S1 The TrackMatcher type tree

S1.1 The Julia language type ecosystem

Julia offers an ecosystem for custom types, which are classified either as abstract or concrete. A *concrete type* or *struct* stores the actual data in fields. An *abstract type* does not hold any information. Its purpose is to classify the concrete type.

Types are organised in a type tree, where the most general description of a group of abstract types is at the top. From these types the tree branches to more specific description in each new generation of abstract types until a concrete type is reached. Concrete types are the end points or leafs of each branch.

A classic example of a type tree is the number type tree given in Fig. S2. A `Number` can be either be a `Real` or `Complex` number and a `Real` can further be divided into `Integer`, floating point (`AbstractFloat`) numbers, `Rational` and irrational (`AbstractIrrational`) numbers. All the above types only classify a number going from a general description to more detailed descriptions. However, no information is stored in these types. The actual numbers are stored in concrete types, e.g. half, single, double precision or `BigFloat` for floating point numbers or *signed* and *unsigned* integer numbers with the special boolean type (`Bool`).

Parametric types are another feature of Julia's type ecosystem, which enforce a certain type on a struct's parameters. A simple example are coordinate pairs:

In the code in Fig. S1, the `abstract type` classifies the precision of the entries in each coordinate. The parameter of `AbstractCoordinate` is forced to be a subtype of `AbstractFloat`, thus, leaving only floats of various precisions as parameter. The `AbstractCoordinate` has two subtypes or children, `Coord2D` and `Coord3D` for 2D coordinates or 3D coordinate pairs considering altitude. Both need to have the same type of parameter `T`, so `T` does not need to be classified as `<: AbstractFloat` for the children.

Types and abstract types can be used to classify a coordinate. Consider the following coordinates:

1. `Coord2D{Float32}(0.0f0, 0.0f0)`
2. `Coord2D{Float64}(0.0, 0.0)`

Example 1: Parametric types of coordinate pairs

```
# Abstract type regulating floating point precision
abstract type AbstractCoordinate{T<:AbstractFloat} end

# Struct to save 2D coordinates
struct Coord2D{T} <: AbstractCoordinate
  lat::T
  lon::T
end

# Struct to save 3D coordinates
struct Coord3D{T} <: AbstractCoordinate
  lat::T
  lon::T
  alt::T
end
```

Figure S1. Code example of parametric types.

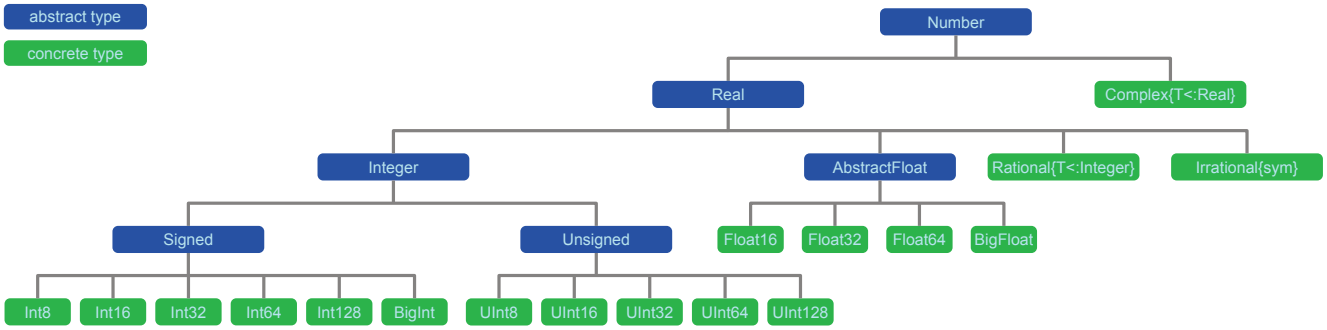


Figure S2. Number type tree in the Julia language as example of the type ecosystem.

3. `Coord3D{Float64}(0.0, 0.0, 0.0)`

All are subtypes of `AbstractCoordinate`, but only coordinate 2 and 3 are subtype of `AbstractCoordinate{Float64}`. It can be further narrowed down that only coordinate 2 is of type `Coord2D{Float64}`. This leaves the user various options to include or exclude certain coordinates in computations. Furthermore, with the given design of the coordinates, it is not possible to mix floating point precisions. Trying to save altitude as `Float64` and lat/lon as `Float32` would lead to an error as all parameters need to be of type `T` (either `Float32` or `Float64`).

S1.2 Data Structure

Data in TrackMatcher is organised in Julia structs to guarantee a unified format. The structure of the type tree is given in Fig. 1 of the main article, while the following schematics give an overview of the currently available data formats to store satellite track data (Fig. S3), aircraft and cloud track data (Fig. S4), as well as intersection data (Fig. S5).

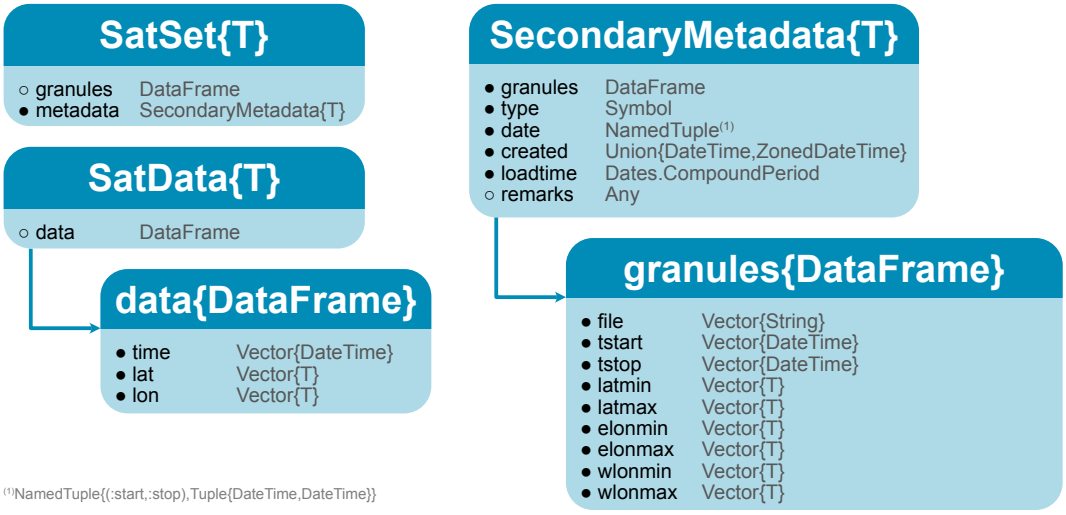


Figure S3. Schematic of the organisation of secondary (satellite) track data. Filled circles in the list of fields indicate mandatory fields with non-empty data formats.

In these schematics, mandatory fields of structs are indicated by filled circles while optional fields are marked by an open circle and may contain empty arrays or data structures, filler objects of type *Missing* or *Nothing* or *NaN*-values.

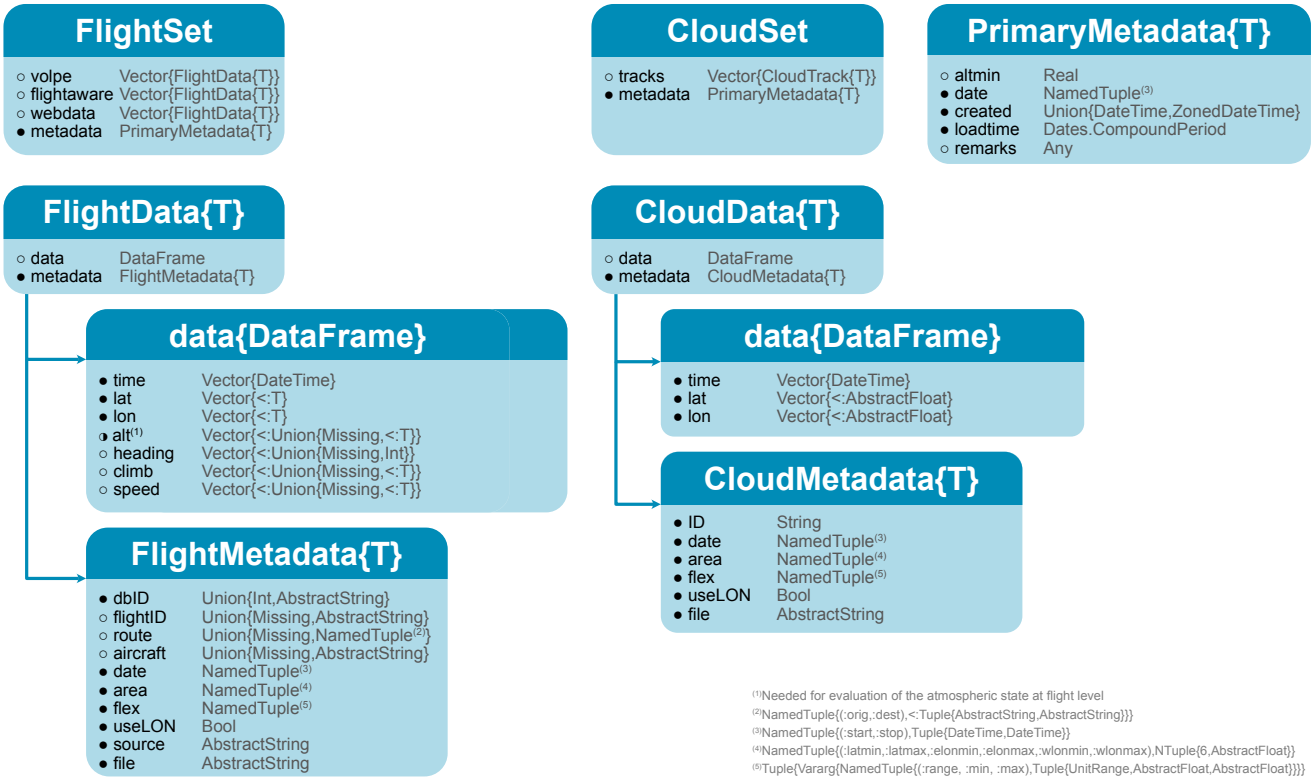


Figure S4. Schematic of the organisation of primary (flight and cloud track) data. Filled circles in the list of fields indicate mandatory fields with non-empty data formats.

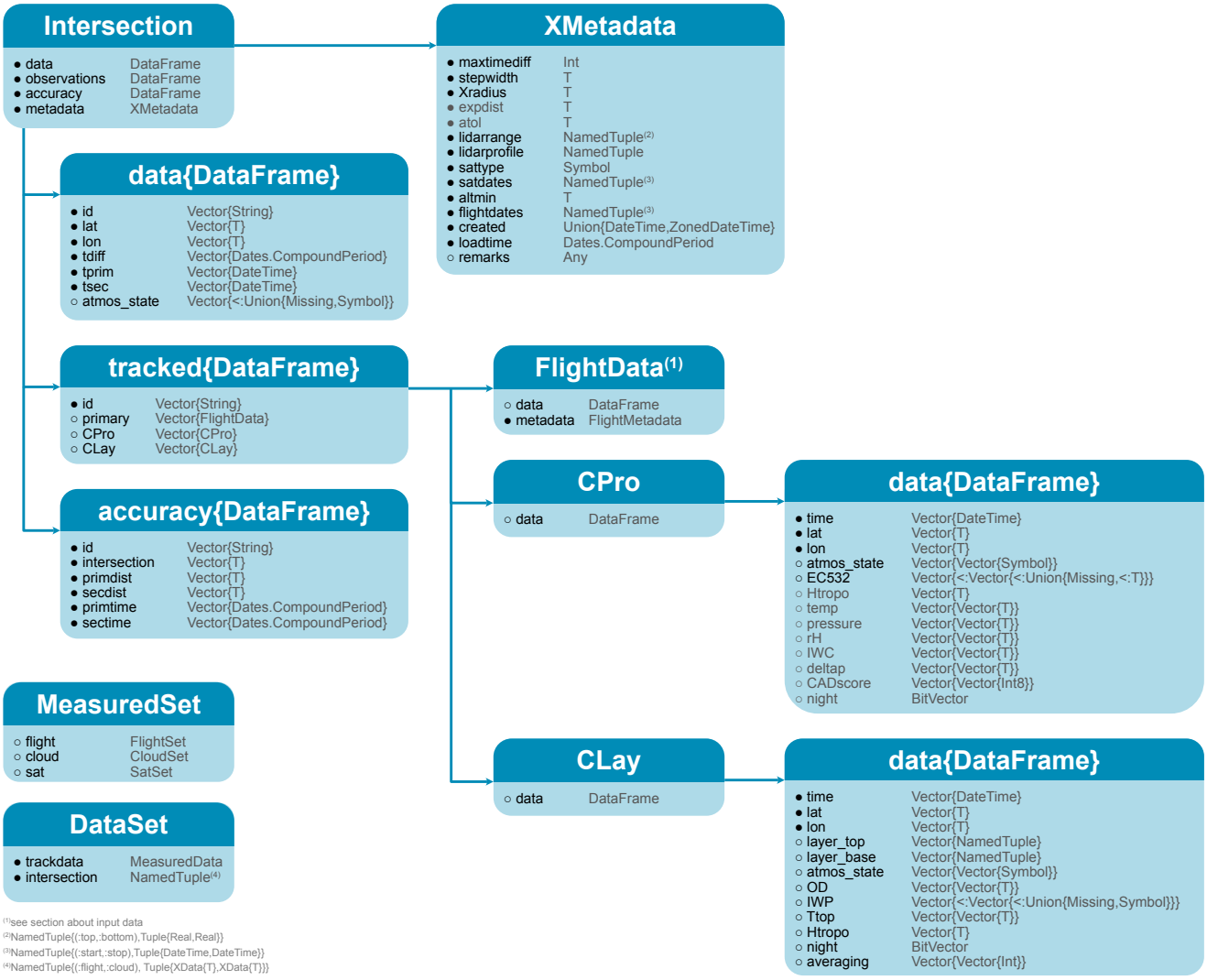


Figure S5. Schematic of the organisation of computed (intersection) and observational output. Furthermore, MeasuredSet stores all primary and secondary data combined and DataSet stores MeasuredSet data together with ComputedData. Filled circles in the list of fields indicate mandatory fields with non-empty data formats.

S2 Guides and example scripts

This section complements the explanations in the main paper by providing more detail and including examples demonstrating the installation and application of TrackMatcher. The section includes script boxes that contain minimal working examples and code boxes, which require additional code to run. If the script is available as Julia (`.jl`) file, it will be displayed in the bottom right corner of the script box. All code is written for Julia version 1.6. While the main article uses generic descriptions of primary and secondary datasets, this section of the ESM is more focused on application and will give more detail about the data sources used for the current project.

S2.1 Installation

TrackMatcher is an unregistered Julia package. It relies on the unregistered PCHIP package – a pure Julia implementation of the piecewise cubic Hermite interpolating polynomial – developed within the TrackMatcher framework. As a complication during installation, only registered dependencies are installed automatically. Therefore, PCHIP needs to be installed in advance. However, Julia's package manager can be used for both packages as a convenient tool:

We recommend installing TrackMatcher to a designated environment using the `activate` command in Julia's package manager (line 2 in Fig. S6 activates the current working directory as project environment). If this line is omitted, TrackMatcher gets installed to the default environment of the current Julia version and is universally accessible from all environments.

Alternatively, you can install TrackMatcher with the provided script *install.jl*. Place the script in a designated directory and run from the terminal `julia install.jl`. The script will install TrackMatcher to the main Julia environment of your current Julia version. To install TrackMatcher to a separate environment, place the script in a project folder and uncomment line 5 of the script to activate a new or existing environment or give the path to the working directory instead of the `.`.

S2.2 Running TrackMatcher

This section includes examples that assume a database consisting of aircraft and cloud track data as well as satellite data with the folder structure of the database given in Fig. S7. Please note that the example database is constructed as simple as possible. In real-case applications, for example, the folder structure and naming conventions of the AERIS/ICARE database should be used for the satellite data as given in the next section. It is also advisable to introduce subfolders for the different flight routes in the FlightAware dataset. For simplicity, it is furthermore assumed that Julia is run from the home folder.

S2.2.1 Conventions and restrictions

For TrackMatcher to run properly without any modifications, the following conventions must be obeyed.

Script 1: Package installation

```
1  julia> ⌨
2  pkg> activate # optional .
3  pkg> add https://github.com/LIM-AeroCloud/PCHIP.jl.git
4  pkg> add https://github.com/LIM-AeroCloud/TrackMatcher.jl.git
5  pkg> instantiate
6  pkg> ⌨ [backspace]
7  julia> import TrackMatcher as tm
```

example/install.jl

Figure S6. Example script for the TrackMatcher installation into an environment in the current folder.

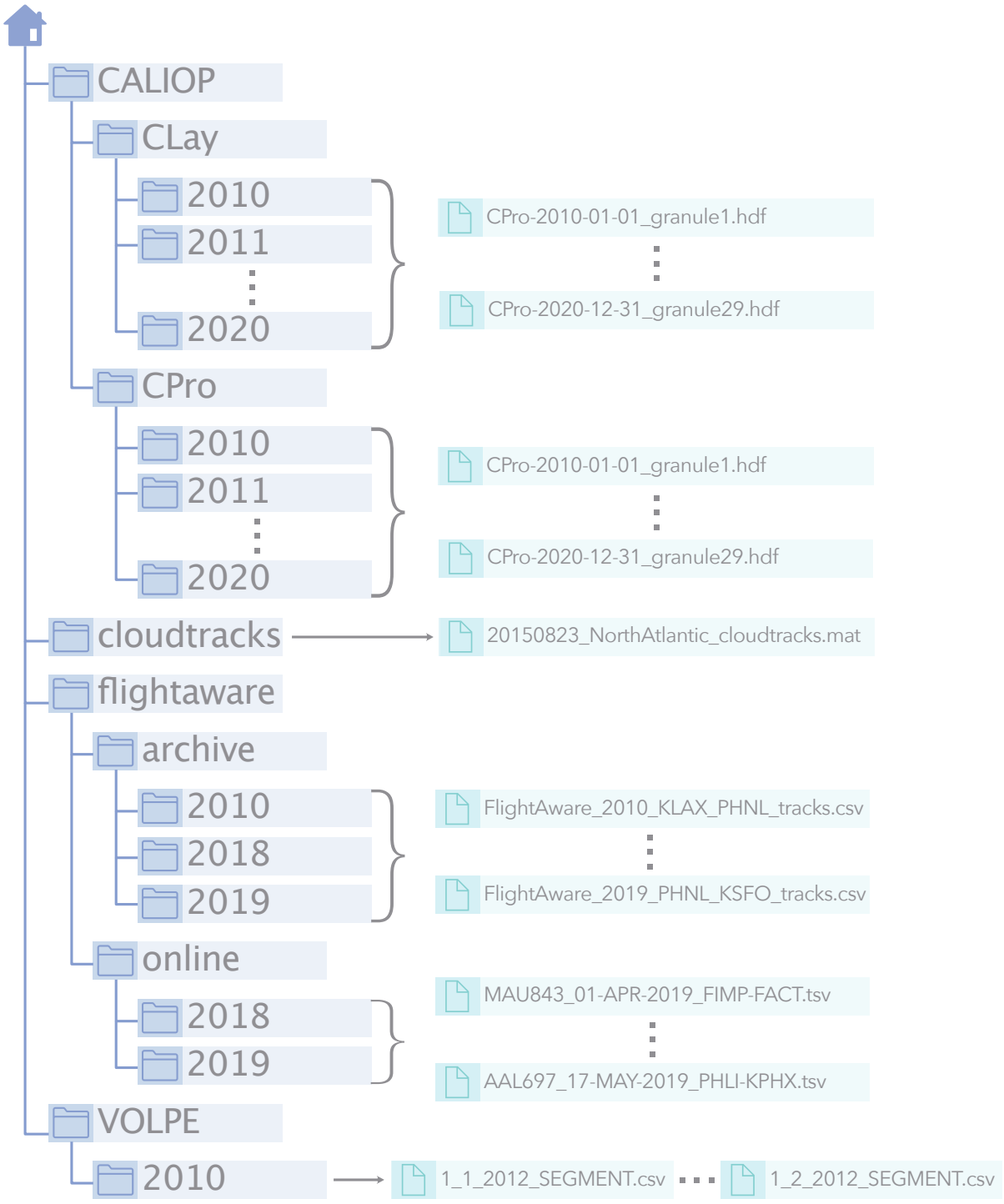


Figure S7. Schematic of the database folder and file organisation assumed in all examples of Sect. S2.

Note 1: Database format

Data columns from csv/tsv files are identified by name and/or column order.

Column names/order can be inferred from the `standardnames` input argument of function `checkcols!` in the unmodified constructors of the respective database struct.
Adjust read functions or add new functions for files with different data format. See Sect. S3 for more information about the programme structure.

Figure S8. Format of input data.

File format and names

TrackMatcher currently processes primary data from the following sources:

- AEDT Fuel Consumption and Emissions Inventory by the Volpe Center
- FlightAware commercial archive
- FlightAware web content
- Cloud tracks provided by Seelig et al. (2021)

Primary data are stored in text files with comma- (csv) or tab-separated values (tsv) or MATLAB files ending with ".mat". Tables in these files are identified by column name and/or order, which can be inferred from the column checks in the unmodified constructors of each track type (see Sect. S3 for more information about the programme structure). In the unmodified constructor of each track struct (`FlightSet`, `CloudSet` or `SatSet`), the routine `checkcols!` is called to ensure correct input data. Besides the correct column names and order, which can be seen from the array `standardnames`, data types are compared to `standardtypes` and the value range must match the range in any entry for a parameter in the dictionary `bounds`.

Furthermore, the copy/paste-content of FlightAware online data needs to be complemented by additional information about the flight ID, the route using the ICAO airport codes of the origin and destination, and the starting date of the flight. The online data will only contain the day of the week and flight-time plus all the tracking information. Flight times are in local time and converted to UTC as described in the next section. For TrackMatcher to receive all necessary information, naming conventions in Fig. S9 must be used for web data files.

Note 2: FlightAware web data file naming scheme

[Flight ID]_[dd]-[mmm]-[yyyy]_[ORIG]_[DEST].[ext]

[FlightID] . . . . . Flight identification number, e.g. "MAU844"
[dd] . . . . . 2-digit number of the day, e.g. "01"
[mmm] . . . . . 3-letter capitalised abbreviation of the month (in English), e.g. JAN
[yyyy] . . . . . 4-digit year, e.g. 2000
[ORIG] . . . . . ICAO code of the origin
[DEST] . . . . . ICAO code of the destination
[ext] . . . . . file extension, use either "tsv", "dat" or "txt"

Figure S9. Naming scheme for files holding FlightAware web content.

Note 3: CALIOP satellite data types

- Cloud profile data (CPro)
- Cloud layer data (CLay)

Figure S10. Data products derived from the CALIOP lidar used in TrackMatcher.

The only source of secondary track data is CALIOP data onboard the CALIPSO satellite. Currently, only data given in Fig. S10 can be processed in TrackMatcher.

We use data by the AERIS/ICARE data centre (https://www.icare.univ-lille.fr/calipso/). TrackMatcher is adjusted to the folder structure and naming conventions of the AERIS/ICARE database. In particular, the data type (cloud layer or profile data) is derived from the keywords CPro and CLay as indicated by the above list. These keywords must be part of the folder and file

Note 4: Satellite data file naming scheme

TrackMatcher is guaranteed to work with CALIPSO cloud layer or profile data as provided by the AERIS/ICARE Data and Services Centre. In particular, files names must include the [product] keyword (as explained below). Names for files with different data types as given in note 3 must be identical except for the [product] keyword.

Folder structure and names

CALIOP/[resolution][product].v[R].[MR]/[yyyy]/[yyyy]_[mm]_[dd]

File names

CAL_LID_L2_[resolution][product]-Standard-V[R]-[MR].[yyyy]-[mm]-[dd]T[HH]-[MM]-[SS]Z[DT].hdf

Legend

keyword	explanation	versions/values tested in TrackMatcher
[resolution]	horizontal averaging of the lidar data	05km
[product]	analysis product, e.g., aerosol/cloud layer or profile	CPro, CLay
[R]	major release of the version	4
[MR]	minor release of the version	20 (and all minor releases of v4)
[yyyy]	4-digit year[1]	2006 – present
[mm]	2-digit month[1]	01 – 12
[dd]	2-digit day[1]	01 – 31
[HH]	2-digit hour[1]	01 – 23
[MM]	2-digit minute[1]	01 – 59
[SS]	2-digit second[1]	01 – 59
[DT]	day-/night-time flag	D = day, N = night

[1] start of the granule

Figure S11. Naming scheme of CALIOP satellite data files.

> **Note 5: Time format in TrackMatcher**
>
> Times in model output are in UTC. Only times of creation in the metadata of a database are in local time using the `ZonedDataTime` format to save the time and time zone.

Figure S12. Time format in TrackMatcher.

names containing the respective data type and this type only. Moreover, all files must use the same naming conventions, where file names for files sharing the same time frame and area only differ in these keywords for the respective analysis products (see Fig. S11 for more information).

Time format

TrackMatcher uses UTC as standard time for any time points saved in the programme. The only exceptions are the times of creation saved in the metadata of any database struct for more convenience. These times are stored as local time with the time zone included. This is realised by the `ZonedDateTime` of the `TimeZones` package.

Online data by FlightAware needs conversion to UTC as it is shown in local time of the system, where the data were retrieved. Moreover, changes in daylight saving during a flight are not considered by FlightAware and the local time at the start is used throughout the whole flight. Thus, using the `localzone` function from the `TimeZones` package to automatically infer local time can lead to problems with flights during daylight saving changes as times would switch from summer to winter time or vice versa within the flight. TrackMatcher tries to retrieve time zone information from the first column header of the data table, where the time zone is given in parentheses. However, the time zone format, such as *CET* or *CEST* for Central European (summer) time, is unknown to the `TimeZones` package. A dictionary with the time zones is hardcoded into TrackMatcher as difference in hours to UTC. This avoids switches of daylight saving during a flight. Only when the time zone is not found in the dictionary, the local time is used. The list of time zones can be easily appended by adding new entries in `TrackMatcher.jl` in the section *Define time zones for FlightAware online data* below the package imports. Time zone codes must be preceded and succeeded by an underscore (`_`). For example, to add Eastern Standard Time, include the snippet given in Fig. S13 in TrackMatcher.jl.

S2.2.2 Default settings

The general order to calculate intercept points in pairs of trajectories with TrackMatcher is

1. Load the TrackMatcher package into the current workspace

Code example 1: Adding timezone support in TrackMatcher

```
zonedict["_EST_"] = tz.tz"UTC-0500"
```

> **Note 6: Timezone support for FlightAware online data**
>
> TrackMatcher currently offers support only for Central European (summer) time (CET and CEST). Add time zone support for your area using code example 1 or open a pull request or issue providing the desired time zone abbreviations at https://github.com/LIM-AeroCloud/TrackMatcher.jl/issues to get permanent time zone support in the next TrackMatcher version.

Figure S13. Hints and code examples for adding time zone support in TrackMatcher.

Script 2: Using TrackMatcher with default options

```
# Import all TrackMatcher functions into workspace
using TrackMatcher

# Load data
flights = FlightSet(
  volpe = "inventory",
  flightaware = "flightaware/archive"
  webdata = "flightaware/online/",
)
clouds = CloudSet("cloudtracks")
sat = SatSet("CALIPSO/CPro")

# Compute intersections
Xflight = Intersection(flights, sat)
Xcloud = Intersection(clouds, sat)
```

Figure S14. Example script for a TrackMatcher run under default settings.

2. Load all desired primary and secondary track data
3. calculate intercept points between primary and secondary trajectories

To have a complete calculation of intersections between the trajectories of two data sets with default settings, run, e.g.:

By default, a constructor for a struct only needs the folder path passed as a string to load the necessary input data from this directory. More than one argument can be given, if TrackMatcher should search several directories recursively. This can be useful, if you want to exclude data from certain years and your data is organised using different folders for every year.

Some adjustments are necessary for data sets with several sources. In the case of aircraft data, both VOLPE and FlightAware data sets use the csv file format. Therefore, data files cannot be identified by file extension alone and a identifier is needed. For TrackMatcher to be able to load data from the different sources correctly, only folders with data of a single source can be passed to the constructor of `FlightSet`.

To identify data sources correctly, data bases in `FlightSet` are loaded using the following keyword arguments:

Instead of giving the folder paths as variable number of arguments of strings, each folder path is given as keyword argument. Not all keyword arguments have to be used as typically data from only one source is used in model runs. If more than one directory should be searched for input files of the same source, files are passed as `Vector{String}` to the respective keyword argument of `FlightSet`. For only one data folder, the `String` can be passed directly to the keyword argument.

Note 7: Aircraft database sources and keywords

- `volpe`: VOLPE AEDT inventory
- `flightaware`: FlightAware archive
- `webdata`: FlightAware web content

Figure S15. Available sources for aircraft track data in TrackMatcher and their keywords (field names in the `FlightSet` struct) to address them.

Code example 2: Alternative flight database loading

```
flights = FlightSet(
  webdata = "flightaware/online/",
  flightaware = ["flightaware/archive/2018", "flightaware/archive/2019"]
)
```

Figure S16. Code example of possible ways to load aircraft track data into TrackMatcher.

Figure S16 gives an example how to load data from different sources either with a single or multiple source folders.

For the calculation of intersections, the `Intersection` struct only needs to be constructed from the `FlightSet` or `CloudSet` and `SatSet` structs. Currently, two separate calculations are needed for flight and cloud tracks to determine intersections with satellite ground tracks. However, for flight tracks, the trajectories from all database sources are combined to derive intersections with satellite ground tracks. In the above example, all intersection data is stored in variable `Xflight` for aircraft data and `Xcloud` for cloud-track data. Explore the fields `data`, `observations`, and `accuracy` to obtain information about the spatial and temporal coordinates of the intersections, further measurements in the vicinity of the intersections, and the accuracy of the calculations. Parameters in the the field `accuracy` are only indicators and are not derived from a proper error propagation.

S2.2.3 TrackMatcher options

Results in TrackMatcher can be influenced by pre-filtering data points or adjusting the precision of the calculation to gain a performance increase. Except for the optional argument `savesecondsattype` in the constructor of `Intersection`, all parameters are *keyword arguments* of the respective constructor of a database or `Intersection`. An overview of all parameters is given in Table 1 of the main article.

This section complements the main article by providing examples for several applications, where adjustments might be necessary. All code examples use the data structure from Fig. S7 as explained in Sect. S2.2.

Current investigations with TrackMatcher focus on aviation effects on cirrus clouds. Therefore, primary flight track data below $5000\,\mathrm{m}$ is disregarded. CALIOP lidar data is processed only up to $15\,\mathrm{km}$. Above this height, no commercial aviation is expected. If you want to consider the whole atmospheric column, you can change the default settings and consider aircraft data below the minimum threshold `altmin` and lidar data above the default in a column range `lidarrange = ([max], [min])` as demonstrated in script 3 of Fig. S17.

All code examples in the following assume the same aircraft and satellite data from script 3 in Fig. S17 has been loaded. Thus, lines 1 to 7 are omitted to avoid repetition.

In TrackMatcher, data cannot only be filtered by altitude, but also by time. TrackMatcher monitors the overpass time of each trajectory at the intersection and calculates the delay time. If the time period between the overpasses increases above a threshold (in minutes), intersections are disregarded. With a focus on atmospheric applications, a default value of $30\,\mathrm{min}$ is chosen to keep interference from advection minimal in the data analysis without a complex investigation of the local air flow patterns. Code example 3 of Fig. S17 shows how to increase the delay time threshold to $2\,\mathrm{h}$, if advection is of no concern.

Code example 4 of Fig. S17 demonstrates ways to influence the storage of measurements in the vicinity of intersections and how to exclude uncertain intersection calculations. Over the open ocean aircraft tracking is considerable less frequent. There can be gaps of more the $1000\,\mathrm{km}$ distance. Any intersections found in those areas must be considered uncertain.

By default, all intersections are saved and the uncertainty of the calculations can be inferred from the parameters `primdist` and `secdist`, which give the distance in meters of the intersection to the nearest measured track point of the primary and secondary trajectory, respectively. Gaps in satellite data are rarely a problem. If the volume of the track data allows the exclusion of uncertain calculations, a threshold (in meters) can be defined with the keyword argument `expdist` that suppresses storage of these data.

Script 3: Adjust altitude ranges

```
# Import TrackMatcher and all exported functions into workspace
using TrackMatcher

# Load data
flights = FlightSet(
  volpe = "inventory",
  flightaware = "flightaware/archive",
  webdata = "flightaware/online/",
  altmin = 0
)
sat = SatSet("CALIPSO/CPro")

# Compute intersections
intersections = Intersection(flights, sat, lidarrange=(Inf,-Inf))
```

Code example 3: Parameters controlling the delay at intersections

```
intersections = Intersection(flights, sat, maxtimediff = 120)
```

Code example 4: Parameters controlling observational data

```
intersections = Intersection(flights, sat, expdist = 100_000,
  primspan = 50, secspan = 50)
```

Code example 5: Parameters controlling accuracy of calculation

```
intersections = Intersection(flights, sat,
    Xradius = 1000,
    stepwidth = 0.1,
    atol = 1
)
```

Figure S17. Examples of TrackMatcher settings to influence model performance and output.

Furthermore, you can adjust the number of measurements that will be saved in the vicinity of intersections. TrackMatcher will always save the closest track point of the primary trajectory and the closest CALIOP lidar measurement from the secondary trajectory. In addition to the data type that was used for the calculation, the user can add measurements of any type given in note 3. Besides measurements at the closest track point to the intersection, data from any number of additional points of the trajectory before and after the closest point can be saved using the keyword arguments `primspan` and `secspan` for the primary and secondary trajectory, respectively. By default no additional data points are saved for the primary trajectory (`primspan = 0`) and 15 additional measurements to either side of the closest point to the intersection of the secondary trajectory are saved (`secspan = 15`). Thus, in total 31 data points are saved including data from the closest track point.

As explained in Sect. 2.4.3 of the main article, TrackMatcher can predict duplicate intersections for one and the same intersection in rare cases. For near-parallel primary and secondary trajectories, multiple intersections can occur, where the user is interested in only one example measurement in the region. The keyword argument `Xradius` of the modified `Intersection`

constructor exists to control this behaviour. For any intersection, only the intersection with the the highest accuracy is saved within `Xradius` in meters. For equally accurate calculations, the intersection with the least time delay of the overpass times at the intersection is saved. `Xradius` is set to 20 000 m by default. It might be wise to reduce the default setting, for example, for trajectories of research flights with many tight loops or near-parallel trajectories, where you want a higher or lower sampling rate.

As the main article explains in Sect. 2.4.2, TrackMatcher needs to interpolate the irregular track points of the primary and secondary trajectory to obtain data points with either equidistant latitude or longitude values. By default, the `stepwidth` parameter for PCHIP interpolation is set to 0.1°, i.e. approximately 10 km. Decreased `stepwidth` values can provide better interpolation, which can be important for highly curved trajectories, trajectories with sharp turns or highly irregular track points. However, refining the step widths will demand more computation time.

Data of the secondary data set is restricted to the time frame ± `maxtimediff` minutes and to the area within a bounding box of the flight track. Due to rounding errors, this can mean that intersections at the edges of the box are not detected. The `atol` parameter exists to increase the bound box by a defined number of degrees (default: `0.1 °`). As shown in the sensitivity studies of the main article (Sect. 3.4 and Table 5), increasing the parameter to `1 °` might lead to additional intersection detections without performance loss. Example box 5 in Fig. S17 demonstrates the use of the `atol` parameter.

By default, TrackMatcher saves all floating point numbers in single precision (`Float32`). This accuracy is completely sufficient for all purposes of TrackMatcher. Moreover, satellite data is only available in single precision. However, `Float64` can be passed as struct parameter to the data structs `FlightSet`, `CloudSet`, and `SatSet` to load data in double precision or as parameter to `Intersection` to enforce data conversion to and subsequent computations in double precision. The user can have a mixture of precisions as in script 4 (Fig. S18), where aircraft data is processed with double precision and single precision is kept in the satellite data. The `Float32` parameter in the `SatSet` constructor is optional for single precision, but was explicitly given to demonstrate the use of the keyword in `SatSet`. `Float64` could also have been used as parameter in `Intersection` to define the

Script 4: Parameters controlling input and output data

```
# Import TrackMatcher and all exported functions into workspace
using TrackMatcher

# Load data
flights = FlightSet{Float64}(
  volpe = "inventory",
  webdata = "flightaware/online/",
  flightaware = "flightaware/archive",
  odelim = '\t',
  savedir = "rel"
)
sat = SatSet{Float32}("CALIPSO", type=:CPro, savedir = "rel")

# Compute intersections
intersections = Intersection(flights, sat, true,
  savedir="rel",
  remarks="test script 4"
)
```

Figure S18. TrackMatcher settings for control of input and output data.

floating point precision. If obsolete, floating point precision will be inferred from the promotion of the floating point precision from both input data structs as in script 4 of Fig. S18.

When the copy paste content of FlightAware online data is read, TrackMatcher autodetects delimiters. Delimiters are typically tabulators. When whitespace is trimmed in the input files, this can cause problems for empty columns at the end of a table. As whitespace is trimmed, the delimiters of the columns are deleted and the column number does not match the actual number of columns. Thus, tabulators are not identified as delimiters. Therefore, you can force a delimiter of the online data files with the keyword argument `odelim` as demonstrated in script 4 (Fig. S18).

For satellite data, the data type (`CPro` jor `CLay`) is automatically detected based on the keywords in the name. Only one type can be stored in `SatSet`. If there is a mixture of data types in the given folder(s), only the data type with the majority of the first fifty files found is used. Hence, in script 4 only `CLay` data would be stored as the CLay folder in the CALIOP folder comes alphabetically before the CPro folder. If you don't want to specify subfolders, but still save another data type, you can force the data type with the keyword argument `type` (see script 4, Fig. S18).

Intersection saves measured lidar columns in the vicinity of the intersection (see also explanations for keyword argument `secspan`). Be default, only the data type that was used for the derivation of intersections is saved. If you want to save the corresponding layer or profile data, set `savesecondsattype` to `true`. This feature demands files of the second data type in the same main folder as the data used for the calculations using the same folder structure and naming conventions just with the keywords `CPro` and `CLay` swapped (see also notes 3 and 4, Fig. S10 and S11).

To be able to reproduce model runs, TrackMatcher saves the file names including the directory of the input data files. For satellite data, this is additionally important to save observations as initially only time, latitude, and longitude are saved. Only, when an intersection is found, additional observational data is retrieved from the files with the file names saved in the metadata. By default, file names are saved as absolute folder paths. This means that data cannot typically be loaded on one system saved and being re-imported on another system, where TrackMatcher computes intersection. As the home directories typically vary on different systems, TrackMatcher would fail to save satellite observations, when being given absolute folder paths. Therefore, the default behaviour can be changed with the `savedir` keyword argument. When `savedir = "rel"` is passed to `FlightSet`, `CloudSet` or `SatSet` as demonstrated in script 4 (Fig. S18), relative folder paths are saved. Such if you keep an identical folder structure within you working directory, relative folder paths enable you to load data on one system and compute intersections on another. This can be useful for large data sets, when you have limited disk space available at a system, but want to do the computation expensive task on finding intersections on it. Moreover, `savedir` can be set to an empty string (`""`) or `false`. In this case, directories are saved as given in the arguments of the constructor (either relative or absolute). When the latter options are used in the `Intersection` constructor, no observations are saved to avoid the problems of model runs on separate systems. Finally, you can attach any remarks or data in any format to the metadata of any database `FlightSet`, `CloudSet`, `SatSet` or `Intersection`.

For convenience, additional constructors exist to combine processes in a model run in a one-line command. The `MeasuredSet` loads the input data of the primary and secondary data set in a single step. The `DataSet` constructor loads first all input data into the `MeasuredSet` struct and additionally calculates intercept points. While this is the most convenient method to compute intersections, users should be cautious with large data sets, especially when fatal errors are possible during model runs. Data with these both methods is only saved at the very end, and if a process fails, previously successful processes are lost. For large data sets, where even the import process of data can take several hours, it might be better to separate the processes and save data after each step.

Script 5 (Fig. S19) shows a use-case example of both methods. Keyword arguments of any constructor for loading primary or secondary input data or computing intersections can be passed to the constructor of `MeasuredSet` or `DataSet`. It is also possible to calculate intersections of both, aircraft and cloud primary trajectories. The output is stored separately as described in in a `NamedTuple` with fields `Xflight` and `Xcloud` for aircraft and cloud data.

The only difference in the combined processes compared to the separate computations is the handover of the directories with input data to TrackMatcher to the constructor as well as the passing remarks. As the remarks keyword is used in each constructor and directories are passed as arguments, it would not be clear in the `MeasuredSet` or `DataSet` constructor, for which process a remark or directory string is meant. Therefore, folder paths and remarks are passed as vector of pairs with keys identifying the

Script 5: Combined processes for data processing in TrackMatcher

```
# Import TrackMatcher and all exported functions into workspace
using TrackMatcher

# Load all input data
input = MeasuredSet{Float64}(
  ["volpe" => "inventory",
  "webdata" => "flightaware/online/",
  "sat" => "CALIOP/"],
  odelim = '\t',
  savedir = "rel"
)

# Load input and compute intersections
data = DataSet(
  ["volpe" => "inventory",
  "webdata" => "flightaware/online/",
  "cloud" => "cloudtracks",
  "sat" => "CALIOP/"],
  maxtimediff=60,
  remarks=["sat" => "layer data!", "Xflight => "no FlightAware Archvie!"]
)
```

Figure S19. Examples for convenience constructors combining the data import of all input data with `MeasuredSet` and additionally also calculating intersections for all loaded data with `DataSet`.

Note 8: Available keys for directories and remarks in combined processes

- `"volpe"`: aircraft data by the Volpe Data Center
- `"flightaware"`: FlightAware archive Data
- `"webdata"`: FlightAware web content
- `"cloud"`: Cloud track data by Seelig et al. (2021)
- `"sat"`: CALIOP satellite Data
- `"Xflight"`: passing remarks to `Intersection` struct for aircraft data
- `"Xcloud"`: passing remarks to `Intersection` struct for cloud data

Figure S20. Possible keys assign directories with input data or remarks to the correct constructor in the `MeasuredSet` or `DataSet` constructor.

process for which the arguments in the values are meant. For the arguments in the values, the same conditions apply as for the original arguments in the single process. Possible processes (keys) that can be address are:
/Users/work/Documents/LIM/PACIFIC/TrackMatcherPaper/ESM/S2-Scripts.tex

Code example 6: Activate Julia environment

```
import Pkg
Pkg.activate("path/to/environment")
```

example/example.jl

Figure S21. Activating environments by scripts in Julia.

S2.2.4 Example run

For a quick test to get acquainted with TrackMatcher, we have added an example in the ESM. It can be found in the folder example. For the example to work, TrackMatcher needs to be installed in the main environment. To install TrackMatcher, follow the installation instructions and don't activate any environments in the package or alternatively add the lines given in Fig. S21 at the beginning of the script example.jl. Additionally, you need to install the JLD2 package to save model output for later analysis. This can be done by typing add JLD2 in the package manager.

Script 6: Run example script from console

```
julia example.jl
```

example/

Script 7: Analysing results from example script

```
$> julia
julia> # Import necessary packages
julia> using TrackMatcher
julia> import JLD2
julia> # Load saved results from Script 6
julia> @JLD2.load "data/results/Xex.jld2"
julia> # Show results in the Xex struct
julia> Xex.data
julia> Xex.data.tidff
julia> Xex.observations.primary[1]
julia> Xex.accuracy
julia> # Exit Julia
julia> exit()
```

example/

Note 9: Importing JLD2 data

When importing saved TrackMatcher data with JLD2, TrackMatcher needs to be imported to the current workspace before the data import, so JLD2 recognises the struct formats of TrackMatcher.

Figure S22. Running the example script and analysing results.

Row	id String	lat Float32	lon Float32	alt Float32?	tdiff Compound…	tprim DateTime	tsec DateTime	atmos_state Symbol?
1	WD-ANZ192/28-JAN-2019/YPAD-NZAA-1	-38.5132	155.012	12496.8	23 minutes, 52 seconds	2019-01-28T03:29:07	2019-01-28T03:52:59	ci
2	WD-CCA838/27-JAN-2019/PHNL-ZBAA-1	53.4153	165.385	11582.4	16 minutes, 48 seconds	2019-01-27T15:12:58	2019-01-27T15:29:46	clear
3	WD-CCA838/27-JAN-2019/PHNL-ZBAA-2	53.1573	140.626	11582.4	-9 minutes, -31 seconds	2019-01-27T17:17:53	2019-01-27T17:08:22	clear
4	WD-JAL770/27-JAN-2019/RJAA-PHKO-1	32.3249	158.006	11582.4	-4 minutes, -56 seconds	2019-01-27T15:40:34	2019-01-27T15:35:38	clear
5	WD-JAL779/26-JAN-2019/PHKO-RJAA-1	24.7473	174.483	11277.6	14 minutes, 16 seconds	2019-01-27T01:39:37	2019-01-27T01:53:53	clear, 5×4 DataFrame

Figure S23. Output of the `data` field in the intersection results.

Input data

The example contains the complete folder structure needed as well as the flight data saved to `example/data/flights`. Satellite data can be obtained from the AERIS/ICARE data centre (https://www.icare.univ-lille.fr/calipso/) free of charge. By registering, you agree to obey the rules of the AERIS/ICARE data centre. There is also a tool within the TrackMatcher framework to help you with downloads of the satellite data after registration available at GitHub under https://github.com/LIM-AeroCloud/ICARE.jl. For the example, `05kmCPro.v4.20` data, i.e. cloud profile data with a horizontal resolution of $0.5\,\mathrm{km}$ in version 4.20, were used.

Running the example

Once satellite data has been downloaded you can simply run the script `example.jl` as given in Fig. S22. This will generate a file `Xex.jld2` in the `example/results` folder. Now you can start a new Julia session, open the `jld2` file, and analyse the results. Alternatively, you can open Julia in the `example/` directory and directly retrace the steps in the `exmaple.jl` script. This way, you do not have to have the JLD2 package installed, if you do not want to save output for later analysis. Results are

Script 8: Run and analyse extended example script

```
$> julia extended_example.jl
$> julia
julia> # Import necessary packages
julia> using TrackMatcher
julia> import JLD2
julia> # Load saved results from Script 6
julia> @JLD2.load "data/results/X_extended.jld2"
julia> # Analyse results
julia> input.flight.webdata[1].metadata.date.start # start time of flight 1
julia> input.sat.granules[1] # 1st granule of sat data
julia> # 1st granule of sat data accessed from overall run:
julia> intersections.trackdata.sat.granules[1]
julia> # Time differences between flight & satellite overpass at intersection:
julia> output.data.tdiff
julia> # Time differences (alternative access with second overall method)
julia> intersections.intersection.flight.data.tdiff
julia> exit() # exit Julia
$>
```

example/

Figure S24. Running the extended example script and analysing results.

directly available from the `Xex` variable. For comparison, results from the author's model runs have been added in the file `data/results/Xex_default.jld2`, which should be equal to the results of the example script.

The example should find 5 intersections. One intersection is within a cirrus cloud, which can be derived from the `atmos_state` column in the `data` field of the intersection data being `ci` instead of `clear`. For another flight, 2 intersections where found as easily visible from the `id` in the intersection data ending with `-2`. Output of the intersection `data` field is given in Fig. S23.

Example combining data import and intercept calculations

For cases with small databases as in this example, it convenient to use constructors that combine processes as given in the `extended_example.jl` script. With the `MeasuredSet` constructor, input data can be loaded in one step, which can then further be processes with the `Intersection` constructor in the usual way. Additionally, the `DataSet` constructor can be used to load data and calculate intersections all in one step.

All constructors accept the parameters (keyword arguments) from the constructors of each individual process. Only the processing of the folder locations has changed slightly. They are passed as vectors of pairs with the keywords given in the list below. The values of each pair can be strings or vectors of strings for multiple folder location. Detailed information can also be found by calling help on each constructor typing `?` in the Julia console (REPL) and in the opening help menu the name of the constructor.

Run the `extended_example.jl` script as indicated in Fig. S24 or each line of the script individually in a Julia session as explained in the previous example. The `extended_example.jl` script will generate the results file `data/results/Xextended.jl`, which can be compared to the author's results in `data/results/Xextended_default.jl`.

- `"volpe"`: AEDT Fuel Consumption and Emissions Inventory by the Volpe Center
- `"flightaware"`: FlightAware commercial archive
- `"webdata"`: FlightAware web content
- `"cloudtracks"`: cloud track data by Seelig et al. (2021)
- `"sat"`: CALIOP cloud profile or layer data (by the AERIS/ICARE data centre)

S3 Programme Structure

All source code is available in the *src* folder from the GitHub repository https://github.com/LIM-AeroCloud/TrackMatcher.jl.git. Table S1 gives an overview of the source files and the type of functions they contain. Further help of the functions purpose can be obtained from their docstrings. Docstrings can be accessed through the help menu in Julia by typing `?<function name>` in the REPL.

To help users and developers adjust the programme to their needs, Fig. S25 to S28 give simplified schematics of the programmes routines and structure. In essence, the schematics show all routines defined in the TrackMatcher source code and their function calls to subordinate routines. Routines are colour-coded according to the source files they are contained in. Figure S25 shows routines related to loading primary data and Fig. S26 shows routines tasked with loading satellite track data and observations. In Fig. S28, routines responsible for track interpolation and finding intercept points as well as constructors for triggering the combined processes of loading all necessary primary and secondary input data and optionally calculating intercept points in one model run are shown. Figure S27 highlights routines responsible for data checks in the constructors of important TrackMatcher structs.

In short, the modified constructors take strings of directories, where the input data is stored and the `findfiles!` routines checks all directories recursively for input files. Function `convertdir` then saves the directories in the desired absolute or relative format. Next, routines related to loading each data format are called and input data are transformed to a unified format with SI units.

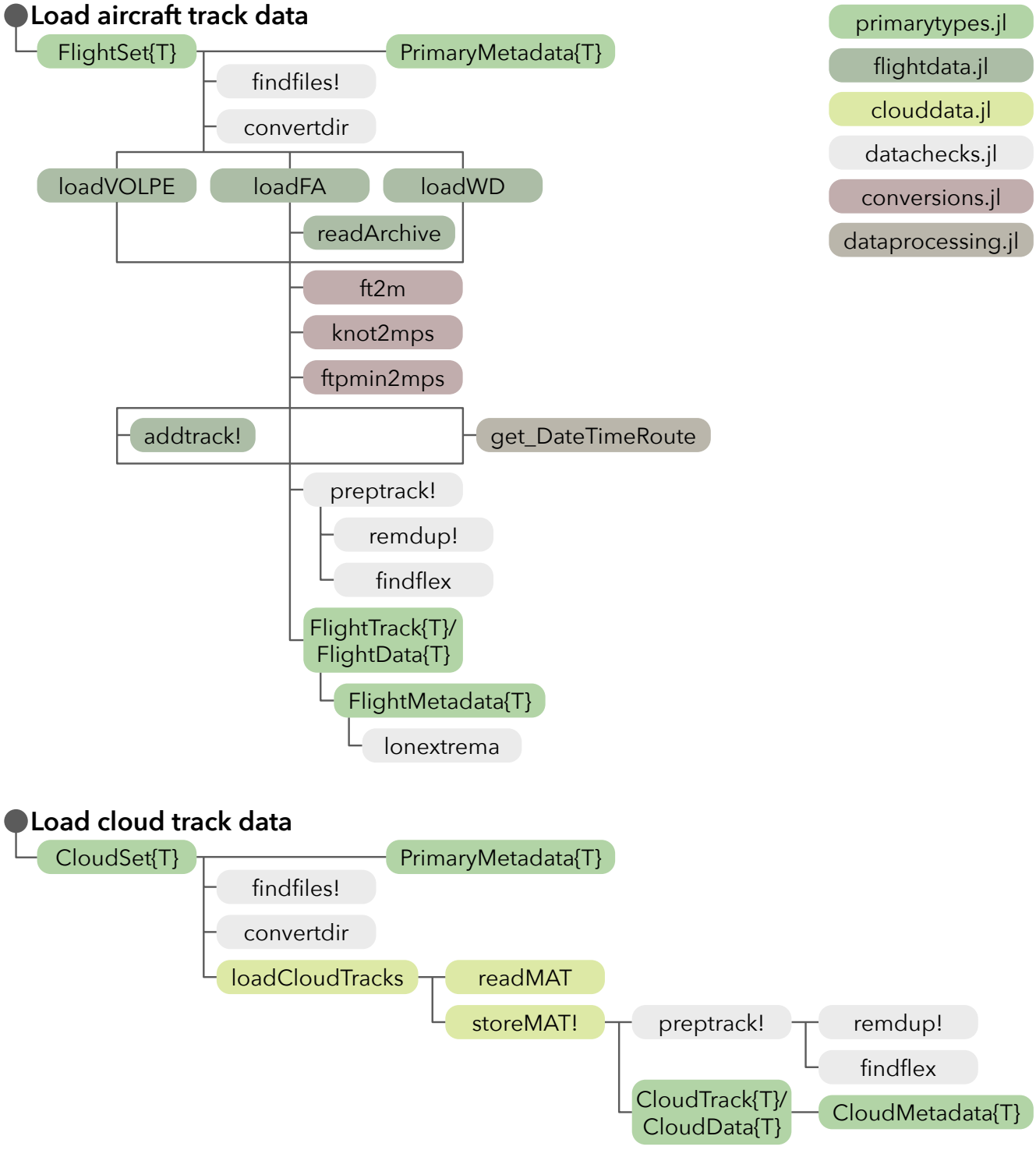


Figure S25. Simplified schematic of the organisation of functions related to loading primary data in TrackMatcher.

Table S1. Source files in TrackMatcher and their content.

source file	purpose	routines
clouddata.jl	loading cloud track data	loadCloudTracks, readMAT, storeMAT!
conversions.jl	type, format, and unit conversions	Float16, Float32, Float64, convertFloats!, convertUTC, earthradius, ft2m, ftpmin2mps, knot2mps
datachecks.jl	system scans, data checks and corrections	convertdir, findfiles!, remdup!, findflex, checkcols!, definebounds, checkbounds!, findbyname!, findbyposition!, findbytype!, correctDF!, checkDBtype, preptrack!, closest_points, lonextrema, withinbounds
dataprocessing.jl	general data processing	abs, addX!, add_intersections!, find_timespan, get_flightdata, get_DateTimeRoute, get_satdata, interpolate_time, init_dict, trim_vec!
flightdata.jl	load aircraft data	loadVOLPE, loadFA, loadWD, addtrack!, readArchive
lidar.jl	processing CALIOP lidar data	get_lidarheights, get_lidarcolumn, classification, feature_classification, atmosphericinfo
match.jl	match tracks, find intersections	find_intersections, findoverlap, interpolate_trackdata, interpolate_satdata, findXcoords
outputtypes.jl	structs with intersection data or to trigger combined calculations	XMetadata, XData, Intersection, MeasuredData, MeasuredSet, Data, DataSet
primarytypes.jl	structs for primary track data	FlightMetadata, CloudMetadata, PrimaryMetadata, FlightData, FlightTrack, FlightSet, CloudData, CloudTrack, CloudSet, PrimarySet
sattypes.jl	structs with satellite track data and observations	SecondaryMetadata, SatData, SatTrack, SatSet, CLay, CPro
TrackMatcher.jl	load dependent packages and include files, time zone setting, export functions, type tree/abstract types	—

Tracks are prepared for interpolation by identifying the prevailing direction, splitting it in segments with strictly monotonic x-data and removing duplicate track points. Finally, data are stored in structs in the TrackMatcher format along with metadata for each individual track in the primary data and overall data set information for both primary and secondary track data.

To calculate intercept points from the structs with the primary and secondary track data, routines are called to find overlap regions in the pairs of trajectories, interpolate the track data and find intersections (Fig. S28). Matching the trajectories is the most complex process, requiring function calls to many different subroutines regulating time interpolation, the retrieval of observational data, the determination of the atmospheric state at the three-dimensional intercept point and data storage of computed intercept points and relevant metadata in a unified format.

To derive the atmospheric conditions, the lidar resolution data are retrieved at each start of intersection computations. Further routines to retrieve the atmospheric condition at the time, coordinates, and altitude of the intersection are called when calculating the intercept points and storing satellite observations (compare Fig. S26 and Fig. S28).

Finally, routines exist to check for the correct input format, which are applied in the unmodified constructors of important TrackMatcher structs (see Fig. S27). For convenience, loading primary and secondary input data can be combined with the `MeasuredSet` constructor and, optionally, combined with the intersection calculation in a single model run with the `DataSet` constructor (see Fig. S28).

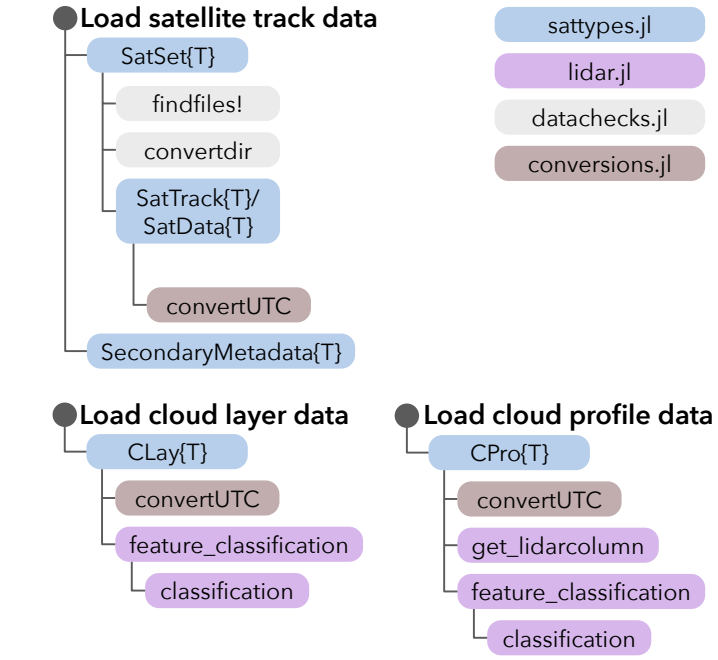


Figure S26. Simplified schematic of the organisation of functions related to loading satellite track data and observations in TrackMatcher.

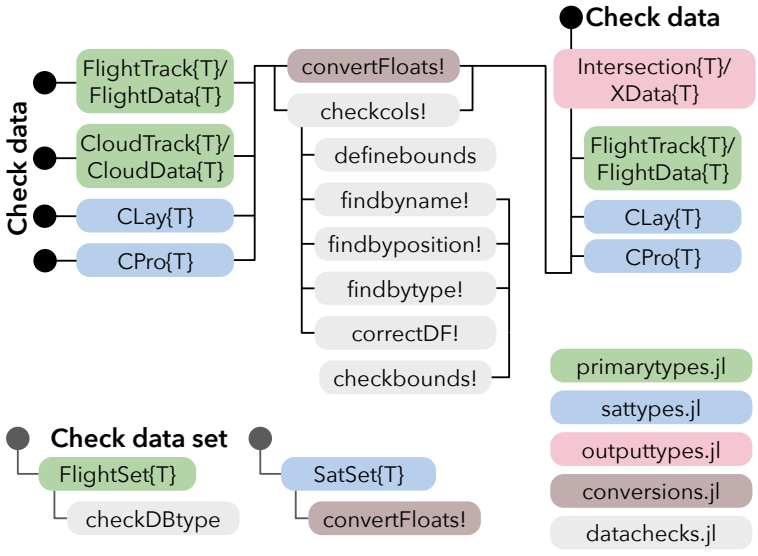
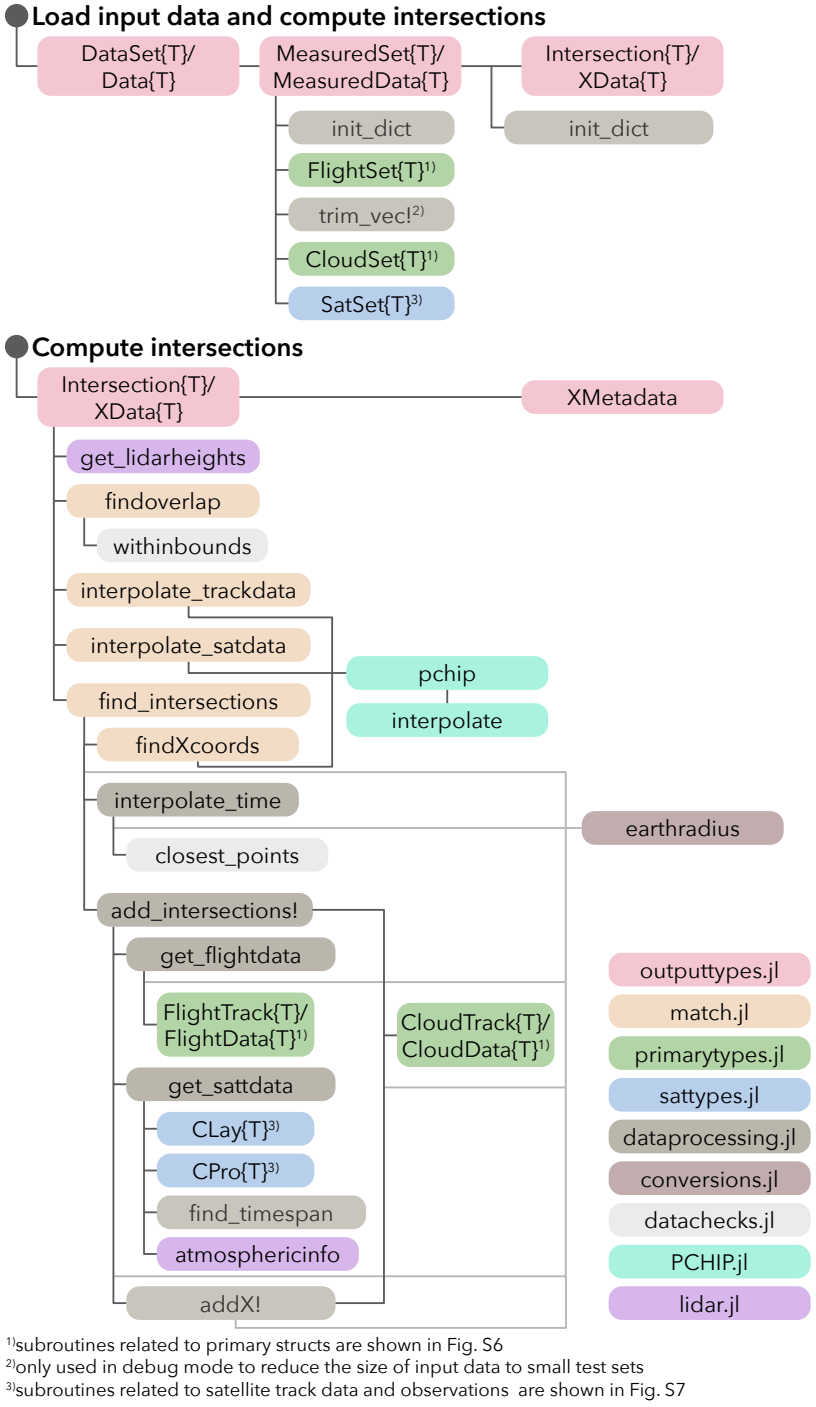


Figure S27. Simplified schematic of the organisation of functions related to data checks in TrackMatcher.



1)subroutines related to primary structs are shown in Fig. S6
2)only used in debug mode to reduce the size of input data to small test sets
3)subroutines related to satellite track data and observations are shown in Fig. S7

Figure S28. Simplified schematic of the organisation of functions related to computing intercept points or triggering combined processes in TrackMatcher.

References

Seelig, T., Deneke, H., Quaas, J., and Tesche, M.: Life cycle of shallow marine cumulus clouds from geostationary satellite observations, accepted by the Journal of Geophysical Research - Atmospheres, 2021.