=2, features=features, projection=Lambert93(), clustering=clustering
).fit_predict(max_workers=12)
```

```
# Color distribution by cluster
from itertools import cycle, islice

n_clusters = 1 + t_cluster.data.cluster.max()
color_cycle = cycle(
    "#fbbb35 #004cb9 #4cc700 #a50016 #510420 #01bcf5 #999999 #e60085 #ffa9c5".split()
)
colors = list(islice(color_cycle, n_clusters))
colors.append("#aaaaaa")  # color for outliers, if any

import matplotlib.pyplot as plt
from random import sample

from traffic.data import airways, aixm_airspaces
from traffic.drawing.markers import rotate_marker, atc_tower, aircraft

with plt.style.context("traffic"):
    fig, ax = plt.subplots(1, figsize=(15, 10), subplot_
↪kw=dict(projection=Lambert93()))

    aixm_airspaces["LFBBPT"].plot(
        ax, linewidth=3, linestyle="dashed", color="steelblue"
    )
    for name in "UN460 UN869 UM728".split():
        airways[name].plot(ax, linestyle="dashed", color="#aaaaaa")

    # do not plot outliers
    for cluster in range(n_clusters):

        current_cluster = t_cluster.query(f"cluster == {cluster}")

        # plot the centroid of each cluster
        centroid = current_cluster.centroid(50, projection=Lambert93())
        centroid.plot(ax, color=colors[cluster], alpha=0.9, linewidth=3)
        centroid_mark = centroid.at_ratio(0.45)
```

(continues on next page)

(continued from previous page)

```
        # little aircraft
        centroid_mark.plot(
            ax,
            color=colors[cluster],
            marker=rotate_marker(aircraft, centroid_mark.track),
            s=500,
            text_kw=dict(s=""),  # no text associated
        )

        # plot some sample flights from each cluster
        sample_size = min(20, len(current_cluster))
        for flight_id in sample(current_cluster.flight_ids, sample_size):
            current_cluster[flight_id].plot(
                ax, color=colors[cluster], alpha=0.1, linewidth=2
            )

    # TODO improve this: extent with buffer
    ax.set_extent(
        tuple(
            x - 0.5 + (0 if i % 2 == 0 else 1)
            for i, x in enumerate(aixm_airspaces["LFBBPT"].extent)
        )
    )

    # Equivalent of Fig. 5
```



11.1.2 Machine-Learning

The anomaly detection method is based on a stacked autoencoder (PyTorch implementation).

```
import torch
from torch import nn, optim, from_numpy, rand
from torch.autograd import Variable

from sklearn.preprocessing import minmax_scale
from tqdm.autonotebook import tqdm


# Stacked autoencoder

class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(50, 24), nn.ReLU(), nn.Linear(24, 12), nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(12, 24), nn.ReLU(), nn.Linear(24, 50), nn.Sigmoid()
        )

    def forward(self, x, **kwargs):
        x = x + (rand(50).cuda() - 0.5) * 1e-3  # add some noise
        x = self.encoder(x)
        x = self.decoder(x)
        return x

# Regularisation term introduced in IV.B.2

def regularisation_term(X, n):
    samples = torch.linspace(0, X.max(), 100, requires_grad=True)
    mean = samples.mean()
    return torch.relu(
        (torch.histc(X) / n * 100 - 1 / mean * torch.exp(-samples / mean))
    ).mean()

# ML part

def anomalies(t: Traffic, cluster_id: int, lambda_r: float, nb_it: int = 10000):

    t_id = t.query(f"cluster=={cluster_id}")

    flight_ids = list(f.flight_id for f in t_id)
    n = len(flight_ids)
    X = minmax_scale(np.vstack(f.data.track[:50] for f in t_id))

    model = Autoencoder().cuda()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)

    for epoch in tqdm(range(nb_it), leave=False):

        v = Variable(from_numpy(X.astype(np.float32))).cuda()

        output = model(v)
        distance = nn.MSELoss(reduction="none")(output, v).sum(1).sqrt()

        loss = criterion(output, v)
```

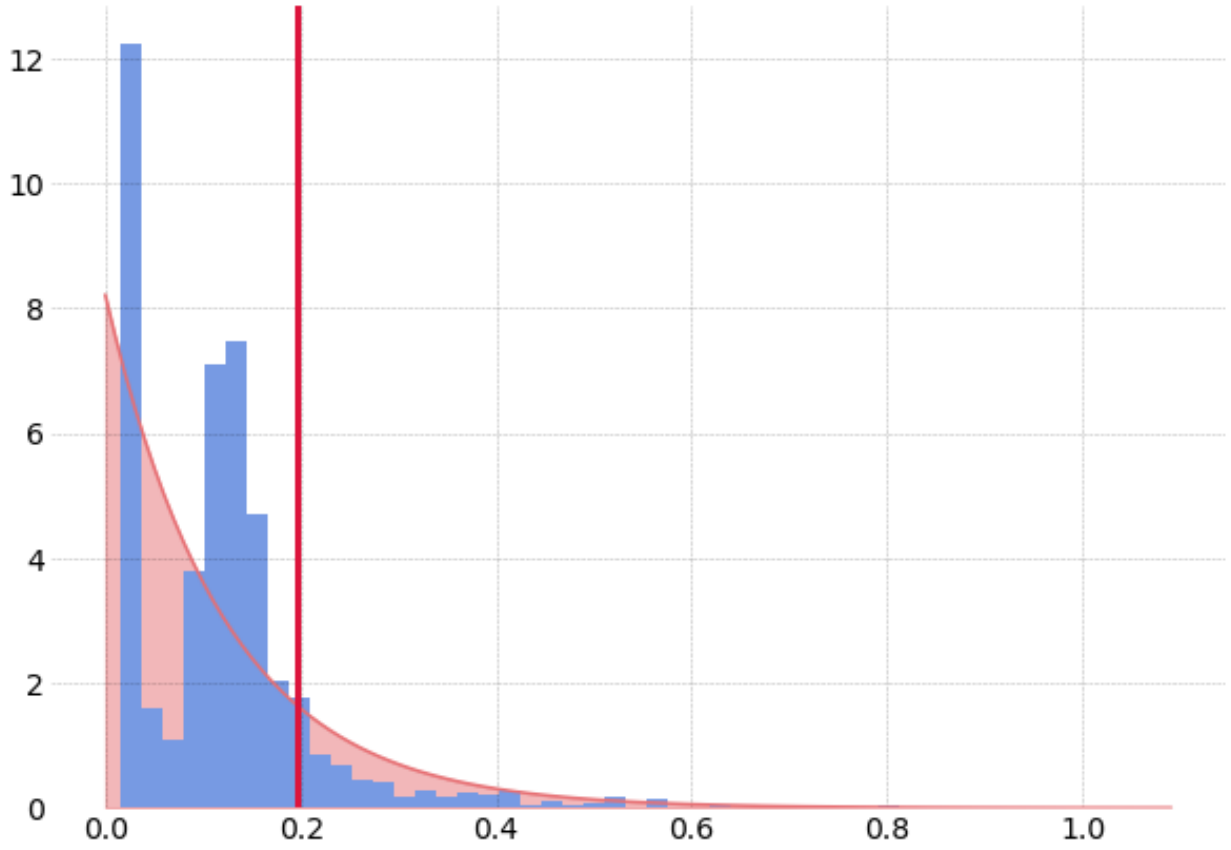
(continues on next page)

(continued from previous page)

```
        # regularisation
        loss = (
            lambda_r * regularisation_term(distance.cpu().detach(), n)
            + criterion(output, v).cpu()
        )

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    output = model(v)
    return (
        (nn.MSELoss(reduction="none")(output, v).sum(1)).sqrt().cpu().detach().numpy()
    )


# no regularisation for this plot
output = anomalies(t_cluster, 3, lambda_r=0, nb_it=3000)
```

The following code plots the distribution of reconstruction errors without regularisation, resulting in two modes in the distribution. The `lambda_r` parameter helps reducing this trend.

```
from scipy.stats import expon

# Equivalent of Fig. 4

with plt.style.context("traffic"):
    fig, ax = plt.subplots(1, figsize=(10, 7))
    hst = ax.hist(output, bins=50, density=True)
    mean = output.mean()
    x = np.arange(0, output.max(), 1e-2)
    e = expon.pdf(x, 0, output.mean())

    ax.plot(x, e, color="#e77074")
    ax.fill_between(x, e, zorder=-2, color="#e77074", alpha=0.5)
    ax.axvline(
        output.mean() * np.log(5), color="crimson", linestyle="solid", linewidth=3
    )
```



11.2 Detecting Controllers' Actions in Past Mode S Data by Autoencoder-Based Anomaly Detection

Xavier Olive, Jeremy Grignard, Thomas Dubot and Julie Saint-Lot

This notebook comes with the paper published at Sesar Innovation Days 2018.
Details are presented in the paper. The following code is provided for reproducibility concerns.

11.2.1 The dataset

The basic dataset consists of one year of trajectories between Paris Orly and Toulouse Blagnac airports. A learning box is arbitrarily defined so as to include about 20 minutes of flight before entering the TMA.

```
from traffic.core import Traffic
from shapely.geometry import box

# Data: one year of traffic between LFPO and LFBO
t = Traffic.from_file("data/2017_lfpo_lfbo.pkl")

# The learning box has been set manually on the data
learning_box = box(
```

(continues on next page)

(continued from previous page)

```
    0.9488888502120971,
    44.287776947021484,
    2.748888850212097,
    45.537776947021484
)
```

The data has been downloaded from the OpenSky Network Impala database. The following recalls all the callsigns selected for this route.

```
# Pretty-print all the callsigns analysed

cs = sorted(t.callsigns)
for i in range(4):
    print(" ".join(cs[7*i:7*(i+1)]))
```

```
AF128UU AFR21ME AFR22MT AFR22SR AFR26SA AFR27GH AFR38DV
AFR43LC AFR47FW AFR49DL AFR51ZU AFR52BW AFR53TQ AFR57XL
AFR59UB AFR611H AFR613R AFR61JJ AFR61UJ AFR65WA AFR77PQ
AFR88DM EZY24EH EZY289N EZY353H EZY4019 EZY4029 EZY98KP
```

11.2.2 Data preparation

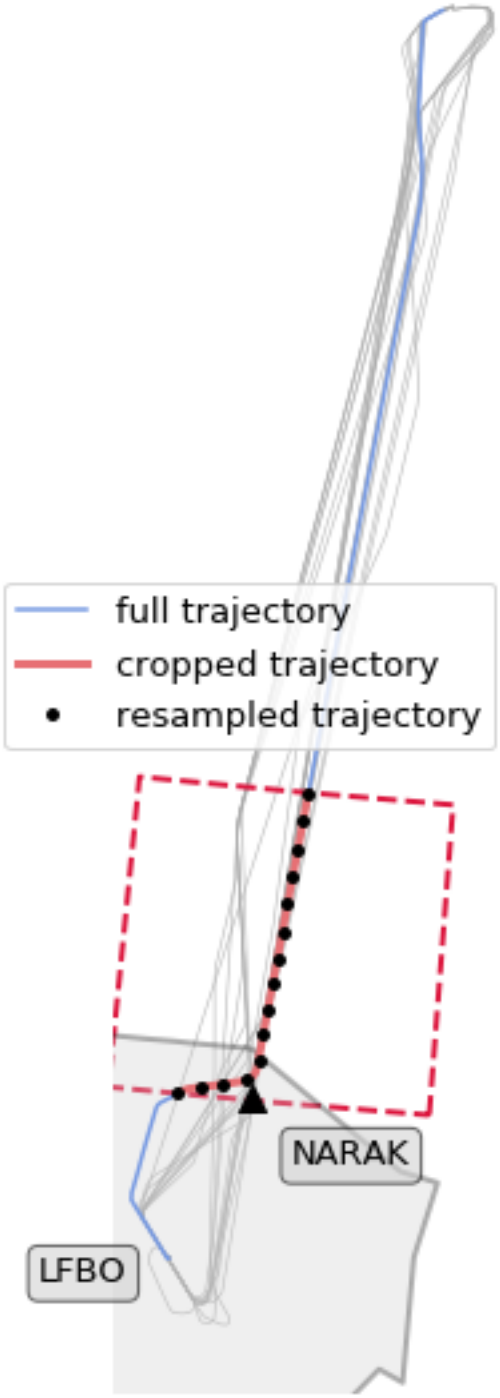
The following plot explains how each trajectory is prepared. The basic idea is to clip the trajectory to the learning box, then to resample each trajectory to a fixed number of samples (15 on the figure, 150 for the machine learning method).

```
%matplotlib inline
import matplotlib.pyplot as plt

from traffic.data import airports, navaids, nm_airspaces
from traffic.drawing import EuroPP, PlateCarree

with plt.style.context("traffic"):

    fig, ax = plt.subplots(subplot_kw=dict(projection=EuroPP()))

    ax.add_geometries(
        [learning_box],
        crs=PlateCarree(),
        facecolor="None",
        edgecolor="crimson",
        linewidth=2,
        linestyle="dashed",
    )

    # TMA of Toulouse Airport
    nm_airspaces["LFBOTMA"].plot(
        ax, edgecolor="black", facecolor="#cccccc", alpha=.3, linewidth=2
    )

    # Beacon marking the beginning of the STAR procedure
    navaids["NARAK"].plot(
        ax,
        zorder=2,
        marker="^",
```

(continues on next page)

(continued from previous page)

```
        shift=dict(units="dots", x=15, y=-15),
        text_kw={
            "s": "NARAK",
            "verticalalignment": "top",
            "bbox": dict(facecolor="lightgray", alpha=0.6, boxstyle="round"),
        },
    )

    airports["LFBO"].point.plot(
        ax,
        shift=dict(units="dots", x=-15),
        alpha=0,
        text_kw=dict(
            s="LFBO",
            verticalalignment="top",
            horizontalalignment="right",
            bbox=dict(facecolor="lightgray", alpha=0.6, boxstyle="round"),
        ),
    )

    # Few trajectories from the data set
    t.plot(ax, color="#aaaaaa", zorder=-2, linewidth=.6, nb_flights=20)

    # Details of the data preparation
    t["EZY24EH_2831"].plot(ax, linewidth=1.5, label="full trajectory")
    t["EZY24EH_2831"].clip(learning_box).plot(
        ax, linewidth=3, label="cropped trajectory"
    )
    t["EZY24EH_2831"].clip(learning_box).resample(15).plot(
        ax,
        linewidth=0,
        marker=".",
        color="black",
        label="resampled trajectory",
    )

    ax.legend()
    ax.outline_patch.set_visible(False)
    ax.background_patch.set_visible(False)
```



The following applies the preprocessing to each trajectory in the dataset.

```
t_clip = Traffic.from_flights(
    flight
    # Median filters on all trajectories
    .filter()
```

(continues on next page)

(continued from previous page)

```
    # Clipping to the learning box
    .clip(learning_box)
    # Resample to 150 samples per flight
    .resample(150)
    for flight in t
)
# Backup to one file
t_clip.to_pickle("data/2017_lfpo_lfbo_prepared.pkl")
```

```
t_clip = Traffic.from_file("data/2017_lfpo_lfbo_prepared.pkl")
t_clip
```

11.2.3 Machine-Learning

The anomaly detection method is based on a shallow autoencoder (PyTorch implementation). For the sake of this example, we focus on the track angle signal. At the end of the training period, we look at the distribution of the reconstruction errors.

```
import numpy as np

from sklearn.preprocessing import minmax_scale
from torch import from_numpy, nn, optim
from torch.autograd import Variable
from tqdm.autonotebook import tqdm

# -- Autoencoder architecture --

class Autoencoder(nn.Module):
    """Basic shallow autoencoder."""
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Linear(150, 64),
            # Activation function
            nn.Sigmoid()
        )
        self.decoder = nn.Sequential(
            nn.Linear(64, 150),
            # Activation function
            nn.Sigmoid()
        )

    def forward(self, x, **kwargs):
        x = self.encoder(x)
        x = self.decoder(x)
        return x


def anomalies(t: Traffic):

    flight_ids = list(f.flight_id for f in t)

    # For this example, we only work on the track angle signal
    X = minmax_scale(np.vstack(f.data.track for f in t))
```

(continues on next page)

(continued from previous page)

```
    model = Autoencoder().cuda()
    criterion = nn.MSELoss()
    optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)

    # Basic training process (GPU)
    loss_evolution = []
    for epoch in tqdm(range(2000)):

        v = Variable(from_numpy(X.astype(np.float32))).cuda()
        output = model(v)
        loss = criterion(output, v)
        loss_evolution.append(loss.data.item())

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    output = model(v)
    # Compute the reconstruction error for each flight
    errors = dict(
        (id_, err)
        for id_, err in zip(
            flight_ids,
            (nn.MSELoss(reduction="none")(output, v).sum(1))
            .sqrt()
            .cpu()
            .detach()
            .numpy(),
        )
    )
    return errors, loss_evolution
```

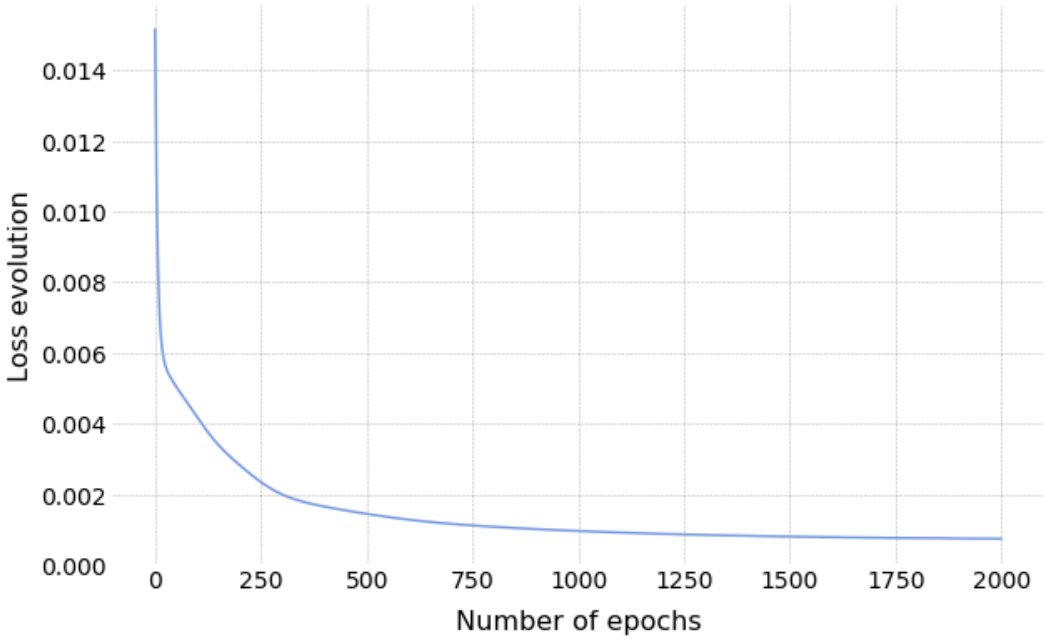
Now we apply the anomaly detection on the preprocessed data and analyse the distribution as explained in the paper.

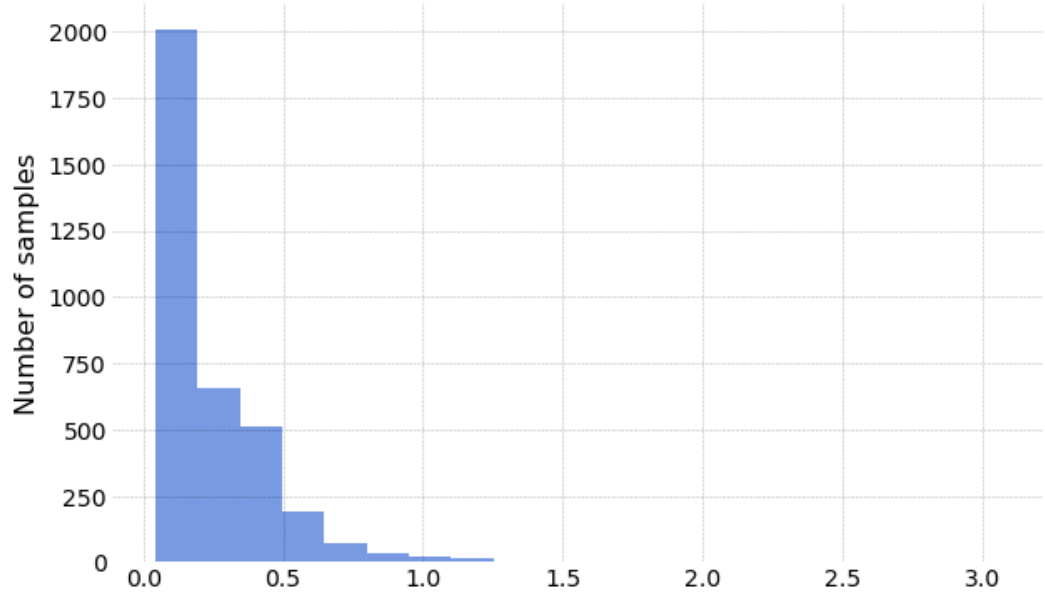
```
errors, loss_evolution = anomalies(t_clip)

with plt.style.context("traffic"):
    fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 21))
    ax1.plot(loss_evolution)
    ax2.hist(errors.values(), bins=20)
    h = ax3.hist(errors.values(), bins=20)
    ax3.set_yscale("log")

    for i, id_ in enumerate(
        [
            "EZY24EH_3324",
            "AFR47FW_2174",
            "AFR61UJ_1321",
            "AFR27GH_348",
            "AFR51ZU_027",
        ]
    ):
        ax3.annotate(
            f"{t[id_].callsign}, {t[id_].stop:%b %d}",
            xy=(errors[id_], h[0][sum(h[1] - errors[id_] < 0) - 1]),
            xytext=(errors[id_], 2 ** (i + 3)),
```

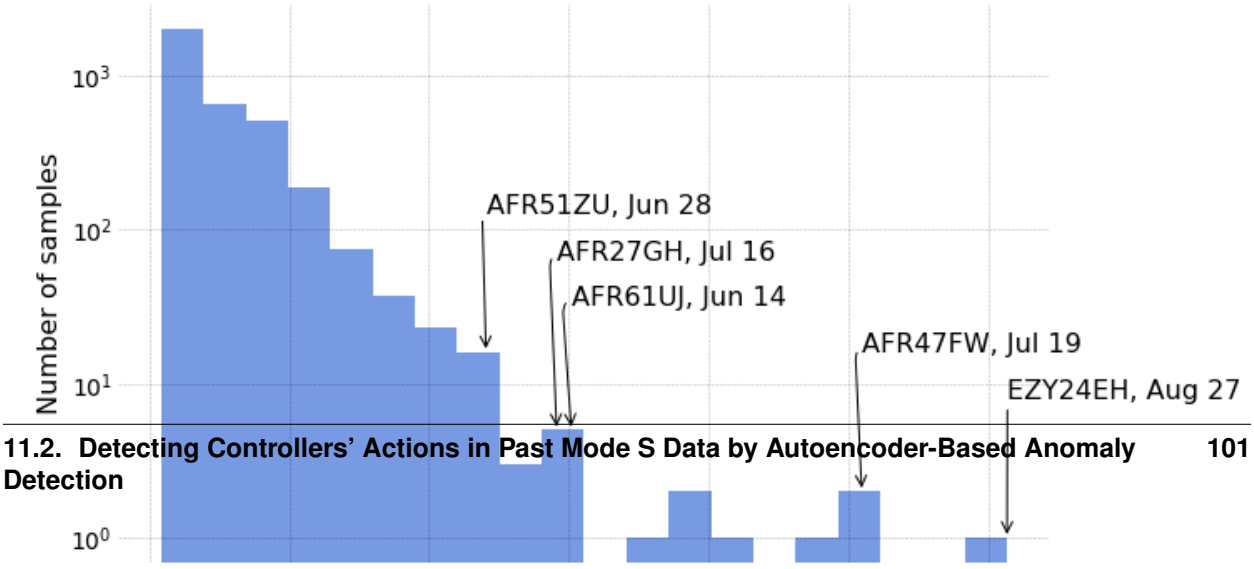
(continues on next page)

(continued from previous page)

```
            fontsize=16,
            horizontalalignment="left",
            arrowprops=dict(
                arrowstyle="->", connectionstyle="arc,angleA=180,armA=70,rad=10"
            ),
        )

    ax1.set_xlabel("Number of epochs", labelpad=10)
    ax1.set_ylabel("Loss evolution")
    ax2.set_ylabel("Number of samples")
    ax3.set_ylabel("Number of samples")
    ax3.set_xlabel("Reconstruction error", labelpad=10)
```


Loss evolution
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
250
500
750
1000
1250
1500
1750
2000
Number of epochs


Number of samples
2000
1750
1500
1250
1000
750
500
250
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0


Number of samples
10^3
10^2
10^1
10^0
AFR51ZU, Jun 28
AFR27GH, Jul 16
AFR61UJ, Jun 14
AFR47FW, Jul 19
EZY24EH, Aug 27

11.3 Quantitative Assessments of Runway Excursion Precursors using Mode S data

Xavier Olive, Pierre Bieber

This notebook comes with the paper published at ICRAT 2018.
Details are presented in the paper. The following code is provided for reproducibility concerns.

11.3.1 Data preparation

The dataset consists of one month of trajectories around Toulouse airport. The first step is to isolate trajectories landing. The following code is a suggestion to select such trajectories.

```
from tqdm.autonotebook import tqdm
from typing import List
from traffic.core import Traffic, Flight

february = Traffic.from_file("data/lfbo_february.pkl.gz")

cumul: List[Flight] = []

# The first step is to assign a specific id to all trajectories
for item in tqdm(february.assign_id()):

    flight = (
        item
        # remove samples with no positional data
        .query("latitude == latitude")
        # default median filters
        .filter()
    )

    # remove trajectories with not enough samples
    if len(item) < 100:
        continue

    # Keep flights landing at LFBO
    guess = flight.guess_landing_airport()
    if guess.airport.icao != "LFBO" or guess.distance > 6000:
        continue

    cumul.append(flight)

t_landing = Traffic.from_flights(cumul)

# backup
# t_landing.to_pickle("data/february_landing.pkl")
```

Since we want to focus on aircraft landing on runway 32, we need to apply a bit more filtering.

The library offers a tool trying to guess the runway on which an aircraft lands (only on major airports). The heuristics must be perfectible: warnings are raised when the assigned runway looks dodgy. We remove by hand trajectories with a raised warning (this could be automated in a better way...)

```
cumul: List[Flight] = []

for item in tqdm(t_landing):

    if item.icao24 in ["38413a", "380efa", "3801da", "398588"]:
        # aircraft with buggy data
        continue
    flight = item.query(
        # vertical_rate being NaN is often a sign of buggy data
        "~onground and vertical_rate == vertical_rate"
    )

    # extra sanity checks to take away invalid trajectories
    if len(flight) < 30 or flight.data.vertical_rate.mean() > -2.5:
        continue

    # only keep trajectories landing on runway 32
    runway = flight.guess_landing_runway()
    if runway.name.startswith("32"):
        cumul.append(
            flight.between(runway.point.start, runway.point.stop)
        )

# We remove the following id because the runway detection seems to have failed
t_32 = Traffic.from_flights(
    f for f in final if f.flight_id !='AIB103_1754'
)

# backup
# t_32.to_pickle("data/february_landing_32.pkl")

t_32
```

```
WARNING:root:(AF118VA_2073) Candidate runway 14L is not consistent with average track␣
↪323.0394445508367.
WARNING:root:(AIB103_1754) Candidate runway 32L is not consistent with average track␣
↪142.5683970215894.
WARNING:root:(AIB1776_555) Candidate runway 14L is not consistent with average track␣
↪180.0.
[...]
```

We end up with a dataset of 1361 trajectories landing at Toulouse airport in February 2017 on QFU32.

However, trimming trajectories to final approaches requires a bit more work. We use here navigational beacons in the official procedures. Only main navaids are provided by the data source embedded in the library: we add the coordinates manually and automate some plotting for following figures.

```
from traffic.core.mixins import PointMixin


class Point(PointMixin):
    """This mixin provides the interface to plot the elements on maps."""

    def __init__(self, lat, lon, name):
        self.latitude = lat
        self.longitude = lon
        self.name = name
```

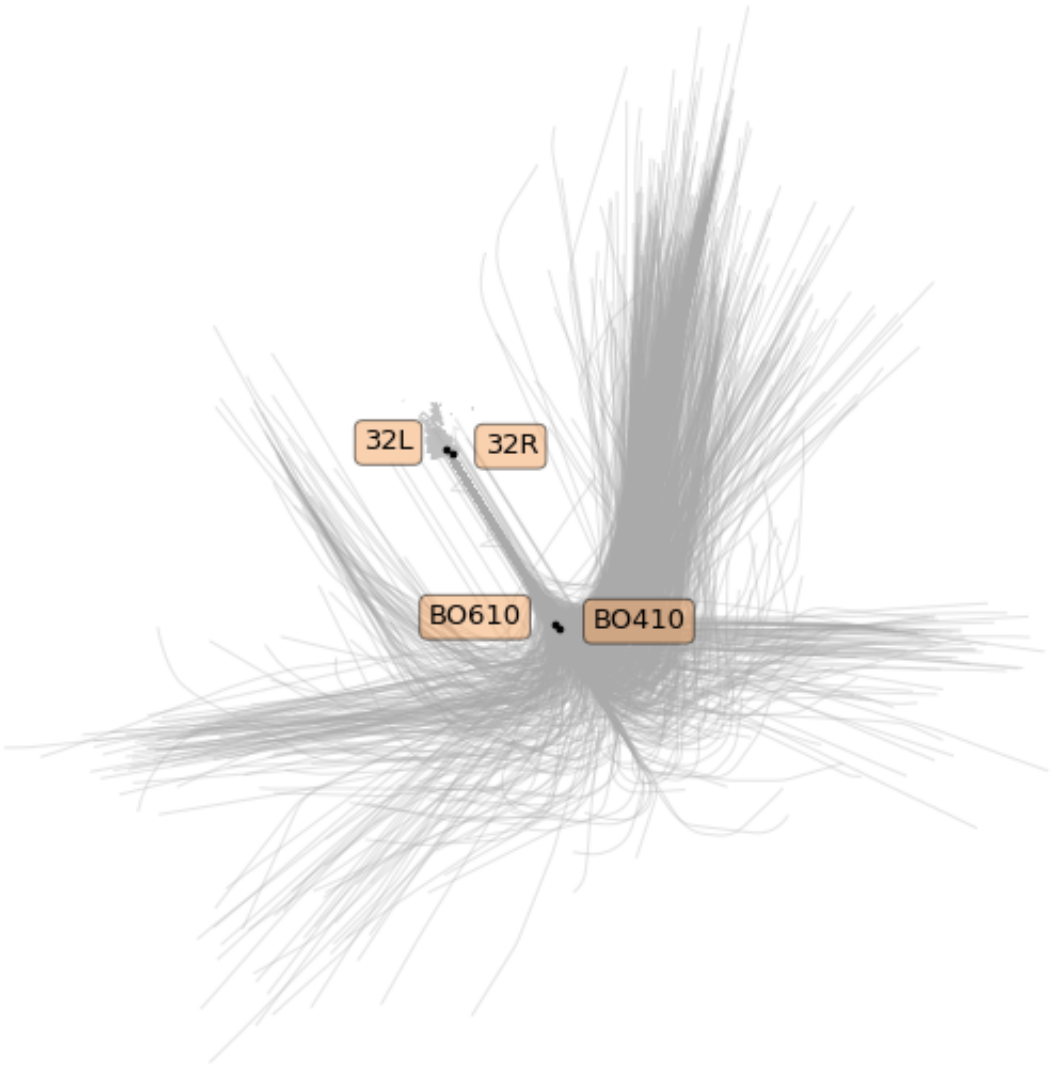
(continues on next page)

(continued from previous page)

```
# Coordinates for key positions for final approach in LFBO
procedure_points = {
    "BO310": Point(lat=43.787917, lon=1.200389, name="BO310"),
    "BO410": Point(lat=43.465222, lon=1.536195, name="BO410"),
    "BO510": Point(lat=43.794389, lon=1.188889, name="BO510"),
    "BO610": Point(lat=43.468472, lon=1.528195, name="BO610"),
    "14L": Point(lat=43.6374315, lon=1.3575536, name="14L"),
    "14R": Point(lat=43.6446126, lon=1.3454186, name="14R"),
    "32L": Point(lat=43.6185805, lon=1.3725227, name="32L"),
    "32R": Point(lat=43.6156582, lon=1.3802184, name="32R"),
}


def params(point_id):
    left_side = point_id in ["BO610", "32L"]
    return dict(
        shift=dict(units="dots", x=-15 if left_side else 15),
        text_kw=dict(
            s=point_id,
            horizontalalignment="right" if left_side else "left",
            bbox=dict(facecolor="sandybrown", alpha=0.5, boxstyle="round"),
        ),
    )


def plot_points(ax):
    for point_id in ["BO610", "BO410", "32L", "32R"]:
        value = procedure_points[point_id]
        value.plot(ax, s=7, zorder=2, **(params(point_id)))
```

With the following map, we can position trajectories landing on QFU32 with respect to the above mentioned navigational beacons.

```
from traffic.drawing import EuroPP
from traffic.data import airports

with plt.style.context('traffic'):
    fig, ax = plt.subplots(
        subplot_kw=dict(projection=EuroPP())
    )

    airports['LFBO'].plot(ax)
    plot_points(ax)

    t_32.plot(ax, alpha=.3, zorder=-2)

    ax.outline_patch.set_visible(False)
    ax.background_patch.set_visible(False)
```



First visual check: so far so good! Some trajectories seem to have their positions quite drifted.

In order to select the final approach, we trim the trajectories between their closest timestamp near one of the BOx10 beacon, and near one of the two runway thresholds. We filter trajectories passing too far away from this beacons in order to get rid of trajectories seeming to have a wrong estimation of their position (probable drifting of the inertial navigation system). We eliminate a few more trajectories but consider it still acceptable for a statistical analysis.

```
from typing import Any, Dict, List

import pandas as pd
from traffic.core import geodesy


cumul: List[Dict[str, Any]] = []

for flight in tqdm(t_32):
    bo = flight.closest_point([proc["BO610"], proc["BO410"]])
```

(continues on next page)

(continued from previous page)

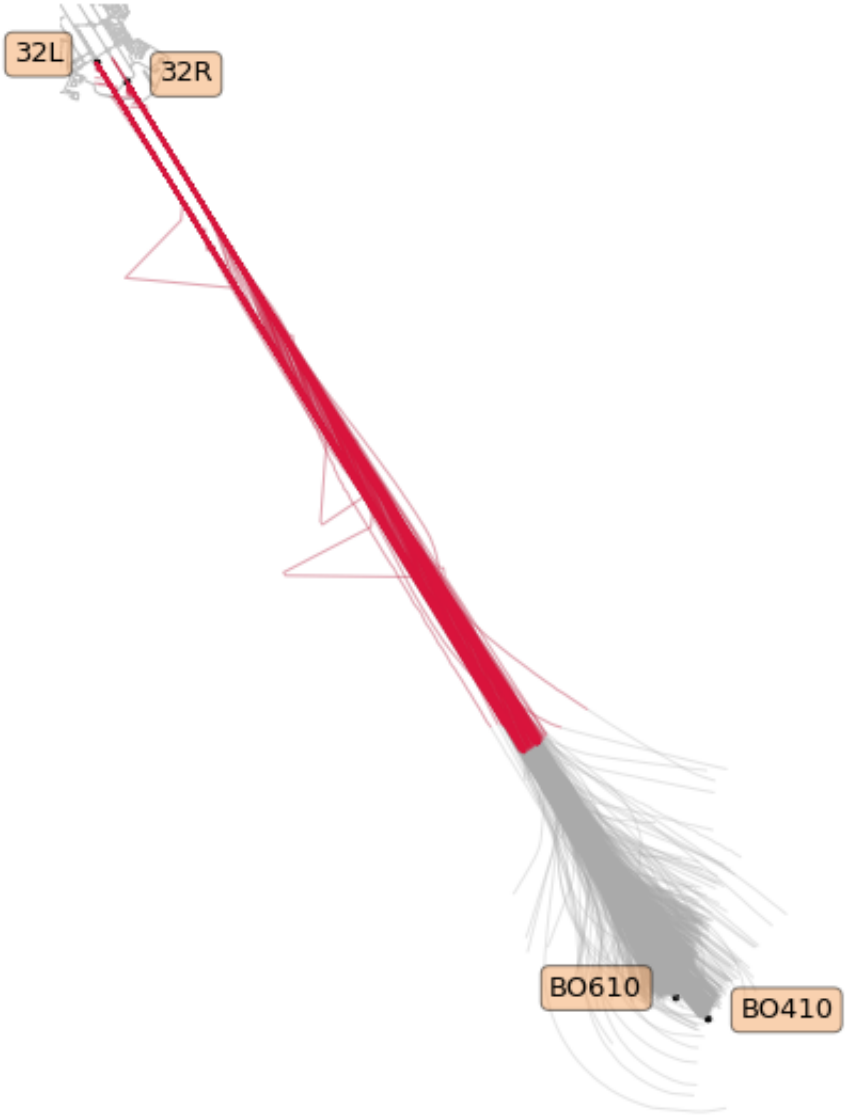
```
    rw = flight.closest_point([proc["32L"], proc["32R"]])
    cumul.append(
        dict(
            flight_id=flight.flight_id,
            # estimation of roll-out point (closest to BOx10)
            bo=bo.name,
            bo_ts=bo.point.timestamp,
            bo_d=bo.distance,
            # time at runway threshold
            rw=rw.name,
            rw_ts=rw.point.timestamp,
            rw_d=rw.distance,
        )
    )

analysis = pd.DataFrame.from_records(cumul)

# backup
# analysis.to_pickle("analysis.pkl")

t_final = Traffic.from_flights(
    t_32[line.flight_id]
    # trim the trajectory to final approach
    .between(line.bo_ts, line.rw_ts)
    # add one column for distance to the proper runway threshold
    .assign(
        distance=lambda df: geodesy.distance(
            df.latitude.values,
            df.longitude.values,
            df.shape[0] * [procedure_points[line.rw].latitude],
            df.shape[0] * [procedure_points[line.rw].longitude],
        )
    )
    # this last filtering removes flight with data which is erroneous or
    # irrelevant to our current case study.
    for _, line in analysis.query("rw_d < 400 and bo_d < 5000").iterrows()
)

# backup
# t_final.to_pickle("data/february_landing_32_final.pkl")

t_final
```

```
%matplotlib inline
import matplotlib.pyplot as plt

from traffic.drawing import EuroPP, rivers
from traffic.data import airports

with plt.style.context('traffic'):
    fig, ax = plt.subplots(
        subplot_kw=dict(projection=EuroPP())
    )

    airports['LFBO'].plot(ax)
    plot_points(ax)
```

(continues on next page)

(continued from previous page)

```
    t_final.plot(ax, alpha=.3,)
    t_final.query('distance < 8 * 1852').plot(ax, color='crimson', alpha=.3,)

    ax.outline_patch.set_visible(False)
    ax.background_patch.set_visible(False)
```



Visual check: in the paper we consider the final 8 nautical miles (in red).

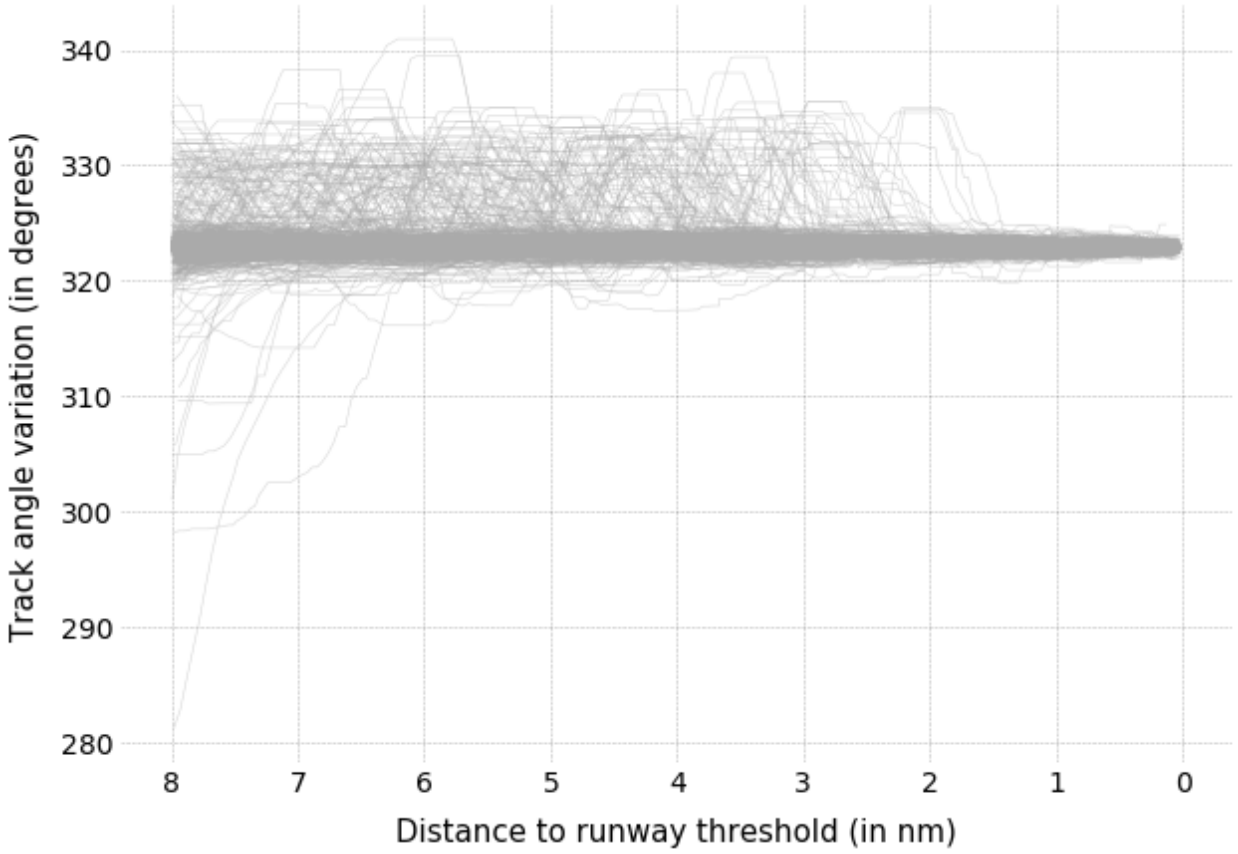
For the following, the idea is to consider all track angle signals and to analyse their modes of variation. Here is the full dataset plotted.

```
with plt.style.context('traffic'):
    fig, ax = plt.subplots(figsize=(10, 7))
    ax.invert_xaxis()
```

(continues on next page)

(continued from previous page)

```
    ax.set_xlabel("Distance to runway threshold (in nm)", labelpad=10, fontsize=15)
    ax.set_ylabel("Track angle variation (in degrees)", labelpad=10, fontsize=15)

    for flight in t_final.query('distance < 8*1852'):
        ax.plot(
            flight.data.distance/1852,
            flight.data.heading,
            color='#aaaaaa',
            alpha=.5,
            linewidth=.5
        )
```



11.3.2 Functional Principal Component Analysis

We resample all trajectories to 50 points and fit a (functional) PCA on the dataset. The following plot displays some modes of variation around the average signal.

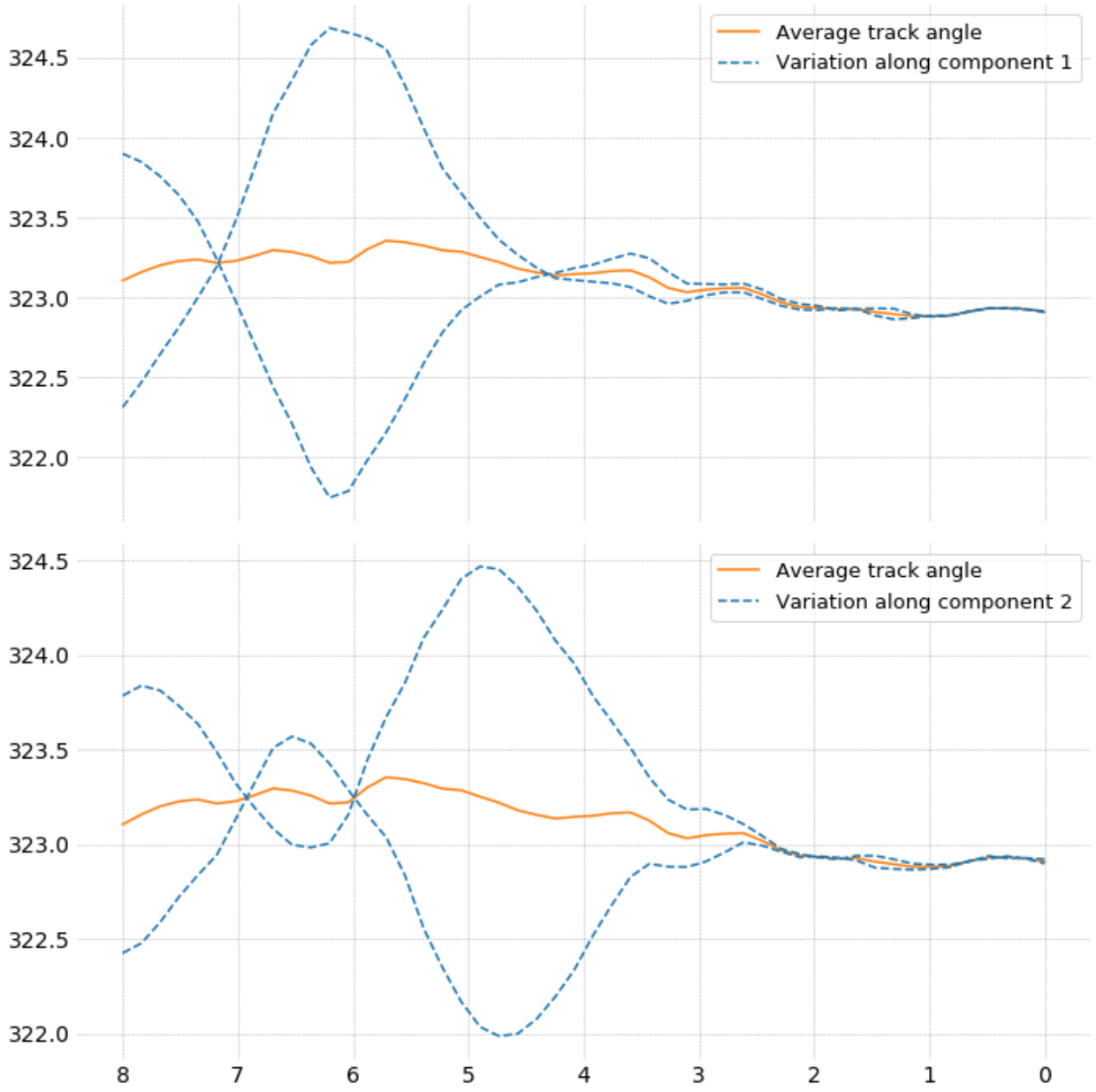
```
import numpy as np
from sklearn.decomposition import PCA

# Prepare a dataset of track angles on final approach
X = np.vstack(
    [
```

(continues on next page)

(continued from previous page)

```
        # for this demonstration we take 50 samples on final approach
        flight.resample(50).data.track
        for flight in t_final.query("distance < 8*1852")
    ]
)

# keep track of the identifier for each trajectory
flight_ids = list(flight.flight_id for flight in t_final)

pca = PCA()
X_t = pca.fit_transform(X)
```

```
with plt.style.context("traffic"):

    fig, ax = plt.subplots(2, 1, figsize=(10, 10), sharex=True)
    xlim = np.linspace(8, 0, 50)

    m_theme = dict(linestyle='solid', color='#ff7f0e')
    v_theme = dict(linestyle="--", color="#1f77b4")

    for i, a in enumerate(ax):

        var_i = np.sqrt(pca.explained_variance_[i + 1])
        theta_i = pca.components_[i + 1]

        a.plot(xlim, pca.mean_, **m_theme,
               label="Average track angle")
        a.plot(xlim, pca.mean_ + var_i * theta_i, **v_theme,
               label=f"Variation along component {i+1}")
        a.plot(xlim, pca.mean_ - var_i * theta_i, **v_theme)

        a.legend()

    a.invert_xaxis()
    fig.tight_layout()
```



One of the main ideas of the paper was to specifically select trajectories with strong components along these modes of variation. It appears they follow a specific pattern of late runway changes.

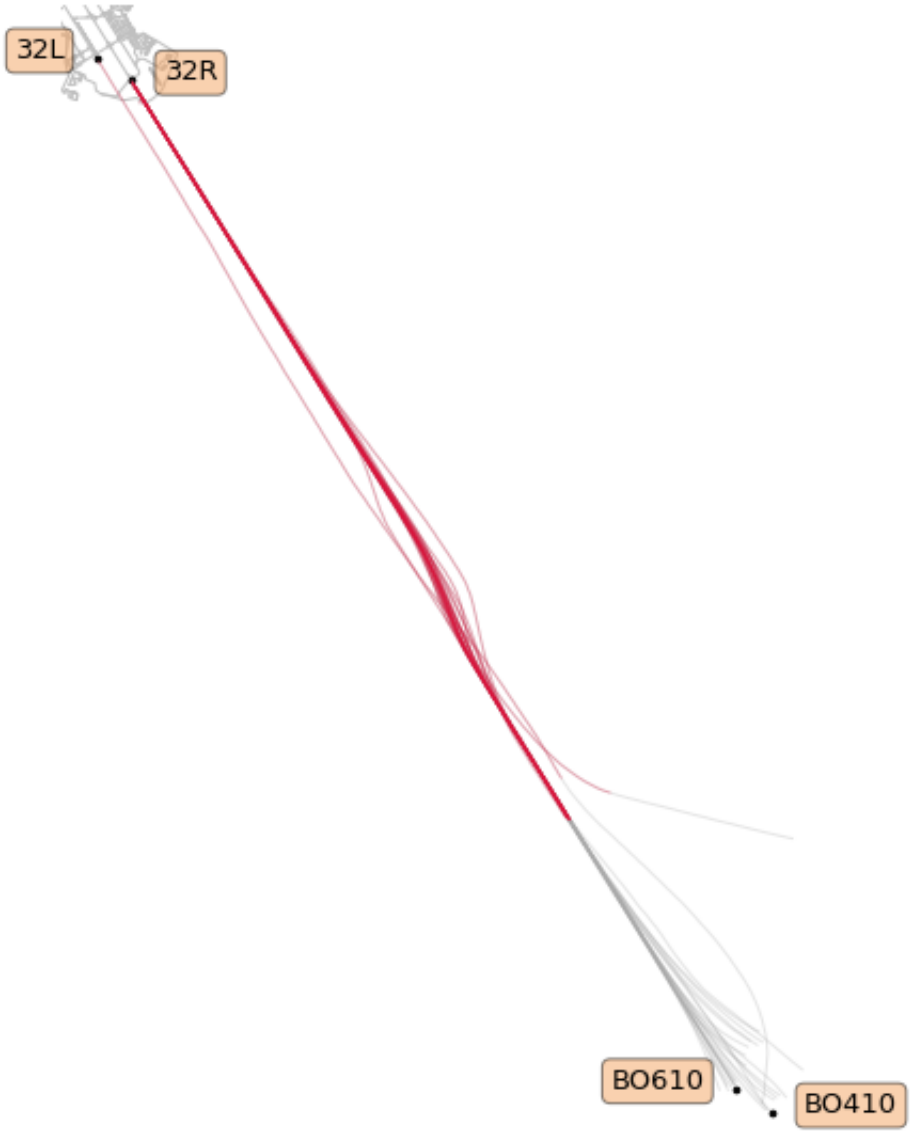
```
second_component = np.abs(X_t[:, 1])
selected_flights = Traffic.from_flights(
    t_final[flight_id]
    for flight_id, component in zip(flight_ids, second_component)
    if component > np.percentile(second_component, 98)
)
```

```
from traffic.drawing import EuroPP, rivers
from traffic.data import airports

with plt.style.context('traffic'):
```

(continues on next page)

(continued from previous page)

```
    fig, ax = plt.subplots(
        subplot_kw=dict(projection=EuroPP())
    )

    airports['LFBO'].plot(ax)
    plot_points(ax)

    selected_flights.plot(ax, color='#aaaaaa', alpha=.3)
    selected_flights.query('distance < 8 * 1852').plot(
        ax, color='crimson', alpha=.3
    )

    ax.outline_patch.set_visible(False)
    ax.background_patch.set_visible(False)
```



Index

D

DataFrameMixin (*class in traffic.core.mixins*), 55

F

from_file() (*traffic.core.mixins.DataFrameMixin class method*), 56

T

to_csv() (*traffic.core.mixins.DataFrameMixin method*), 56
to_hdf() (*traffic.core.mixins.DataFrameMixin method*), 56
to_json() (*traffic.core.mixins.DataFrameMixin method*), 56
to_parquet() (*traffic.core.mixins.DataFrameMixin method*), 56
to_pickle() (*traffic.core.mixins.DataFrameMixin method*), 56