# openSkies - Integration of Aviation Data into the R Ecosystem

*by Rafael Ayala, Daniel Ayala, Lara Sellés Vidal, and David Ruiz*

**Abstract** Aviation data has become increasingly more accessible to the public thanks to the adoption of technologies such as Automatic Dependent Surveillance-Broadcast (ADS-B) and Mode S, which provide aircraft information over publicly accessible radio channels. Furthermore, the OpenSky Network provides multiple public resources to access such air traffic data from a large network of ADS-B receivers. Here, we present **openSkies**, the first R package for processing public air traffic data. The package provides an interface to the OpenSky Network resources, standardized data structures to represent the different entities involved in air traffic data, and functionalities to analyze and visualize such data. Furthermore, the portability of the implemented data structures makes **openSkies** easily reusable by other packages, therefore laying the foundation of aviation data engineering in R.

## Introduction

The ADS-B aircraft surveillance technology enables the tracking of aircraft through the reception by ground stations of a signal broadcast by the aircraft itself (Rekkas and Rees, 2008). Such a system does not require any external input and is instead fully dependent on data obtained by the aircraft navigation system. Additionally, unlike other radar-based systems used for aircraft surveillance, no interrogation signals from the ground stations are required. This is in opposition to, for example, the air traffic control radar beacon system operating in Mode S (Selective), where specific aircraft send a response message only after reception of another signal (the interrogation) emitted by a ground antenna. In the ADS-B, messages broadcast by an aircraft can include precise location information, typically determined by a Global Navigation Satellite System, such as GPS (Global Positioning System). Alternatively, it is possible to estimate the position of the aircraft from the timestamps of reception of the broadcast signal at different stations through multilateration (Kaune et al., 2012).

The OpenSky Network initiative provides free access to high-quality air traffic data derived from a network of thousands of ground stations that receive ADS-B messages, and, more recently, FLARM ("Flight Alarm") messages as well (Schäfer et al., 2014). Both ADS-B and FLARM messages are essentially radio signals broadcasted by an aircraft that can be received by other aircraft or ground stations. They carry positional information and other data about the broadcasting aircraft (with the main difference being that the FLARM system is primarily aimed at avoiding collisions. Therefore, it also transmits a prediction of the trajectory of the aircraft in the next 20 seconds). More than 25 trillion ADS-B messages and over 3 billion FLARM messages have been collected by the OpenSky Network since 2013, corresponding to over 400,000 aircraft in 190 countries. Such data provides the basis to develop novel algorithms for the determination of aircraft position and the analysis of air traffic data. Access to the available data is provided through two main tools: a live API (which also supports the retrieval of historical data, although with some limitations) and an Impala shell (Apache Software Foundation, 2020) that provides fast access to the full historical dataset, to which free access can be obtained for research and other non-profit purposes. However, these resources by themselves do not provide the data in appropriate data structures that make them amenable to advanced analyses available in R and other R packages.

We present **openSkies**, an R package that aims to provide a well-established and documented basis for the analysis of air traffic in R and to facilitate the future development of related algorithms. To that extent, access to the OpenSky Network live API and Impala shell is provided, as well as a decoder of raw ADS-B messages. A set of data structures that represent relevant concepts (such as aircraft, flights, or airports) is also implemented and used to store the data retrieved from the OpenSky Network. Relevant methods to process and analyze the corresponding data are associated with each data structure.

The package also includes functions to perform statistical analysis and to visualize the retrieved data, enabling, for example, the clustering of trajectories and the representation of aircraft and flight paths in a given airspace. Additional functionalities can be easily built on the already implemented classes and functions, either in future versions of **openSkies** or in other packages developed by the R community. In the following sections, we describe the different features of the package and demonstrate how each of them can be accessed and applied to real data.

Firstly, we describe the implemented data structures required to model air traffic (Data structures). We then show the functions that allow instantiating said classes with data obtained from the OpenSky Network (Information retrieval). Next, the currently available tools to visualize (Visualization of air

traffic) and cluster (Clustering of aircraft trajectories) are presented. Finally, a tool to decode raw ADS-B messages is presented (Decoding ADS-B messages). Examples with real data are provided throughout the different sections to illustrate the features of the package.

## Data structures

In order to establish standardized data structures that could serve as the basis for future developments in air traffic analysis in R, a set of R6 classes representing the frequently involved entities was implemented. R6 was chosen as the class system due to the possibility to establish formal definitions of reference classes, as well as the smaller memory footprint and faster speeds of object instantiation, field access, and field setting compared to base R's Reference Classes system.

Additionally, R6 classes are portable. Therefore, the classes defined in **openSkies** can be used by other packages.

In its current version (1.1.3), **openSkies** defines classes for the following entities: aircraft, airports, flight instances, flight routes, single aircraft state vectors, and series of aircraft state vectors (Figure 1).

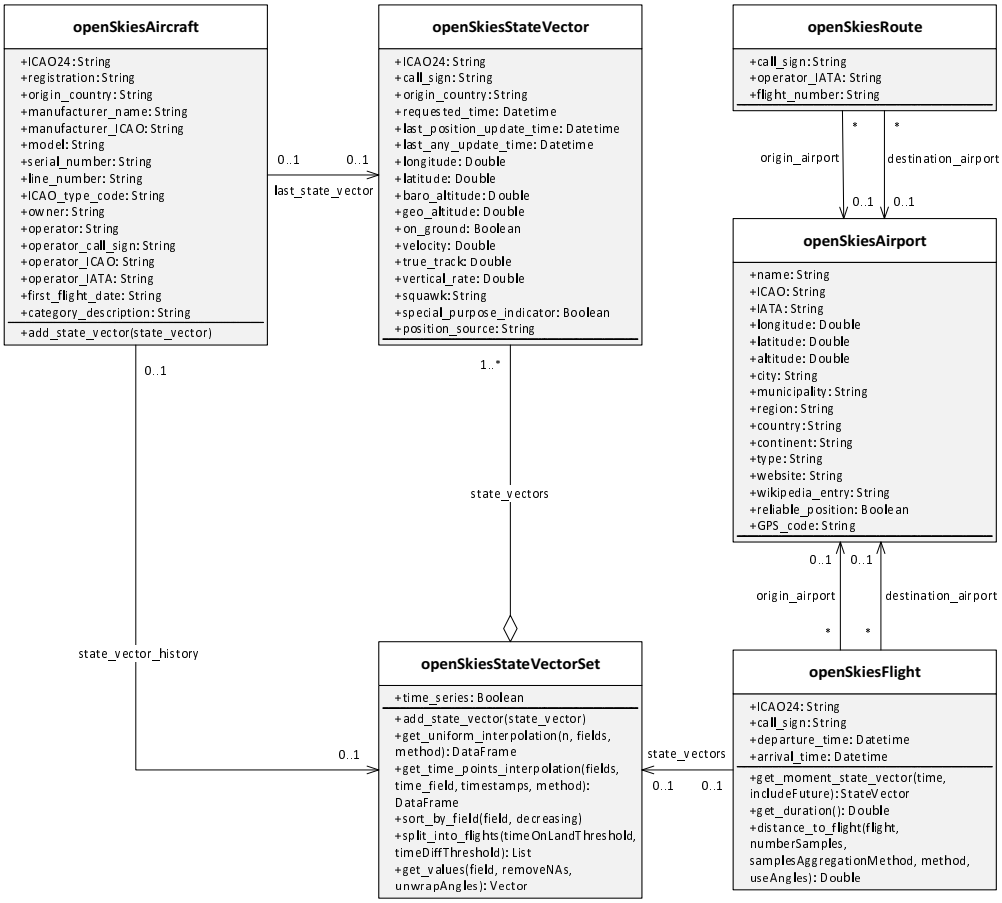


**Figure 1:** R6 classes implemented in **openSkies**.

Class `"openSkiesStateVector"` represents the state vector of an aircraft at a given time. A state vector comprises multiple pieces of information about the aircraft and its status, including positional information (latitude, longitude, altitude, whether the aircraft is on the ground) and trajectory information (velocity, track angle, vertical rate). Sets of `"openSkiesStateVector"` are represented with class `"openSkiesStateVectorSet"`, which contains two fields: a list of objects of class `"openSkiesStateVector"` and a logical, indicating if the set of state vectors corresponds to a time series of the same aircraft (field `time_series`). Class `"openSkiesStateVectorSet"` contains methods to add more objects of class `"openSkiesStateVector"` to the contained list of state vectors, get interpolations of any of the numerical fields of the state vectors (uniformly spaced or at specific time points), sort the list of state vectors by a specific field, get the values of a given field for all the state vectors, and split the set of state vectors into individual flight instances. The latter operation is performed by firstly grouping the state vectors by aircraft and then splitting them into flights based on two conditions:

either the aircraft staying on the ground for a given amount of time or the aircraft not sending any status update for a given period (with default values of 300 seconds and 1800 seconds).

A flight instance is defined as a single flight performed by an aircraft, including typically (but not necessarily) take-off, air transit, and landing. Flights are represented class `"openSkiesFlight"`, which contain fields for the aircraft that performed the flight, the departure and arrival times, the airports of origin and destination (as objects of class `"openSkiesAirport"`), and a set of state vectors describing the aircraft during the flight, which is an object of class `"openSkiesStateVectorSet"` with field `time_series=TRUE`. The class contains methods to get the state vector of the aircraft for a specified time, calculate the duration of the flight, and calculate the distance (based on features extracted from the trajectories) to another object of class `"openSkiesFlight"`.

Airports are represented by class `"openSkiesAirport"`, with fields describing, among others, the airport codes, its position (latitude, longitude, and altitude), and its location in terms of administrative divisions of different levels.

Flight routes, which differ from flight instances in that they are designated paths followed regularly by aircraft to go from one airport to another one, are represented by class `"openSkiesRoute"`. The class contains fields for the call sign and flight number associated with the route, its operator, and its airports of origin and destination (objects of class `"openSkiesAirport"`).

Finally, aircraft are represented by class `"openSkiesAircraft"`, which contains fields describing the aircraft model, registration, manufacturer, owner, and operator. Objects of this class can also contain a field with an object of class `"openSkiesStateVectorSet"` representing the history of known state vectors of the aircraft in a given period.

## Information retrieval

While it is possible to manually instantiate any of the classes previously described by providing values for the required fields with information obtained from any source, a more convenient way is to instantiate them from information automatically retrieved from the OpenSky Network. To that extent, a series of functions that retrieve data from the different resources offered by the OpenSky Network is implemented. State vectors can be accessed with two functions. Firstly, `getSingleTimeStateVectors` retrieves all the state vectors received at a specified time as an `"openSkiesStateVectorSet"` object with field `time_series=FALSE`. In the default behavior, state vectors received from any aircraft at any location are returned. However, the results can be filtered by specifying one or more aircraft and the boundaries of an area of interest. On the other hand, `getAircraftStateVectorsSeries` retrieves a time series of state vectors for a given aircraft with the specified time resolution, in the form of an `"openSkiesStateVectorSet"` object with field `time_series=TRUE`. Both functions can retrieve data from the OpenSky Network through two resources integrated seamlessly. By default, the live API is employed. The live API does not require registration but imposes limitations (larger for anonymous users) on the retrieval of historical data and the time resolution with which data can be accessed. Additionally, large queries can take considerable amounts of time, especially when retrieving time series of state vectors. Therefore, the functions can also access the database through the Impala shell, which requires registration but does not impose such limits and performs queries significantly faster. Authentication for these two functions can be performed by providing login details through the `username` and `password` arguments, and access through the Impala shell can be enabled by authorized users simply by setting argument `useImpalaShell=TRUE`. New users can register through the OpenSky Network website, and Impala shell access should be directly requested to the OpenSky Network administrators.

In both cases, the resulting objects of class `"openSkiesStateVectorSet"` can be used to instantiate objects of class `"openSkiesFlight"` by calling the `split_into_flights` method. Alternatively, functions to retrieve flight instances from the OpenSky Network are available, based on the aircraft that performed them (`getAircraftFlights`) or the airport of origin (`getAirportDepartures`) or destination (`getAirportArrivals`). In all cases, it is possible to include the time series of state vectors corresponding to each flight by setting `includeStateVectors=TRUE` and airport metadata through `includeAirportsMetadata=TRUE`.

Airport and aircraft metadata can also be retrieved independently with the `getAirportMetadata` and `getAircraftMetadata` functions by providing the ICAO 4-letter code of the airport of interest or the ICAO 24 bit address of the target aircraft, respectively.

Finally, information about routes can be accessed by providing the call sign of the route to the `getRouteMetadata` function, which includes the possibility to retrieve the metadata of the associated airports of origin and destination if `includeAirportsMetadata=TRUE`.

In the following example, we aim to demonstrate some of the most common methods for retrieving air traffic data and analyzing it.

```
library(openSkies)

## In the following example, we retrieve information about all flights
## that departed from Frankfurt Airport the 29th of January 2018 after 12 pm.
## It should be noted that such large requests can take a long time unless
## performed with the Impala shell
## It should also be noted that, in this and the following examples, the
## values for the username and password arguments should be substituted with
## personal credentials registered at the OpenSky Network
flights <- getAirportArrivals(airport="EDDF", startTime="2018-01-29 12:00:00",
                              endTime="2018-01-29 24:00:00", timeZone="Europe/Berlin",
                              includeStateVectors = TRUE, timeResolution = 60,
                              useImpalaShell = TRUE, username = "user",
                              password = "password")
## We can then easily check the amount of flights
length(flights) # 316 flights

## Trajectories of the 316 flights can be obtained by retrieving
## the set of state vectors of each flight
trajectories_frankfurt <- lapply(flights, function(f) f$state_vectors)

## It is also possible to retrieve all state vectors received from a
## given aircraft in a given period of time. In the following example, we
## obtain all the state vectors received for aircraft with ICAO24 code
## 403003 between 12 PM of the 8th of October, 2020 and 6 PM of the
## 9th of October, 2020
stateVectors <- getAircraftStateVectorsSeries("403003", startTime = "2020-10-08 12:00:00",
                                              endTime = "2020-10-09 18:00:00",
                                              timeZone="Europe/London",
                                              timeResolution = 60,
                                              useImpalaShell = TRUE,
                                              username = "user",
                                              password = "password")

## The ensemble of state vectors can then be split into individual
## flight instances, revealing that the aircraft performed 6 flights
## in the analyzed period
flights <- stateVectors$split_into_flights()
length(flights) # 6 flights

## Let us get some additional information about the flights performed by
## this aircraft. For example, the maximum speed that it reached in km/h
maxSpeed <- max(stateVectors$get_values("velocity", removeNAs = TRUE))/1000*3600

## The maximum speed that it reached is just above 210 km/h. This is well below
## the speed typically used by commercial passenger jets, usually around 800 km/h
## Investigation of the aircraft model confirms that it is indeed a small CESSNA 152
aircraftData <- getAircraftMetadata("403003")
aircraftData$model
```

## Visualization of air traffic

Two main methods for visualization of air traffic are currently available in **openSkies**: plotting of flights and plotting of aircraft at a given time.

The first method aims to represent flights performed by one or more aircraft over a given period of time. If only a single flight is to be plotted, the `plotRoute` function can be used, which receives as its main input an object of class `"openSkiesStateVectorSet"` with `time_series=TRUE`. Alternatively, `plotRoutes` can be used for plotting multiple flights, receiving in this case as input a list of `"openSkiesStateVectorSet"` objects with `time_series=TRUE`. The color of the routes can be manually set by providing a vector of color names to `pathColors` and setting `literalColors=TRUE`. It is also possible to color the routes according to the value of any property by providing such values as a factor to `pathColors` and setting `literalColors=FALSE`.

The second visualization method (`plotPlanes`) enables the plotting of all aircraft flying over a defined area at a particular time. An `"openSkiesStateVectorSet"` object with each state vector representing the position of an aircraft at the desired time should be provided as input, most frequently obtained through `getSingleTimeStateVectors`.

All plotting functions allow plotting onto any object of class ggmap from the **ggmap** package (Kahle et al., 2019). If no ggmap object is provided, one will be created internally, with boundaries large enough to contain all the positions in the provided `"openSkiesStateVectorSet"` objects, plus additional space determined by the `paddingFactor` argument.

The following examples demonstrate the application of visualization functions. First, the formerly retrieved trajectories for the CESSNA 152 are plotted with different colors:

```
## First, let us obtain the trajectories of the flights performed
## by the CESSNA 152
trajectories_CESNA152 <- lapply(flights, function(f) f$state_vectors)

## Then, we create a color palette
library(viridis)
colors <- magma(length(trajectories))

## Then, the trajectories are plotted
plotRoutes(trajectories, pathColors = colors, lineSize = 1.2,
           lineAlpha = 0.8, includeArrows = TRUE,
           paddingFactor = 0.05)
```

The resulting plot is shown in Figure 2. Next, we plot all the aircraft flying over Switzerland in a given instant:

```
## Firstly we retrieve the state vectors of all aircraft flying over Switzerland
## the 2nd of March, 2018 at 14.30 (London time)
vectors_switzerland <- getSingleTimeStateVectors(time="2018-03-02 14:30:00",
                                                 timeZone="Europe/London",
                                                 minLatitude=45.8389,
                                                 maxLatitude=47.8229,
                                                 minLongitude=5.9962,
                                                 maxLongitude=10.5226,
                                                 username="user",
                                                 password="password",
                                                 useImpalaShell = TRUE)

## Then, the aircraft are plotted
plotPlanes(vectors_switzerland)
```

The result is shown in Figure 3.

## Clustering of aircraft trajectories

As a means to analyze trajectories, **openSkies** provides the functionalities to perform clustering of trajectories. The first step is to obtain a matrix of features of the trajectories to be clustered. This can be achieved by providing a list of `"openSkiesStateVectorSet"` objects to the `getVectorSetListFeatures` function.

The `"openSkiesStateVectorSet"` objects in the list should each correspond to a single flight instance, and therefore it is advisable to first split the data into individual flights with method `split_into_flights`. It is possible to include track angles in the generated features matrix by setting `useAngles=TRUE`, which can be desirable if flights going along the same route but in opposite directions should be classified separately.

The computed features matrix (or, alternatively, the list of `"openSkiesStateVectorSet"` objects, in which case the features matrix will be calculated internally) can then be passed to the `clusterRoutes` function. A number of different clustering algorithms can be selected through the `method` argument. Possible methods are `dbscan`, `kmeans`, `hclust`, `fanny`, `clara`, and `agnes`.

The desired number of clusters can be specified through `numberClusters`, although this argument is ignored if the DBSCAN algorithm (Hahsler and Piekenbrock, 2019) is used since it does not require *a priori* determination of the number of clusters. Instead, the *eps* parameter controls the size of the

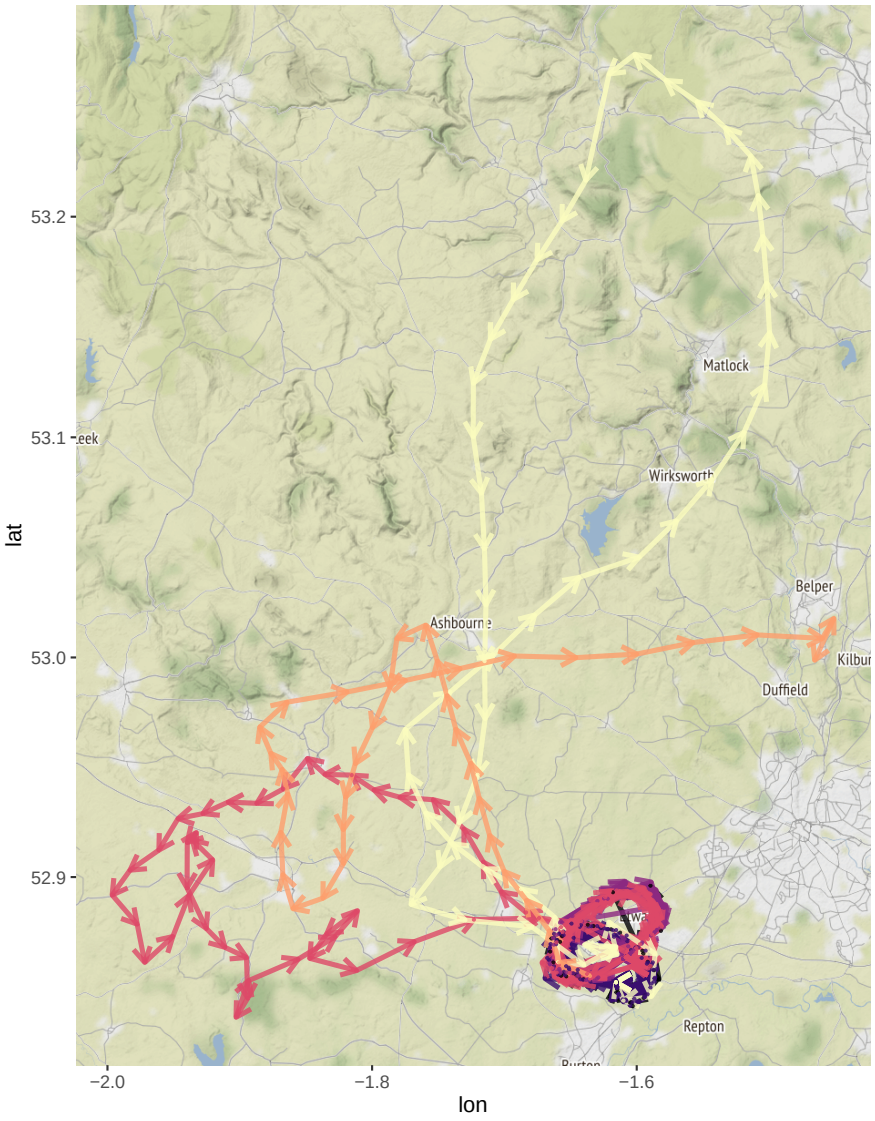


**Figure 2:** Visualization of flights performed by a CESSNA 152.

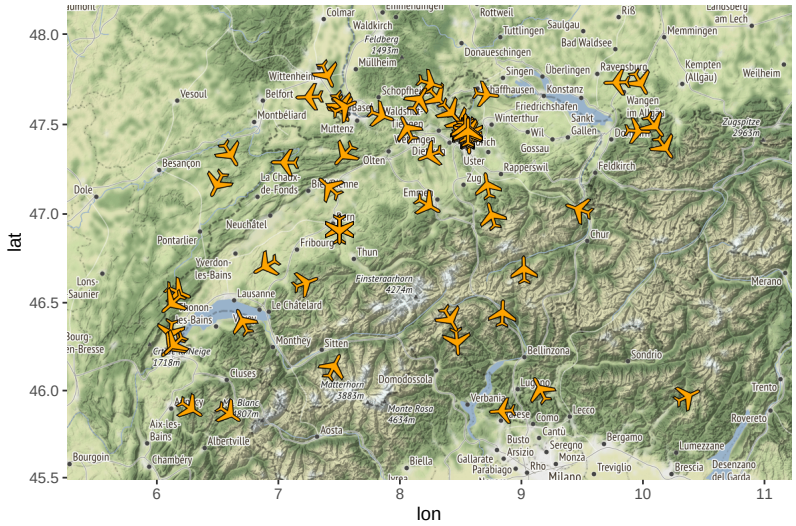


**Figure 3:** Visualization of aircraft flying over Switzerland.

epsilon neighborhood to be passed to the DBSCAN, which influences the number of found clusters (any two trajectories will be considered to belong to the same cluster if their calculated distance is below the threshold epsilon).

For all other methods (Maechler et al., 2019), if a number of clusters is not chosen, it will be internally estimated. The results of clustering can be visualized by providing the factor with the assigned clusters to the `pathColors` argument of `plotRoutes`. In the following example, we perform clustering of the previously obtained set of 316 flights departing from Frankfurt airport (Figure 4).

```
## As an example, let´s cluster the previously retrieved 316 flights departing
## from Frankfurt airport into 8 clusters with the k-means algorithm
clusters = clusterRoutes(trajectories_frankfurt, "kmeans", numberClusters = 8)

## The results can be visualized with plotRoutes
plotRoutes(trajectories_frankfurt, pathColors = clusters$cluster,
           literalColors = FALSE, paddingFactor = 0.1)

## The clusters differ in their broad area of destination. For example,
## clusters 4, 5 and 6 comprise mostly flights to North America, Japan and
## central Asia respectively. Due to their geographical proximity, it is expected
## that flights from cluster 5 are closer to flights from cluster 6 than
## to those assigned to cluster 4. Let us confirm this:
flights[clusters$cluster==5][[1]]$distance_to_flight(flights[clusters$cluster==6][[1]])
## Returns 123.3449
flights[clusters$cluster==5][[1]]$distance_to_flight(flights[clusters$cluster==4][[1]])
## Returns 348.4957

## The shorter distance between flights of clusters 5 and 6 is in accordance with
## the expectations.
```

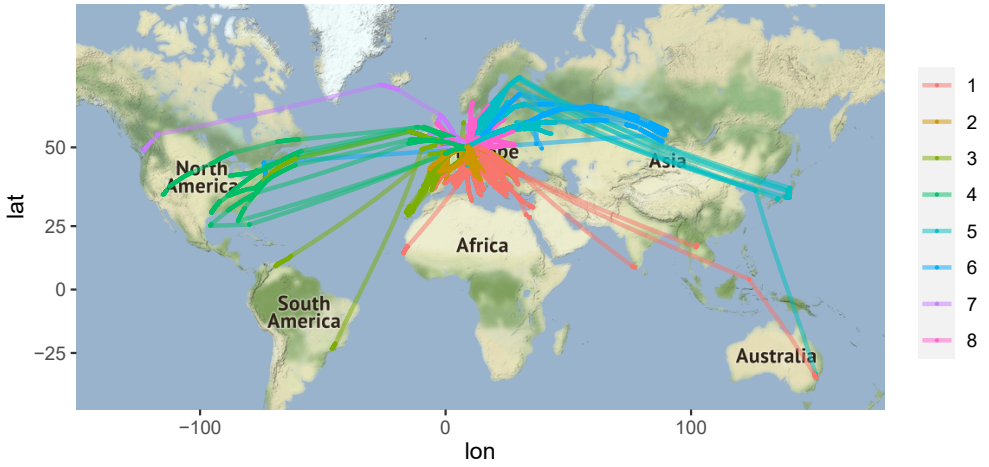


**Figure 4:** Clustering of flights departing from Frankfurt airport.

## Decoding ADS-B messages

Finally, **openSkies** also provides a decoder of raw ADS-B messages, implemented as an instance of R6 class `"adsbDecoder"` with the methods required to decode messages. This tool is useful for users willing to extract air traffic data from their own sources, such as private ADS-B receivers. Firstly, the messages, typically available in hexadecimal format, must be converted to binary format, which can be achieved with the `hexToBits` function. The binary messages can then be passed to the `decodeMessage` method, which decodes a single message. It should be noticed that, due to the Compact Position Reporting format with which the position of airborne aircraft is encoded, at least 2 consecutive positional ADS-B messages are required in order to unequivocally determine the position of the aircraft (referred to as *even* and *odd* frames).

Therefore, the decoder caches the most recently even and odd decoded message, to decode the next complementary positional message.

It is also possible to provide a list of messages in binary format to the `decodeMessages` method, which will directly provide positional information about the aircraft if the list contains two complementary even and odd positional messages. In the following example, we demonstrate both the sequential and batch decoding of ADS-B messages:

```
## Let use the following set of 2 ADS-B messages in hexadecimal format
## to illustrate the ADS-B decoder
message1 <- "8D40621D58C386435CC412692AD6"
message2 <- "8D40621D58C382D690C8AC2863A7"

## First, the messages must be converted to binary format:
message1_bin <- ADSBDecoder$hexToBits(message1)
message2_bin <- ADSBDecoder$hexToBits(message2)

## We can then obtain positional information by sequentially decoding
## both messages. Note that no positional information is obtained from the
## data extracted from message1, since this is the first message of the two
## that, as a pair, encode positional information.
message1_decoded <- ADSBDecoder$decodeMessage(message1_bin)
message2_decoded <- ADSBDecoder$decodeMessage(message2_bin)
message2_decoded$data$lat # The latitude is 52.2572 degrees North
message2_decoded$data$lon # The longitude is 3.919373 degrees East

## We can also provide both messages as a list to decode them simultaneously:
all_messages_decoded <- ADSBDecoder$decodeMessages(list(message1_bin, message2_bin))
```

## Future work

**openSkies** is in active development. Therefore, novel features are expected to be available through frequent updates.

Some of the features that will become available in the short and medium-term include:

- Smoothing and filtering of trajectories to remove spurious position outliers through both linear and non-linear methods.
- Automatic detection of the different segments of flight instances (such as take-off, initial climb, cruise altitude, descent, and landing).
- Detection of in-flight events and flight patterns. For example, delays in landing due to multiple reasons, which are often seen as small periodical trajectories near the destination airport.
- Application of detected in-flight events and flight patterns to obtain additional information about the involved aircraft, such as aircraft, type when this is not readily available.

## Summary

New technologies such as ADS-B surveillance and initiatives like the OpenSky Network have largely increased the amount of air traffic data that is publicly available, as well as its accessibility. However, the data provided by said resources must be parsed and formatted adequately in order to be analyzed in-depth and serve as the starting point to develop new algorithms useful for the field, such as novel multilateration methods.

However, while appropriate toolboxes have been implemented in other languages (Olive and Basora, 2019), such a standardized set of tools was not available in R until the development of **openSkies**, which aims to fill this void. One of the key features of our package is the automation and streamlining of the data engineering required in order to access air traffic data at a large scale and to enable data scientists to make meaningful analyses with it. The implemented portable data structures and functionalities provide a solid base to apply the vast range of functionalities provided by other R packages to air traffic data. The ecosystem created with future packages that interoperate with **openSkies**, together with reference datasets (Schäfer et al., 2020), should serve to standardize and expand the scope of air traffic data analysis in R.

## Acknowledgements

The development of this work is supported by the Spanish Ministry of Science and Innovation (grant code PID2019-105471RB-I00) and the Regional Government of Andalusia (grant code P18-RT-1060).

## Bibliography

Apache Software Foundation. *Apache Impala Guide*, 2020. URL http://impala.apache.org/docs/build/impala-3.4.pdf. [p551]

M. Hahsler and M. Piekenbrock. *dbscan: Density Based Clustering of Applications with Noise (DBSCAN) and Related Algorithms*, 2019. URL https://CRAN.R-project.org/package=dbscan. R package version 1.1-5. [p555]

D. Kahle, H. Wickham, and S. Jackson. *ggmap: Spatial Visualization with ggplot2*, 2019. URL https://CRAN.R-project.org/package=ggmap. R package version 3.0.0. [p555]

R. Kaune, C. Steffes, S. Rau, W. Konle, and J. Pagel. Wide area multilateration using ADS-B transponder signals. In *2012 15th International Conference on Information Fusion*, pages 727–734, 2012. URL https://ieeexplore.ieee.org/document/6289874. [p551]

M. Maechler, P. Rousseeuw, A. Struyf, and M. Hubert. *cluster: "Finding Groups in Data": Cluster Analysis Extended*, 2019. URL https://CRAN.R-project.org/package=cluster. R package version 2.1.0. [p557]

X. Olive and L. Basora. A Python Toolbox for Processing Air Traffic Data: A Use Case with Trajectory Clustering. In *Proceedings of the 7th OpenSky Workshop 2019*, pages 73–84, 2019. URL https://doi.org/10.29007/sf1f. [p558]

C. Rekkas and M. Rees. Towards ADS-B implementation in Europe. In *2008 Tyrrhenian International Workshop on Digital Communications - Enhanced Surveillance of Aircraft and Vehicles*, pages 1–4, 2008. URL https://doi.org/10.1109/TIWDC.2008.4649019. [p551]

M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic, and M. Wilhelm. Bringing up OpenSky: A large-scale ADS-B sensor network for research. In *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*, pages 83–94, 2014. URL https://doi.org/10.1109/IPSN.2014.6846743. [p551]

M. Schäfer, M. Strohmeier, M. Leonardi, and V. Lenders. LocaRDS: A Localization Reference Data Set, 2020. URL https://arxiv.org/abs/2012.00116. *arXiv* preprint arXiv:2012.00116. [p558]

*Rafael Ayala*
*Okinawa Institute of Science and Technology Graduate University*
*7542 Onna, Onna-Son, Kunigami, Okinawa 904-0411*
*Japan*
*ORCiD: 0000-0002-9332-4623*
rafael.ayala@oist.jp

*Daniel Ayala*
*University of Seville*
*ETSI Informática, Avda. de la Reina Mercedes, s/n, Sevilla E-41012*
*Spain*
*ORCiD: 0000-0003-2095-1009*
dayala1@us.es

*Lara Sellés Vidal*
*Imperial College London*
*Exhibition Road, London, SW7 2AZ*
*United Kingdom*
*ORCiD: 0000-0003-2537-6824*
lara.selles12@imperial.ac.uk

*David Ruiz*
*University of Seville*

*ETSI Informática, Avda. de la Reina Mercedes, s/n, Sevilla E-41012*
*Spain*
*ORCiD: 0000-0003-4460-5493*
druiz@us.es