Package 'transittraj'

November 12, 2026

Title Reconstruct and Visualize Transit Vehicle Trajectories

Version 1.2.0

Date 2026-11-11

Description Today's public transit vehicles produce a large amount of automatic vehicle location (AVL) data. This data is very useful for planning and performance studies, but can be noisy, error-prone, and sparse. This package provides tools for cleaning AVL point data and turning it into continuous, differentiable, monotonic, and invertible vehicle trajectory functions, based on the work of Robbennolt et al. (2025) <doi:10.48550/arXiv.2509.00119> and Huang et al. (2023) <doi:10.1109/ITSC57777.2023.10422524>.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.3

URL https://utel-uiuc.github.io/transittraj/

BugReports https://github.com/UTEL-UIUC/transittraj/issues

Imports dplyr, geos, gganimate, ggnewscale, ggplot2, ggspatial, hms, ivs, leaflet, magrittr, purrr, rlang, sf, slider, tidyr, tidytransit, viridis

Depends R (>= 3.5)

LazyData true

Suggests knitr, prettymapr, rmarkdown, spelling, testthat (>= 3.0.0)

VignetteBuilder knitr

Config/Needs/website rmarkdown, vembedr

Config/testthat/edition 3

Language en-US

NeedsCompilation no

Author Benjamin O'Brien [aut, cre, cph], Lewis Lehe [aut]

Maintainer Benjamin O'Brien <obrienbenjaminj@gmail.com>

Repository CRAN

Date/Publication 2026-11-11 23:30:02 UTC

Contents

clean_incomplete_trips . . . 2
clean_jumps . . . 4
clean_overlapping_subtrips . . . 6
export_animation . . . 7
filter_by_route . . . 9
get_gtfs_service_dates . . . 10
get_gtfs_trajectory_fun . . . 11
get_linear_distances . . . 15
get_shape_geometry . . . 17
get_stop_distances . . . 18
get_trajectory_fun . . . 19
get_trip_extremes . . . 21
group_trajectories . . . 22
lacmta_avl . . . 23
lacmta_gtfs . . . 24
make_monotonic . . . 26
new_transittraj_data . . . 28
plot.avltrajectory_group . . . 29
plot_animated_line . . . 30
plot_interactive_gtfs . . . 35
plot_trajectory . . . 37
predict.avltrajectory_group . . . 40
print.avltrajectory_group . . . 44
project_onto_route . . . 45
summary.avltrajectory_group . . . 46
trim_trips . . . 47
validate_monotonicity . . . 48
validate_tides . . . 49

Index **51**

`clean_incomplete_trips` *Filter out entire trips which do not meet distance or duration requirements*

Description

This function identifies trips that do not meet some acceptable duration and distance traveled ranges, or that have large time or distance gaps in the middle. Violating trips are removed.

Usage

```
clean_incomplete_trips(
  distance_df,
  max_trip_distance = Inf,
  min_trip_distance = -Inf,
  max_trip_duration = Inf,
  min_trip_duration = -Inf,
  max_distance_gap = Inf,
  max_time_gap = Inf,
  return_removals = FALSE
)
```

Arguments

distance_df	A dataframe of linearized AVL data. Must include trip_id_performed, event_timestamp, and distance.
max_trip_distance	Optional. The maximum distance traveled over one trip, in units of input distance. Default is Inf.
min_trip_distance	Optional. The minimum distance traveled over one trip, in units of input distance. Default is -Inf.
max_trip_duration	Optional. The maximum duration of one trip, in seconds. Default is Inf.
min_trip_duration	Optional. The minimum duration of one trip, in seconds. Default is -Inf.
max_distance_gap	Optional. The maximum change in distance between two observations, in units of input distance. Default is Inf.
max_time_gap	Optional. The maximum time between two observations, in seconds. Default is Inf.
return_removals	Optional. A boolean, should the function return a dataframe of trips removed and why? Default is FALSE.

Value

The input distance_df, with violating trips removed. If return_removals = TRUE, a dataframe of trips removed and why.

Examples

```
# Set my parameters
my_min_dist <- 1000
my_max_gap <- 1000

# Get input data
lineE_no_jumps <- new_transittraj_data("clean_jumps")
```

```
dim(lineE_no_jumps)

# Run function
lineE_clean_trips <- clean_incomplete_trips(distance_df = lineE_no_jumps,
                                            min_trip_distance = my_min_dist,
                                            max_distance_gap = my_max_gap)
dim(lineE_clean_trips)
head(lineE_clean_trips)
```

`clean_jumps` *Apply median filters to detect large jumps (i.e., outliers) in trajectories.*

Description

Noise in GPS trajectories can manifest itself as one or more points lying far away from points recorded at a similar time. This function identifies these points using median filters. By default, outliers are removed. See `Details` for a discussion of removal methodologies.

Usage

```
clean_jumps(
  distance_df,
  neighborhood_width = 7,
  t_cutoff = 3,
  min_median_deviation = -Inf,
  max_median_deviation = Inf,
  evaluate_tails = FALSE,
  evaluate_implosions = FALSE,
  replace_outliers = FALSE,
  return_removals = FALSE
)
```

Arguments

`distance_df`	A dataframe of linearized AVL data. Must include `trip_id_performed`, `event_timestamp`, and `distance`.
`neighborhood_width`	Optional. An integer representing the total sliding window width around each observation. Default is 7 (3 on either side).
`t_cutoff`	Optional. For Hampel filters, number of standardized MADs away to consider an outlier. Default is 3.
`min_median_deviation`	Optional. A numeric, the minimum allowed deviation of an observation from its window median, in units of distance. Default is `-Inf`.
`max_median_deviation`	Optional. A numeric, the maximum allowed deviation of an observation from its window median, in units of distance. Default is `Inf`.

`evaluate_tails` Optional. A boolean, should the beginning and ending observations, before a complete window can be formed, be evaluated? Default is `FALSE`.

`evaluate_implosions`
Optional. A boolean, should points in implosion sequences be evaluated? "Implosions" occur when more than half of a window is constant. Default is `FALSE`.

`replace_outliers`
Optional. A boolean, should points identified as outliers be replaced by their window median? Default is `FALSE`.

`return_removals`
Optional. A boolean, should the function return a dataframe of points removed and why? Default is `FALSE`.

Details

There are many different types of median filters. In general, these filters create a sliding window around each point (here, controlled by `neighborhood_width`) and treats a point based on its deviation from the median of that window. This function supports two main ways of classifying outliers based on their deviation:

- Raw deviation: `min_median_deviation` and `max_median_deviation` set bounds for an acceptable deviation between a point and the median of the window around it, in units of the input `distance` column.
- Hampel filter: Uses the median absolute deviation (MAD), the median of deviations from the median. With a conversion factor (`s = 1.48`), this is analogous to a standard error. The `t_cutoff`, then, is analogous to an acceptable window of `t` values.

Both of these can be used at the same time. If multiple criteria are set, a point will be removed if it violates any criterion.

A Hampel filter is generally considered highly robust, and is the recommended approach. There are, however, two main limitations to be aware of:

- It can struggle at the beginnings and ends of trips, before a complete window can be formed. Use `evaluate_tails` to skip these.
- When more than hald of a window has the same value, the MAD is 0 and an observation is guaranteed to be flagged as an outlier. This is known as an "implosion". As we expect noise in GPS data, even when a vehicle is standing still, this is unlikely. It can occur, however, near trip terminals, where many GPS points snap to the exact same point on the route alignment. Use `evaluate_implosions` to identify and skip points in an implosion.

Once a point has been identified as an outlier, there are two possible treatments, controlled by `replace_outliers`:

- Replacement with the window median. This is the most common approach to median filters, but is likely not appropriate for AVL data. Replacement may introduce non-monotonicities.
- Removal of the point. This is a less common approach, but may be a more sensible for this application, given that interpolating curves will be fit later in the cleaning process.

Value

The input `distance_df` with violating points removed. If `return_removals = TRUE`, a dataframe with observations removed and why.

References

Pearson, Ronald K., Yrjö Neuvo, Jaakko Astola, and Moncef Gabbouj. 2016. "Generalized Hampel Filters." EURASIP Journal on Advances in Signal Processing 2016 (1): 87. https://doi.org/10.1186/s13634-016-0383-6.

Examples

```
# Set my parameters
my_cutoff = 2.5
my_neighborhood = 9

# Get input data
lineE_no_overlaps <- new_transittraj_data("clean_overlapping_subtrips")
dim(lineE_no_overlaps)

# Run function
lineE_no_jumps <- clean_jumps(distance_df = lineE_no_overlaps,
                              neighborhood_width = my_neighborhood,
                              t_cutoff = my_cutoff)
dim(lineE_no_jumps)
head(lineE_no_jumps)
```

---

clean_overlapping_subtrips

Remove trips with multiple operators or vehicles assigned to the same trip ID

Description

In some AVL systems, multiple vehicles or operators may be logged to the same trip ID at the same time. This may be acceptable in some scenarios (e.g., a vehicle/operator tradeoff mid-trip). Other times, it may be an error, with these distinct (trip, vehicle, operator) truples running simulataneously. This function identifies both scenarios, and gives the option to remove either.

Usage

```
clean_overlapping_subtrips(
  distance_df,
  check_operator = FALSE,
  remove_single_observations = TRUE,
  remove_non_overlapping = FALSE,
  return_removals = FALSE
)
```

Arguments

distance_df	A dataframe of linearized AVL data. Must include event_timestamp, trip_id_performed, and vehicle_id. Optionally, may include operator_id.
check_operator	Optional. A boolean, should the function check for overlaps of multiple operator_ids? Default is FALSE.
remove_single_observations	Optional. A boolean, should subtrips with only one observation be removed? Default is TRUE.
remove_non_overlapping	Optional. A boolean, should trips with multiple vehicles or operators that do not overlap be removed? Default is FALSE.
return_removals	Optional. A boolean, should the function return a dataframe of trips removed and why? Default is FALSE.

Value

The input distance_df, with violating trips removed. If return_removals = TRUE, a dataframe with trip IDs removed and why.

Examples

```
# Get input data
lineE_dists <- new_transittraj_data("get_linear_distances")
dim(lineE_dists)

# Run function
lineE_no_overlaps <- clean_overlapping_subtrips(distance_df = lineE_dists)
dim(lineE_no_overlaps)
head(lineE_no_overlaps)
```

---

export_animation *Save an animation at a desired quality*

Description

This function is a helepr for gganimate's anim_save(), providing a simplified, though less feature-rich, version of these functions. Animations are saved as .gifs at the desired path. With this function, publication-quality (high-resolution and smooth) animations are possible, but take a long time to render.

Usage

```
export_animation(
  anim_object,
  path,
  duration = 30,
```

```
  fps = 10,
  width = 7.5,
  height = 5.5,
  dpi = 100
)
```

Arguments

anim_object	A gganimate object.
path	A string representing the desired path and name at which to save animation.
duration	Optional. A numeric, in seconds, representing the length of the animation. Default is 30.
fps	Optional. The frames per second of the saved animation. Default is 10.
width	Optional. The width of the exported image, in inches. Default is 7.5
height	Optional. The height of the exported image, in inches. Default is 5.5.
dpi	Optional. The resolution, in dots per inch, of the image. Default is 100.

Value

File saved to the desired directory.

Examples

```
lineE_traj <- new_transittraj_data("get_trajectory_fun")

# Set my parameters
my_features <- data.frame(name = c("Metro Center"),
                          distance = c(24556))
my_dist_range <- c(24000, 25000)

# Create `gganimate` object
anim_line <- plot_animated_line(trajectory = lineE_traj,
                                feature_distances = my_features,
                                route_color = "firebrick4",
                                label_field = "name",
                                label_alpha = 0.8,
                                label_pos = "right",
                                distance_lims = my_dist_range,
                                center_vehicles = TRUE,
                                timestep = 1)

# Create a place to store your file
my_file_name <- tempfile("my_animation", fileext = ".gif")
print(my_file_name)

# Run function: save animation locally
if (interactive()) {
  # Note: Due to long processing times, animations will only be rendered &
  #       displayed if run during an interactive session. See `example()`.
  export_animation(anim_object = anim_line,
```

```
                    path = my_file_name,
                    width = 4, height = 8, dpi = 150,
                    fps = 10, duration = 10)
}
```

`filter_by_route` *Filter GTFS to a desired route(s) and direction(s)*

Description

This function returns a new `tidygtfs` object with only the information relevant to your desired routes and directions. All fields included in the input `gtfs` will be filtered. See `Details` for more information about required files and fields

Usage

```
filter_by_route(gtfs, route_ids, dir_id = NULL)
```

Arguments

`gtfs`	A `tidygtfs` object.
`route_ids`	A vector containing the desired route ID(s).
`dir_id`	Optional. A vector containing the desired direction ID(s).

Details

The following files and fields are required for this function:

- `routes`: with `route_id` and `agency_id`
- `agency`: with `agency_id`
- `trips`: with `route_id`, `direction_id`, `shape_id`, `service_id`, and `trip_id`
- `stop_times`: with `stop_id` and `trip_id`

The following files are optional. If included, they must include the listed fields:

- `stops`: with `stop_id`
- `shapes`: with `shape_id`
- `calendar`: with `service_id`
- `calendar_dates`: with `service_id`
- `transfers`: with `trip_id` and `stop_id`
- `frequencies`: with `trip_id`
- `fare_rules`: with `route_id`
- `feed_info`

For these optional files, the function will detect whether they are present. If so, they will be filtered; if not, they will be left `NULL` in the new GTFS. If any required file or field is missing, an error will be thrown describing what is missing.

Value

A tidygtfs object containing only information relevant to the desired route and direction.

Examples

```
# Set my parameters
my_route <- "804"
my_dir <- 0

# Filter WMATA GTFS
lineE_gtfs <- filter_by_route(gtfs = lacmta_gtfs,
                              route_ids = my_route,
                              dir_id = my_dir)
summary(lineE_gtfs)
```

---

get_gtfs_service_dates

Get a dataframe of all service dates and their service IDs from a GTFS

Description

This function returns a dataframe with each date covered by a GTFS and the `service_id` run on that date. This data is extracted from the `calendar.txt` and `calendar_dates.txt` files, depending on how the GTFS is structured. See `Details` for a discussion.

Usage

```
get_gtfs_service_dates(
  gtfs,
  date_min = NULL,
  date_max = NULL,
  use_calendar_table = "calendar"
)
```

Arguments

`gtfs`	A tidygtfs object.
`date_min`	Optional. The starting (earliest possible) `Date` object for the returned dataframe. Default is `NULL`, where the earliest date in the GTFS will be used.
`date_max`	Optional. The ending (latest possible) `Date` object for the returned dataframe. Default is `NULL`, where the latest date in the GTFS will be used.
`use_calendar_table`	Optional. Should the GTFS's `calendar.txt` or `calendar_dates.txt` be used for the feasible date range? Must be `"calendar"` or `"calendar_dates"`. Default is `"calendar"`.

Details

The GTFS standard allows for two different structurings of `calendar.txt` and `calendar_dates.txt`:

- Standard service in `calendar.txt`, with exceptions in `calendar_dates.txt`. Here, `calendar.txt` will list the standard service ID by weekday (e.g., Monday, Tuesday, etc.), and `calendar_dates.txt` lists specific dates which are exceptions to this. In this scenario, `get_gtfs_service_dates()` will get enumerate all weekdays and dates in `calendar.txt`, and assign the correct `service_id` to it, depending on if the date is listed as an exception in `calendar_dates.txt`.
- All dates of service are enumerated in `calendar_dates.txt`, and `calendar.txt` is not used. In this scenario, `get_gtfs_service_dates()` will simply filter, clean, and return this table.

Use the input parameter `use_calendar_table` to control which method to use. If `use_calendar_table = "calendar"`, the former method will be used; if `use_calendar_table = "calendar_dates"`, the latter will be used. To restrict the date enumeration to only a specific window, set `date_min` and `date_max`.

This function is also intended for GTFS feeds with only one service ID per day. Some GTFS providers (including `lacmta_gtfs`) have unique `service_ids` by route, and thus service dates do not have unique `service_ids`. Consider filtering your GTFS to a single route before using this function (see `filter_by_route()`). If there are multiple service IDs on a given day, the first appearing will be returned.

Value

A dataframe with `Date` column `date` and character column `service_id`.

Examples

```
# Set parameters
study_date <- as.Date("2026-05-27")

# Get needed input data
lineE_gtfs <- filter_by_route(gtfs = lacmta_gtfs, route_ids = "804",
                              dir_id = 0)

# Run function: get service ID by day in date range
study_service_ids <- get_gtfs_service_dates(gtfs = lineE_gtfs,
                                            date_min = study_date,
                                            date_max = study_date,
                                            use_calendar_table = "calendar")
print(study_service_ids)
```

`get_gtfs_trajectory_fun` *Fit continuous trajectory interpolating curves from GTFS schedule data*

Description

This function fits a continuous vehicle trajectory function to scheduled GTFS `stop_times`. This function operates as a "function factory", returning a function (closure) which takes a timestamp and returns each trip's position. A separate curve is fit for each trip, and stored in a special trajectory object class. The default interpolating method is linear, but spline-based techniques are also supported. See `Details` for a discussion.

Usage

```
get_gtfs_trajectory_fun(
  gtfs,
  shape_geometry = NULL,
  project_crs = 4326,
  date_min = NULL,
  date_max = NULL,
  use_calendar_table = "calendar",
  agency_timezone = NULL,
  use_stop_time = "departure",
  add_stop_dwell = 0,
  add_distance_error = 0,
  interp_method = "linear",
  find_inverse_function = TRUE,
  return_group_function = TRUE,
  inv_tol = 0.01
)
```

Arguments

`gtfs`	A `tidygtfs` object.
`shape_geometry`	Optional. The SF object to project onto. Must include the field `shape_id`. See `get_shape_geometry()`. Default is `NULL`, where all shapes in `gtfs` will be used.
`project_crs`	Optional. A numeric EPSG identifer indicating the coordinate system to use for spatial calculations. Consider setting to a Euclidian projection, such as the appropriate UTM zone. Default is 4326 (WGS 84 ellipsoid).
`date_min`	Optional. The starting (earliest possible) `Date` object for the returned dataframe. Default is `NULL`, where the earliest date in the GTFS will be used.
`date_max`	Optional. The ending (latest possible) `Date` object for the returned dataframe. Default is `NULL`, where the latest date in the GTFS will be used.
`use_calendar_table`	Optional. Should the GTFS's `calendar.txt` or `calendar_dates.txt` be used for the feasible date range? Must be `"calendar"` or `"calendar_dates"`. Default is `"calendar"`.
`agency_timezone`	Optional. A timezone string (see `OlsonNames()`) indicating he appropriate timezone for the stop times. Default is `NULL`, where the timezone in `agency.txt` will be used.

use_stop_time Optional. A string, which stop time column should be used for the timepoint? Must be one of "arrival" (use arrival_time), "departure" (use departure_time), or "both", (timepoints will be created at both the stop arrival and departure). Default is "departure".

add_stop_dwell Optional. A numeric. If use_stop_time = "both", but scheduled arrival and departure times are equal (i.e., no dwell), how many seconds of dwell should be added? This will adjust forward the departure_time. Default is 0.

add_distance_error
Optional. If non-zero, each "flat" observation will be adjusted by this amount forwards, in units of input distance. Default is 0.

interp_method Optional. The type of interpolation function to be fit. Either "linear", or a spline method from stats::splinefun(). Default is "linear".

find_inverse_function
Optional. A boolean, should the numeric inverse function (time ~ distance) be calculated? Default is TRUE.

return_group_function
Optional. A boolean, should the returned trajectory object be grouped into a single function? If FALSE, will return a list (indexed by trip_id_performed) of single trajectory objects. Default is TRUE.

inv_tol Optional. A numeric in the units of input distance, the tolerance used when calculating the numeric inverse function. Default is 0.01.

Details

Stops, Dwells, and Monotonicity:

To fit an interpolating trajectory function, each observation must include distance and timestamp pairs throughout each trip. While stop_times does include a shape_dist_traveled field, this is optional and often left empty by agencies. Additionally, small distortions in spatial projections mean that projected GPS points may not align perfectly with the agency's calculated shape_dist_traveled. As such, this function uses get_stop_distances() to get the distance of each stop along each shape for each trip. Alternatively, all stops and trips can be referenced to the same spatial feature using shape_geometry. Consider setting project_crs to the same spatial projection used to linearize AVL GPS points.

The trajectory functions are fit using the times a trip is scheduled to serve each stop. There is some ambiguity here: should a stop's timestamp be when the vehicle arrives, or departs? This can be controlled using use_stop_time, set to "departure" for departure_time, "arrival" for arrival_time, or "both" to include both departure_time and arrival_time as distinct observations (i.e., distance & timestamp pairs).

Often times, however, a GTFS schedule will not have different arrival_time and departure_time values, especially if the timetable was not developed considering stop-level dwell times. In this scenario, it may be best to use only one of departure_time or arrival_time. If a dwell is desired, use add_stop_dwell to simulate a dwell time at each stop. This will increase the departure_time by the number of seconds specified.

Adding dwells opens a new consideration, however: the trajectory will no longer be strictly monotonic, as the vehicle will hold at a constant distance for some period of time. This is only a concern if find_inverse_function = TRUE, which requires strictly monotonic input data. If both dwell times and an inverse function are desired, consider setting add_distance_error > 0 to restore strict monotonicity. See make_monotonic() for more details.

Interpolating Methods:

The goal of this function is to fit a continuous function representing a GTFS trip's scheduled distance traveled as a function of time. This function supports two types of interpolating curves:

- Linear interpolation, for `interp_method = "linear"`. This will fit a simple linear function between stops.
- Spline interpolation, for `interp_method` set to any method supported by `stats::splinefun()` (i.e., `"fmm"`, `"natural"`, `"periodic"`, `"monoH.FC"`, or `"hyman"`.)

By default, `interp_method = linear`, and linear interpolation is the recommended method for schedule trajectories. This is because timetable development typically assumes a constant running speed over a corridor, so linearly connecting stop times will best reflect a trip's scheduled trajectory.

Inverse Functions:

Often times, we are concerned not with the position of a vehicle at a particular time, but the time at which a vehicle crosses a specific point in space. This can be accomplished by computing an inverse trajectory function. If `find_inverse_function = TRUE` (the default), a numeric inverse to the fit trajectory function will be found, with a tolerance controlled by `inv_tol`.

Because the inverse function is numerical, it can be found for any type of interpolating curve (linear or spline). However, the input data must be strictly monotonic for the trajectory curve to be invertible. If `find_inverse_function = TRUE`, this will be verified before proceeding (see `validate_monotonicity()`).

The Trajectory Object:

A trajectory function does not exist by itself; rather, it requires the context about the trip it describes, as well as its inverse function. As such, `get_trajectory_fun()` returns an AVL trajectory object. If `return_group_function = TRUE` (the default), the function will return a single object containing:

- A vector of `trip_id_performed`s present in `distance_df`.
- A list of fit trajectory functions (closures), one per trip, indexed by their `trip_id_performed`.
- A list of fit inverse trajectory functions (closures), one per trip, indexed by their `trip_id_performed`.
- Information about how the trajectory and inverse trajectory functions were fit, including `interp_method`, `use_speeds`, and `inv_tol`.
- A vector each for the minimum distances, maximum distances, minimum times, and maximum times of each trip. These inform the domain and range of the trajectory function and its inverse, preventing extrapolation beyond the time or distance range actually served by a trip.
- The agency's timezone (see `OlsonNames()`), as extracted from the `event_timestamp` column.

Alternatively, if `return_group_function = FALSE`, a separate trajectory object will be returned for each trip, as a list of objects indexed by their `trip_id_performed`.

More information about the trajectory object classes and how to use them is available at `vignette("articles/intro-traj`

Value

If `return_group_function = TRUE`, a grouped trajectory object. If `FALSE`, a list of single trajectory objects, indexed by their `trip_id_performed`.

Examples

```
# Set my parameters
my_crs <- 32611
my_start_date <- as.Date("2026-05-27")
my_end_date <- as.Date("2026-05-27")

# Get input data
lineE_gtfs <- new_transittraj_data("filter_by_route")
lineE_shape <- new_transittraj_data("get_shape_geometry")

# Run function: build trajectory
lineE_scheduled_traj <- get_gtfs_trajectory_fun(gtfs = lineE_gtfs,
                                                project_crs = my_crs,
                                                date_min = my_start_date,
                                                date_max = my_end_date)

# Show trajectory: summary
summary(lineE_scheduled_traj)

# Show trajectory: plot (just a handful of trips)
ordered_trips <- get_trip_extremes(lineE_scheduled_traj) %>%
   dplyr::arrange(min_time) %>%
   dplyr::pull(trip_id_performed)
plot_trajectory(trajectory = lineE_scheduled_traj,
                plot_trips = ordered_trips[20:25],
                traj_color = "indianred3")
```

---

get_linear_distances *Linearize latitude-longitude GPS points on a provided route shape*

Description

This functions projects raw AVL data, as GPS latitude-longitude points, onto a provided route geometry, returning each point's distance of that point along the shape its the beginning terminal.

Usage

```
get_linear_distances(
  avl_df,
  shape_geometry,
  clip_buffer = NULL,
  original_crs = 4326,
  project_crs = 4326
)
```

Arguments

avl_df	A dataframe of raw AVL data. Must include at least `longitude` and `latitude` columns. See `validate_tides()`.

shape_geometry	The SF object to project onto. Must be only one shape. See get_shape_geometry().
clip_buffer	Optional. The distance, in units of the chosen spatial projection, to clip the GPS points. Only points within this distance of the shape_geometry will be kept. Default is NULL, where no clip will be applied.
original_crs	Optional. A numeric EPSG identifier. If a dataframe is provided for points, this will be used to define the coordinate system of the longitude / latitude values. Default is 4326 (WGS 84 ellipsoid).
project_crs	Optional. A numeric EPSG identifer indicating the coordinate system to use for spatial calculations. Consider setting to a Euclidian projection, such as the appropriate UTM zone. Default is 4326 (WGS 84 ellipsoid).

Details

To simplify the user experience, this function takes in shape_geometry as an sf object. The internal calculations, however, are performed using the geos library, as it is substantially faster and more memory-efficient for large datasets.

A limitation of geos, however, is that it only performs planar, not ellipsoid, calculations. Given the relatively small spatial range of most local transit routes, this is reasonable if the data is appropriately projected into a Euclidean coordinate system. Consider setting project_crs to a relevant local plane when performing both get_linear_distances() and when retrieving shape geometries through get_shape_geometry().

Value

The input avl_df with latitude and longitude columns replaced by a distance column, in the units of the spatial projection used (e.g., meters if using WGS UTM).

Examples

```
# Set my parameters
my_buffer <- 50 # meters
my_crs <- 32611

# Get input data
lineE_avl <- new_transittraj_data("lineE_avl")
lineE_shape <- new_transittraj_data("get_shape_geometry")
dim(lineE_avl)

# Run function
lineE_dists <- get_linear_distances(avl_df = lineE_avl,
                                    shape_geometry = lineE_shape,
                                    clip_buffer = my_buffer,
                                    project_crs = my_crs)
dim(lineE_dists)
head(lineE_dists)
```

get_shape_geometry *Get the geometry of a route shape*

Description

This function returns an SF multilinestring of the route alignments from GTFS shapes. Similar to `tidytransit::get_geometry()`, but allows filtering by `shape_id` and projection to a new coordinate system. See `Details` for requirements on the input GTFS.

Usage

```
get_shape_geometry(gtfs, shape = NULL, project_crs = 4326)
```

Arguments

`gtfs`	A `tidygtfs` object.
`shape`	Optional. A vector of GTFS `shape_ids` to pull. Default is NULL, where all `shape_ids` in `gtfs` will be used.
`project_crs`	Optional. A numeric EPSG identifer indicating the coordinate system to use for spatial calculations. Consider setting to a Euclidian projection, such as the appropriate UTM zone. Default is 4326 (WGS 84 ellipsoid).

Details

A `shapes` file must be present in your GTFS object. This file must contain at least the following fields:

- `shape_id`
- `shape_pt_lat`
- `shape_pt_lon`
- `shape_pt_sequence`

Value

An SF multilinestring, with one multilinestring object per `shape_id`.

Examples

```
# Set my parameters
my_shape <- "804EB_RC_221121"
my_crs = 32611

# Get shape from WMATA GTFS
lineE_shape <- get_shape_geometry(gtfs = lacmta_gtfs,
                                  shape = my_shape,
                                  project_crs = my_crs)
print(lineE_shape)
```

`get_stop_distances` *Get the distances of stops along routes*

Description

This function returns the linear distance of each stop along a route shape, starting from the route's beginning terminal. Unless a `shape_geometry` is provided, stops will be project onto all `shape_id`s that serve them. If a `shape_geometry` is provided, the function will look only for stops served by that shape.

Usage

```
get_stop_distances(gtfs, shape_geometry = NULL, project_crs = 4326)
```

Arguments

`gtfs`	A `tidygtfs` object.
`shape_geometry`	Optional. The SF object to project onto. Must include the field `shape_id`. See `get_shape_geometry()`. Default is `NULL`, where all shapes in `gtfs` will be used.
`project_crs`	Optional. A numeric EPSG identifer indicating the coordinate system to use for spatial calculations. Consider setting to a Euclidian projection, such as the appropriate UTM zone. Default is 4326 (WGS 84 ellipsoid).

Value

A dataframe containing `stop_id`, the `shape_id` it was projected onto, and `distance`, in units of the spatial projection (e.g., meters if using WGS UTM).

Examples

```
# Set my parameters
my_shape <- "804EB_RC_221121"
my_crs <- 32611
my_route <- "804"
my_dir <- 0

# Get needed GTFS data
lineE_gtfs <- filter_by_route(gtfs = lacmta_gtfs, route_ids = my_route,
                              dir_id = 0)
lineE_shape <- get_shape_geometry(gtfs = lacmta_gtfs, shape = my_shape,
                                  project_crs = my_crs)

# Run stop distances function
lineE_stop_dists <- get_stop_distances(gtfs = lineE_gtfs,
                                       shape_geometry = lineE_shape,
                                       project_crs = my_crs) %>%
   dplyr::select(-c(stop_desc, stop_url, tpis_name, location_type))
head(lineE_stop_dists)
```

get_trajectory_fun *Fit continuous trajectory interpolating curves from AVL data*

Description

This function fits a continuous vehicle trajectory to cleaned AVL points (see `vignette("articles/data-workflow-la")`). This function operates as a "function factory", returning a function (closure) which takes a timestamp and returns each trip's position. A separate curve is fit for each trip, and stored in a special trajectory object class. The default interpolating method is a velocity-informed piecewise cubic interpolating polynomial, but linear interpolation and other spline-based techniques are also supported. See `Details` for a discussion.

Usage

```
get_trajectory_fun(
  distance_df,
  interp_method = "monoH.FC",
  use_speeds = TRUE,
  find_inverse_function = TRUE,
  inv_tol = 0.01,
  return_group_function = TRUE
)
```

Arguments

`distance_df`	A dataframe of linearized AVL data. Must include `trip_id_performed`, `event_timestamp`, and `distance`. If `use_speed = TRUE`, must also include `speed`.
`interp_method`	Optional. The type of interpolation function to be fit. Either `"linear"`, or a spline method from `stats::splinefun()`. Default is `"monoH.FC"`.
`use_speeds`	Optional. A boolean, should curves be constrained by observed AVL speeds? Should only be used with `interp_method = "monoH.FC"`, but `monoH.FC` does not require speeds. Default is `TRUE`.
`find_inverse_function`	Optional. A boolean, should the numeric inverse function (time ~ distance) be calculated? Default is `TRUE`.
`inv_tol`	Optional. A numeric in the units of input `distance`, the tolerance used when calculating the numeric inverse function. Default is `0.01`.
`return_group_function`	Optional. A boolean, should the returned trajectory object be grouped into a single function? If FALSE, will return a list (indexed by `trip_id_performed`) of single trajectory objects. Default is `TRUE`.

Details

Interpolating Methods:

The goal of this function is to fit a continuous function representing a vehicle's distance traveled as a function of time, for each trip. This function supports to types of interpolating curves:

- Linear interpolation, for `interp_method = "linear"`. This will fit a simple linear function, ignorant to recorded speed values.
- Spline interpolation, for `interp_method` set to any method supported by `stats::splinefun()` (i.e., `"fmm"`, `"natural"`, `"periodic"`, `"monoH.FC"`, or `"hyman"`.) Only `interp_method = "monoH.FC"` supports use of recorded speed values.

By default, `interp_method = "monoH.FC"` and `use_speeds = TRUE`. This will yield a continuous, differentiable, and invertible trajectory. If the input `distance` and speed values satisfy Fritsch-Carlson, the interpolating function is guaranteed to be montonic. See `make_monotonic()`. This is equivalent to the velocity-informed monotonic peicewise cubic Hermite interpolating polynomials with monotonic enforcement (VCHIP-ME) technique proposed by Robbennolt et al. (2026), and is our recommended approach.

Note that `use_speeds = TRUE` requires `interp_method = "monoH.FC"`, but `interp_method = "monoH.FC"` does not require `use_speeds = TRUE`. In the latter scenario, a "velocity-ignorant' Fritsch-Carlson interpolating function can be created (Huang et al., 2023). If input `distance` values are monotonic, this curve is guaranteed to be monotonic.

Inverse Functions:

Often times, we are concerned not with the position of a vehicle at a particular time, but the time at which a vehicle crosses a specific point in space. This can be accomplished by computing an inverse trajectory function. If `find_inverse_function = TRUE` (the default), a numeric inverse to the fit trajectory function will be found, with a tolerance controlled by `inv_tol`.

Because the inverse function is numerical, it can be found for any type of interpolating curve (linear or spline). However, the input data must be strictly monotonic for the trajectory curve to be invertible. If `find_inverse_function = TRUE`, this will be verified before proceeding (see `validate_monotonicity()`).

The Trajectory Object:

A trajectory function does not exist by itself; rather, it requires the context about the trip it describes, as well as its inverse function. As such, `get_trajectory_fun()` returns an AVL trajectory object. If `return_group_function = TRUE` (the default), the function will return a single object containing:

- A vector of `trip_id_performed`s present in `distance_df`.
- A list of fit trajectory functions (closures), one per trip, indexed by their `trip_id_performed`.
- A list of fit inverse trajectory functions (closures), one per trip, indexed by their `trip_id_performed`.
- Information about how the trajectory and inverse trajectory functions were fit, including `interp_method`, `use_speeds`, and `inv_tol`.
- A vector each for the minimum distances, maximum distances, minimum times, and maximum times of each trip. These inform the domain and range of the trajectory function and its inverse, preventing extrapolation beyond the time or distance range actually served by a trip.
- The agency's timezone (see `OlsonNames()`), as extracted from the `event_timestamp` column.

Alternatively, if `return_group_function = FALSE`, a separate trajectory object will be returned for each trip, as a list of objects indexed by their `trip_id_performed`.

More information about the trajectory object classes and how to use them is available at `vignette("articles/intro-traj`

Value

If `return_group_function = TRUE`, a grouped trajectory object. If `FALSE`, a list of single trajectory objects, indexed by their `trip_id_performed`.

References

Robbennolt, Jake, Sirajum Munira, and Stephen D. Boyles. 2026. "A Comparative Study of Spline-Based Trajectory Reconstruction Methods Across Varying Automatic Vehicle Location Data Densities." Paper presented at 2026 Transportation Research Board Annual Meeting, January 11. http://arxiv.org/abs/2509.00119.

Huang, Yuzhu, Awad Abdelhalim, Anson Stewart, Jinhua Zhao, and Haris Koutsopoulos. 2023. "Reconstructing Transit Vehicle Trajectory Using High-Resolution GPS Data." 2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC), September 24, 5247–53. https://doi.org/10.1109/ITSC57777.2023.10422524.

Examples

```
# Get input data
lineE_mono <- new_transittraj_data("make_monotonic")

# Run function: grouped trajectory object
lineE_traj_grouped <- get_trajectory_fun(distance_df = lineE_mono)
summary(lineE_traj_grouped)

# Run function: list of single trajectory objects
lineE_traj_singles <- get_trajectory_fun(distance_df = lineE_mono,
                                         return_group_function = FALSE)
length(lineE_traj_singles)
summary(lineE_traj_singles[[2]])
```

`get_trip_extremes` *Get the distance and time range of each trip in a trajectory object*

Description

This function extracts the time and distance ranges stored in a trajectory object and formats them into a dataframe. The dataframe can be filtered to a desired set of `trip_id_performeds`.

Usage

```
get_trip_extremes(trajectory, filter_trips = NULL)
```

Arguments

`trajectory`	A trajectory object.
`filter_trips`	Optional. A vector of `trip_id_performeds` to filter the dataframe to. At least one must of `filter_trips` must be present in `trajectory`. Default is `NULL`, where all `trip_id_performeds` in `trajectory` are returned.

Value

A dataframe with the columns `trip_id_performed`, `min_time`, `max_time`, `min_dist`, and `max_dist`.

Examples

```
# Get input data
lineE_traj <- new_transittraj_data("get_trajectory_fun")

# Run function
lineE_extremes <- get_trip_extremes(lineE_traj)
print(lineE_extremes)
```

---

group_trajectories *Group existing trajectory objects or split them apart*

Description

Trajectory objects hold the trajectory functions, and related information, from one or more trip IDs. This function groups the fit trajectories from multiple trips into one object, or splits a grouped object into many single trajectory objects, one for each trip. See `help(get_trajectory_fun)` for more information.

Usage

```
group_trajectories(trajectories, grouping)
```

Arguments

`trajectories`	A trajectory object to operate on. Can be a list of single trajectories, a list of grouped trajectories, or one grouped trajectory.
`grouping`	A character string, either `"group"` to group all trajectories in `trajectories`, or `"split"` to split `trajectories` into a list of single trajectories.

Value

If `grouping = "group"`, a group trajectory object; if `grouping = "split"`, a list of single trajectory objects.

Examples

```
# Get input data
lineE_mono <- new_transittraj_data("make_monotonic")

# Fit a list of single trajectory functions
lineE_traj_singles <- get_trajectory_fun(distance_df = lineE_mono,
                                         return_group_function = FALSE)

# Show sample singles
```

```
print(length(lineE_traj_singles))
print(lineE_traj_singles[[2]])

# Run function: group singles
lineE_traj_grouped <- group_trajectories(trajectories = lineE_traj_singles,
                                         grouping = "group")
summary(lineE_traj_grouped)

# Run function: split apart again
lineE_traj_singles_2 <- group_trajectories(trajectories = lineE_traj_grouped,
                                           grouping = "split")
print(length(lineE_traj_singles_2))
print(lineE_traj_singles_2[[2]])
```

`lacmta_avl` *LA Metro AVL Data*

Description

This dataset contains TIDES-formatted automatic vehicle location (AVL) data from the Los Angeles County Metropolitan Transportation Authority (LACMTA), or Metro. Pings for all Lines A and E trips starting between 6:00 and 8:00 AM on May 27, 2026 are included. The data was accessed via Caltrans's open-source bucket of TIDES `vehicle_locations` tables. This dataset is inteded to be used alongside the static GTFS feed provided in `lacmta_gtfs`.

Usage

```
lacmta_avl
```

Format

`lacmta_avl`**:**

A dataframe with 14,179 rows and 11 columns.

location_ping_id A unique ID for each row.

service_date The date of the trip's beginning.

trip_id_performed Trip IDs, matching those in GTFS.

latitude, longitude The GPS ping longitude and latitude.

speed The recorded speed, in meters per second.

vehicle_id An ID corresponding to each vehicle.

event_timestamp POSIXct time objects, including the day, time, and local timezone.

direction_id Direction IDs, matching those in GTFS. For Line A, `0` is northbound and `1` is southbound; for Line E, `0` is eastbound and `1` is westbound.

shape_id Shape IDs, matching those in GTFS. Each route and direction has one shape ID.

route_id Route IDs, matching those in GTFS. `"801"` is Line A, and `"804"` is Line E.

Details

The dataset contains two light rail routes, with two directions for each:

- Line A: Pomona North to Downtown Long Beach
- Line E: Downtown Santa Monica to Atlantic

Source

https://tides.dds.dot.ca.gov/

Examples

```
# Print the header
head(lacmta_avl)

# Filter the data
lineE_avl <- lacmta_avl %>%
    dplyr::filter((route_id == "804") & (direction_id == 0))
print(unique(lineE_avl$shape_id))

# Use in the AVL cleaning workflow
lineE_shape <- get_shape_geometry(gtfs = lacmta_gtfs,
                                  shape = "804EB_RC_221121",
                                  project_crs = 32611)
lineE_dists <- get_linear_distances(avl_df = lineE_avl,
                                    shape_geometry = lineE_shape,
                                    clip_buffer = 50,
                                    project_crs = 32611)
head(lineE_dists)
```

---

lacmta_gtfs *LA Metro GTFS*

Description

This dataset is a portion of the rail General Transit Feed Specification (GTFS) from the Los Angeles County Metropolitan Transportation Authority (LACMTA), or Metro. This feed version was first published on May 27, 2026, and was valid through May 28, 2026. This dataset is intended to be used alongside the TIDES AVL data provided in `lacmta_avl`.

Usage

```
lacmta_gtfs
```

Format

`lacmta_gtfs`:

A `tidytransit` `tidygtfs` object (list) with 8 files.

agency The GTFS `agency.txt` file.

routes The GTFS `routes.txt` file.

trips The GTFS `trips.txt` file.

stop_times The GTFS `stop_times.txt` file.

stops The GTFS `stops.txt` file.

shapes The GTFS `shapes.txt` file.

calendar The GTFS `calendar.txt` file.

calendar_dates The GTFS `calendar_dates.txt` file.

fare_rules The GTFS `fare_rules.txt` file.

Details

The GTFS feed has been filtered to two light rail routes, with two directions for each, on one service date (May 27, 2026):

- Line A: Pomona North to Downtown Long Beach
- Line E: Downtown Santa Monica to Atlantic

Source

https://www.transit.land/feeds/f-9q5-metro~losangeles~rail

Examples

```
# Print the tidytransit summary
summary(lacmta_gtfs)

# Filter by route & direction
my_route <- "804"
my_dir <- 0
lineE_gtfs <- filter_by_route(gtfs = lacmta_gtfs,
                              route_ids = my_route,
                              dir_id = my_dir)
summary(lineE_gtfs)

# Extract route alignments
lineE_shapes <- get_shape_geometry(gtfs = lineE_gtfs)
print(lineE_shapes)
```

`make_monotonic` *Correct distance observations, and optionally speeds, to be weakly or strictly monotonic*

Description

Due to error in GPS position and speed measurements, raw AVL data is often not monotonic, creating difficulties for advanced analyses. This function presents a variety of options to correct data, resulting in distance values, and optionally speeds, which form a strictly or weakly monotonic curve. See `Details` for more information.

Usage

```
make_monotonic(
  distance_df,
  correct_speed = FALSE,
  add_distance_error = 0,
  direction = "bidirectional",
  return_changes = FALSE
)
```

Arguments

`distance_df`	A dataframe of linearized AVL data. Must include `trip_id_performed`, `event_timestamp`, and `distance`. If `correct_speed = TRUE`, must also include `speed`.
`correct_speed`	Optional. A boolean, should speeds be corrected to meet adjusted distances and Fritsch-Carlson conditions? Default is `FALSE`.
`add_distance_error`	Optional. If non-zero, each "flat" observation will be adjusted by this amount forwards, in units of input `distance`. Default is `0`.
`direction`	Optional. The step direction for monotonic enforcement. One of `"forward"`, `"backward"`, or `"bidirectional"`. Default is `"bidirectional"`.
`return_changes`	Optional. Should a dataframe of each observation changed be returned? Default is `FALSE`.

Details

There are two primary types of monotonicity:

- Weak, allowing the function to be increasing or constant
- Strict, forcing the function to always increase and never stand still.

While we know that, in practice, transit vehicles rarely move backwards (i.e., reverse) along their route, random noise in GPS observations will introduce non-monotonicities (i.e., drifting backwards). Weak monotonicity most accurately describes real transit vehicle trajectories: we expect

the vehicle to either move forwards, or stand still at a stop. However, strict monotonicity is a convenient mathematical property that allows us to find the inverse trajectory (i.e., retrieve time as a function of distance).

There are three ways in which monotonicity can be enforced upon a function's position:

- Forward-stepping: Beginning at the beginning of a trip, iterate forwards in time through distance observations. If the distance ever drifts backward, it will be "pulled up" to the cumulative maximum distance observed by that point.
- Backward-stepping: Beginning at the end of a trip, iterate backwards in time through distance observations. If the distance ever drifts forward, it will be "pushed down" to the cumulative minimum distance observed by that point.
- Bidirectional: Because noise in the distance observations occurs randomly in both directions, during stationary periods, forward-stepping will bias the trajectory positively (i.e., the trajectory is ahead of the actual vehicle position), and backward-stepping will bias the trajectory negatively (i.e., the trajectory is behind the actual vehicle position). Additionally, forward-stepping can disrupt the acceleration cycle at the end of a stationary period, and backward-stepping can disrupt the deceleration cycle at the beginning of a stationary period. The bidirectional approach remedies this by performing both directions of correction on the original data, then averaging the two outcomes.

Choose between the two monotonicity types by setting `add_distance_error`. If `add_distance_error = 0` (the default), a weakly monotonic trajectory is returned; otherwise, the trajectory will be strictly monotonic, with a perturbation of `add_distance_error` added (for forward-stepping) or subtracted (for backward-stepping) in flat regions. Choose between the two step directions using `direction`.

In addition to distance corrections, some applications (e.g., fitting a velocity-informed interpolation spline) require speeds to satisfy certain monotonic conditions. If `correct_speed = TRUE`, the following corrections will be made:

- For strict monotonicity (`add_distance_error > 0`), speeds must be non-zero. At each point where the recorded `speed == 0`, the speed will be replaced by `add_distance_error` divided by the time between that point and the previous point.
- For both strict and weak monotonicity, speeds will be adjusted to meet the Fritsch-Carlson (1980) constraints. Often, only a handful of input `speed` values will be adjusted.

If recorded speed values are not present, set `correct_speed = FALSE`. However, if you are interested in later fitting a velocity-informed interpolating curve, such as Fritsch-Carlson's piecewise cubic polynomials, consider setting `correct_speed = TRUE` to guarantee a monotonic interpolating curve.

After using this function to perform corrections, use `validate_monotonicity()` to check if weak, strict, and Fritsch-Carlson speed conditions are met.

Value

The input `distance_df` with distances and speeds adjusted. If `return_changes = TRUE`, a dataframe with observations changed.

References

Fritsch, F. N., and R. E. Carlson. 1980. "Monotone Piecewise Cubic Interpolation." SIAM Journal on Numerical Analysis. https://doi.org/10.1137/0717021.

Robbennolt, Jake, Sirajum Munira, and Stephen D. Boyles. 2026. "A Comparative Study of Spline-Based Trajectory Reconstruction Methods Across Varying Automatic Vehicle Location Data Densities." Paper presented at 2026 Transportation Research Board Annual Meeting, January 11. http://arxiv.org/abs/2509.00119.

Examples

```
# Set my parameters
my_dist_err = 0.001

# Get input data
lineE_trimmed <- new_transittraj_data("trim_trips")

# Run function
lineE_mono <- make_monotonic(distance_df = lineE_trimmed,
                             add_distance_error = my_dist_err,
                             correct_speed = TRUE)
head(lineE_mono)
```

---

new_transittraj_data *Retrieve an object from a particular step of* transittraj*'s workflow*

Description

This function runs transittraj's AVL cleaning and trajectory reconstruction workflow up until a certain point (as defined by func_output), then returns the object at that point. A subset of the lacmta_avl dataset is used. This is primarily intended for use in testing and examples. The workflow applied here is the same as what is in vignette("articles/data-workflow-la").

Usage

```
new_transittraj_data(func_output = NULL)
```

Arguments

func_output	The transittraj function to return an output for. Should be a string corresponding to the function name. Default is NULL, where a vector of allowed inputs will be returned.

Value

The object returned by the specified function.

Examples

```
# Get AVL data after projection onto route
lineE_dists <- new_transittraj_data("get_linear_distances")
head(lineE_dists)

# Get a full, fit trajectory
lineE_traj <- new_transittraj_data("get_trajectory_fun")
summary(lineE_traj)
```

`plot.avltrajectory_group` *Quickly plot an AVL trajectory*

Description

This function generates a quick plot of a single or grouped trajectory object. Using the trajectory function, the entire trajectory will be plotted at a temporal resolution of 10 seconds. For grouped trajectories, a maximum of 50 trips will be plotted. For more control over plotting and formatting, see `plot_trajectory()`.

Usage

```
## S3 method for class 'avltrajectory_group'
plot(x, ...)

## S3 method for class 'avltrajectory_single'
plot(x, ...)
```

Arguments

`x`	A trajectory object.
`...`	Other parameters (not used).

Value

A `ggplot2` object.

Examples

```
# Get input data
lineE_traj_grouped <- new_transittraj_data("get_trajectory_fun")
lineE_traj_singles <- new_transittraj_data("get_trajectory_fun_single")

# Plot: Grouped trajectory object
plot(lineE_traj_grouped)

# Plot: Single trajectory object
plot(lineE_traj_singles[[2]])
```

plot_animated_line *Animate vehicle trajectory or AVL data*

Description

These functions use the input trajectory object or TIDES AVL data to animate vehicles progressing along their routes. This can be visualized in two ways:

- `plot_animated_line()` simplifies the route alignment into a single straight line and shows the vehicles moving down this line.
- `plot_animated_map()` plots the full route's alignment and shows the vehicles moving through space.

Both functions allow the plotting of spatial features and labels for these features. A `gganimate` object is returned, which can be further modified and customized as desired.

Usage

```
plot_animated_line(
  trajectory = NULL,
  distance_df = NULL,
  plot_trips = NULL,
  timestep = 5,
  distance_lims = NULL,
  center_vehicles = FALSE,
  feature_distances = NULL,
  transition_style = "linear",
  convert_to_timezone = TRUE,
  route_color = "coral",
  route_width = 3,
  route_alpha = 1,
  feature_shape = 21,
  feature_outline = "black",
  feature_fill = "white",
  feature_size = 2,
  feature_stroke = 1.25,
  feature_alpha = 1,
  feature_legend = NULL,
  veh_shape = 23,
  veh_outline = "grey30",
  veh_fill = "white",
  veh_size = 3,
  veh_stroke = 2,
  veh_alpha = 0.8,
  veh_legend = NULL,
  label_field = NULL,
  label_size = 3,
```

```
  label_alpha = 0.6,
  label_pos = "left"
)

plot_animated_map(
  shape_geometry,
  trajectory = NULL,
  distance_df = NULL,
  plot_trips = NULL,
  timestep = 5,
  distance_lims = NULL,
  center_vehicles = FALSE,
  feature_distances = NULL,
  background = "osm",
  background_zoom = 0,
  bbox_expand = NULL,
  transition_style = "linear",
  convert_to_timezone = TRUE,
  route_color = "coral",
  route_width = 3,
  route_alpha = 1,
  feature_shape = 21,
  feature_outline = "black",
  feature_fill = "white",
  feature_size = 2,
  feature_stroke = 1.25,
  feature_alpha = 1,
  feature_legend = NULL,
  veh_shape = 23,
  veh_outline = "grey30",
  veh_fill = "white",
  veh_size = 3,
  veh_stroke = 2,
  veh_alpha = 0.8,
  veh_legend = NULL,
  label_field = NULL,
  label_size = 3,
  label_alpha = 0.6,
  label_pos = "out"
)
```

Arguments

`trajectory`	Optional. A trajectory object, either a single trajectory or grouped trajectory. If provided, `distance_df` must not be provided. Default is `NULL`.
`distance_df`	Optional. A dataframe of time and distance points. Must include at least `event_timestamp`, `distance`, and `trip_id_performed`. If provided, `trajectory` must not be provided. Default is `NULL`.

plot_trips Optional. A vector of trip_id_performeds to plot. Default is NULL, which will plot all trips provided in the trajectory or distance_df.

timestep Optional. If trajectory is provided, the time interval, in seconds, between interpolated observations to plot. Default is 5.

distance_lims Optional. A vector with (minimum, maximum) distance values to plot.

center_vehicles
Optional. A boolean, should all vehicle points be centered to start at the same time (0 seconds)? Default is FALSE.

feature_distances
Optional. A dataframe with at least numeric distance for features. Default is NULL.

transition_style
Optional. A gganimate transition_style, specifying how the animation transitions from point to point. See gganimate::ease_aes(). Default is "linear".

convert_to_timezone
Optional. Should numeric epoch times be converted back to readable hour-minute-second time values, using the agency timezone? Default is TRUE.

route_color Optional. A color string for the color of the route alignment. Default is "coral".

route_width Optional. A numeric, the linewidth of the route alignment. Default is 3.

route_alpha Optional. A numeric, the opacity of the route alignment. Default is 1.

feature_shape Optional. A numeric specifying the ggplot2 point shape, or a dataframe mapping an attribute in feature_distances to a shape. Must contain column shape. Default is 21 (circle).

feature_outline
Optional. A color string, or a dataframe mapping an attribute in feature_distances to a color. Must contain column outline. Default is "black".

feature_fill Optional. A color string, the inside fill of feature points. Default is "white".

feature_size Optional. A numeric, the size of the feature point. Default is 2.

feature_stroke Optional. A numeric, the linewidth of the feature point outline. Default is 1.25.

feature_alpha Optional. A numeric, the opacity of the feature point. Default is 1.

feature_legend Optional. A boolean, should a legend be shown for feature formatting? Default is NULL, where a legend will only appear if the shape or outline format is mapped to.

veh_shape Optional. A numeric specifying the ggplot2 point shape, or a dataframe mapping an attribute in distance_df or trajectory to a shape. Must contain column shape. Default is 23 (diamond).

veh_outline Optional. A color string, or a dataframe mapping an attribute in distance_df or trajectory to a color. Must contain column outline. Default is "grey30".

veh_fill Optional. A color string, the inside fill of the vehicle point. Default is "white".

veh_size Optional. A numeric, the size of the vehicle point. Default is 3.

veh_stroke Optional. A numeric, the linewidth of the vehicle point outline. Default is 2.

veh_alpha Optional. A numeric, the opacity of the vehicle point. Default is 0.8.

veh_legend	Optional. A boolean, should a legend be shown for vehicle formatting? Default is NULL, where a legend will only appear if the shape or outline format is mapped to.
label_field	Optional. A string specifying the column in feature_distances with which to label the feature lines. Default is NULL, where no labels will be plotted.
label_size	Optional. The font size of the feature labels. Default is 3.
label_alpha	Optional. The opacity of the feature labels. Default is 0.6.
label_pos	Optional. A string specifying the label position on the graph. Options include: • plot_animated_line(): "left" or "right". Default is "left". • plot_animated_map(): cardinal or intermediate compass direction (e.g., "N", "SW", etc.), or "in"/"out". Default is "out".
shape_geometry	An SF object representing the route alignment. See get_shape_geometry().
background	Optional. The OSM background (basemap) for the animation. See rosm::osm.image(). Default is "cartolight".
background_zoom	Optional. The zoom, relative to the "correct" level, for the background basemap. Default is 0.
bbox_expand	Optional. The distance by which to expand the plotting window in both directions. Default is NULL, which will expand the window by 0.05% of the larger dimension (or 0.0025% if distance_lims is provided).

Details

Input Trajectory Data:

There are two ways to provide data for these plotting functions:

- A single or grouped trajectory object. This will use the direct trajectory function at a resolution controlled by timestep. This is simplest, and looks best when zooming in using distance_lims. The only attribute that can be mapped to if using a trajectory is trip_id_performed.
- A distance_df of TIDES AVL data. This will use the distance and time point pairs for plotting, and draw linearly between them. This will look similar to a plot using trajectory when zoomed out. It is most useful if you want to map formatting to attributes other than trip_id_performed, such as a vehicle or operator ID. If starting with a trajectory, but the additional control over formatting is desired, consider using predict() to generate distance and time points to plot, then joining the desired attributes to the trip_id_performed column.

Note that only one of trajectory and distance_df can be used. If both (or neither) are provided, an error will be thrown.

Features and Labels:

Often it is useful to plot the features of a route, such as its stops/stations or the traffic signals it passes through. Use feature_distances to provide information about spatial features to plot. Each row in feature_distances should include at least a distance column. Each of these rows will be plotted as a point on the route line.

These features can also be labeled. Set label_name to a character string corresponding to a field in feature_distances to generate labels with this field as their text. The color of the label will automatically match that of the feature they describe. The label placement is controlled by label_pos, which has the following options available:

- `plot_animated_line()`: Either `"left"` or `"right"` of the route line. The y-value will be that of the feature it describes.
- `plot_animated_map()`. A cardinal or intermediate compass direction (`"N"`, `"SE"`, etc.) relative to the feature point. Or, `"in"`/`"out"`, relative to the center of the plot.

Note that for `plot_animated_map()` the `feature_distances` must still be linear distances, not a spatial datatype. To retrieve distance values for spatial features, see `get_stop_distances()` and `project_onto_route()`.

Formatting Options:

Once a layer is created on a `ggplot2` object, it is difficult to change its formatting. As such, this function attempts to provide as much flexibility in formatting its layers as possible. The resulting plot includes four layers in the following order:

- Route line, controlled by `route_color`, `route_width`, and `route_alpha`.
- Features, controlled by `feature_fill`, `feature_outline`, `feature_shape`, `feature_size`, `feature_alpha`, and `feature_stroke`.
- Labels, controlled by `label_size`, `label_alpha`, and `label_pos`.
- Vehicles, controlled by `veh_fill`, `veh_outline`, `veh_shape`, `veh_size`, `veh_alpha`, and `veh_stroke`.

All of these formats can be controlled by inputting a single string or numeric. The following attributes can also be modified using a dataframe, mapping them to attributes of the layer:

- `veh_shape` and `feature_shape`: A dataframe with one column named `shape`, and another column sharing a name with a column in `distance_df` or `feature_distances` (or, if using `trajectory`, a column named `trip_id_performed`). The values in `shape` should be valid `ggplot2` point shapes, and the values in the mapping column should match the values in feature or trip column.
- `veh_outline` and `feature_outline`: A dataframe with one column named `outline`, and another column sharing a name with a column in `distance_df` or `feature_distances` (or, if using `trajectory`, a column named `trip_id_performed`). The values in `outline` should be valid color strings, and the values in the mapping column should match the values in feature or trip column.

Note that if inputting `trajectory`, instead of `distance_df`, `veh_shape` and `veh_outline` can only be mapped to `trip_id_performed`. If using `distance_df`, they may be mapped to any column in `distance_df` (e.g., vehicle or operator IDs).

Basemaps:

The function `plot_animated_map()` has one additional layer to format: the basemap beneath the route alignment. OpenStreetMaps basemaps are used here. See a full list of available basemaps using `rosm::osm.types()`.

In addition to the map itself, the zoom level on the map can be adjusted using `background_zoom`. This will describe a zoom level relative to the "correct" level of your bounding box (i.e., what you what see if you looked at that data in an online mapping platform). Setting to a negative value (zooming out) will substantially speed up rendering, but will give a much lower resolution may.

Finally, the bounding box of the basemap can also be set. The bounding box is defined relative to the spatial range of `trajectory` or `distance_df`. The default is expansion is 0.05% (0.0025% for `distance_lims != NULL`) of the larger dimension (northing or easting) of the vehicle location bounding box. To customize, det `bbox_expand` to some numeric in the distance units of the `shape_geometry`'s spatial projection (e.g., meters if using a WGS UTM projection).

Value

A gganimate object.

Examples

```
# Get input data
lineE_traj <- new_transittraj_data("get_trajectory_fun")
lineE_shape <- new_transittraj_data("get_shape_geometry")

# Set my parameters
my_features <- data.frame(name = c("Metro Center"),
                          distance = c(24556))
my_dist_range <- c(24000, 25000)

# Run function: Line animation
anim_line <- plot_animated_line(trajectory = lineE_traj,
                                feature_distances = my_features,
                                route_color = "firebrick4",
                                label_field = "name",
                                label_alpha = 0.8,
                                label_pos = "right",
                                distance_lims = my_dist_range,
                                center_vehicles = TRUE,
                                timestep = 1)

# Run function: Map animation
anim_map <- plot_animated_map(trajectory = lineE_traj,
                              shape_geometry = lineE_shape,
                              feature_distances = my_features,
                              route_color = "firebrick4",
                              label_field = "name",
                              label_alpha = 0.8,
                              distance_lims = my_dist_range,
                              center_vehicles = TRUE,
                              timestep = 1,
                              bbox_expand = 40)

if (interactive()) {
  # Note: Due to long processing times, animations will only be rendered &
  #       displayed if run during an interactive session. See `example()`.
  anim_line
  anim_map
}
```

`plot_interactive_gtfs` *Generate a Leaflet viewer of GTFS routes and stops*

Description

This function generates a simple Leaflet-based interactive map viewer of a GTFS. This function is intended for quick and easy visualization of a GTFS feed. As such, formatting options are relatively limited.

Usage

```
plot_interactive_gtfs(
  gtfs,
  background = "Esri.WorldGrayCanvas",
  color_palette = "Dark2"
)
```

Arguments

`gtfs`	A tidytransit GTFS object.
`background`	Optional. A string for the background of the transit map, from Leaflet's provider library (see `leaflet::providers$`). Default is Esri's light gray canvas (`"Esri.WorldGrayCanvas"`).
`color_palette`	Optional. A string for the Leaflet color palette to color routes. If `"gtfs"`, will use color codes in the GTFS `routes` file. Default is `"Dark2"`.

Details

Route Shapes and Stops:

The primary goal of this function is to visualize and explore each GTFS shape, including its associated `route_id` and `direction_id`. This function will plot all shapes and stops present in the input `gtfs`. To plot only a specific route or direction, first filter the feed using `filter_by_route()`.

Routes have both pop-ups and hover labels. The hover label shows the shapes's `route_id` (from the `trips` file). The pop-up will show the `route_id`, `direction_id`, and `shape_id`.

Stops also have both pop-ups and hover labels. The hover label will show the point's `stop_id` (from the `stops` file). The pop-up will show the `stop_name` and `stop_id`.

Formatting:

Two formatting options are available through this function: basemaps and route color palettes.

The `background` parameter allows you to customize the background basemap. Esri's light grey canvas is the default, as it is excellent for providing geographic context while still allowing the routes to stand out. To see the available options, type `leaflet::providers$` into your console.

The route colors can be customized in two different ways:

- Using the `gtfs`'s colors. Typically, the `routes` file in a GTFS feed will contain a field `route_color`; this is the color you see in most public-facing mapping/navigation applications (e.g., Google Maps, Transit App, etc.). If this is present in the input `gtfs` feed, setting `color_palette = "gtfs"` will use this field to color each shape.
- Using a named color palette. Without `gtfs` colors, this function assigns colors categorically (using `leaflet::colorFactor()`). To set the palette, input a string corresponding to a palette name from `RColorBrewer`, a palette name from `viridis`, a vector of color names (with the same length as the number of shapes), or some other color function. See Leaflet's colors vignette for more information.

Value

A Leaftlet object.

Examples

```
plot_interactive_gtfs(gtfs = lacmta_gtfs,
                      color_palette = "gtfs")
```

`plot_trajectory` *Plot vehicle trajectories or AVL data*

Description

This function uses the input trajectory object or TIDES AVL data to draw a trajectory plot (i.e., linear distance versus time) for each trip. This function allows for the plotting of spatial features and labels for these features. A `ggplot2` object is returned, which can be further modified and customized as desired.

Usage

```
plot_trajectory(
  trajectory = NULL,
  distance_df = NULL,
  plot_trips = NULL,
  timestep = 5,
  distance_lims = NULL,
  center_trajectories = FALSE,
  feature_distances = NULL,
  convert_to_timezone = TRUE,
  traj_color = "coral",
  traj_type = "solid",
  traj_width = 1,
  traj_alpha = 1,
  traj_legend = NULL,
  feature_color = "grey30",
  feature_type = "dashed",
  feature_width = 0.8,
  feature_alpha = 0.8,
  feature_legend = NULL,
  label_field = NULL,
  label_size = 3,
  label_alpha = 0.6,
  label_pos = "left"
)
```

Arguments

trajectory	Optional. A trajectory object, either a single trajectory or grouped trajectory. If provided, distance_df must not be provided. Default is NULL.
distance_df	Optional. A dataframe of time and distance points. Must include at least event_timestamp, distance, and trip_id_performed. If provided, trajectory must not be provided. Default is NULL.
plot_trips	Optional. A vector of trip_id_performeds to plot. Default is NULL, which will plot all trips provided in the trajectory or distance_df.
timestep	Optional. If trajectory is provided, the time interval, in seconds, between interpolated observations to plot. Default is 5.
distance_lims	Optional. A vector with (minimum, maximum) distance values to plot.
center_trajectories	Optional. A boolean, should all trajectories be centered to start at the same time (0 seconds)? Default is FALSE.
feature_distances	Optional. A dataframe with at least numeric distance for features. Default is NULL.
convert_to_timezone	Optional. Should numeric epoch times be converted back to readable hour-minute-second time values, using the agency timezone? Default is TRUE.
traj_color	Optional. A color string, or a dataframe mapping an attribute in distance_df or trip_id_performed in trajectory to a color. Must contain column color. Default is "coral".
traj_type	Optional. A string specifying the ggplot2 linetype, or a dataframe mapping an attribute in distance_df or trip_id_performed in trajectory to a linetype. Must contain column linetype. Default is "solid".
traj_width	Optional. A numeric, the width of the trajectory line. Default is 1.
traj_alpha	Optional. A numeric, the opacity of the trajectory line. Default is 1.
traj_legend	Optional. A boolean, should a legend be shown for trajectory formatting? Default is NULL, where a legend will only appear if the linetype or color format is mapped to.
feature_color	Optional. A color string, or a dataframe mapping an attribute in feature_distances to a color. Must contain column color. Default is grey30.
feature_type	Optional. A string specifying the ggplot2 linetype, or a dataframe mapping an attribute in feature_distances to a linetype. Must contain column linetype. Default is "dashed".
feature_width	Optional. A numeric, the width of the feature line. Default is 0.8.
feature_alpha	Optional. A numeric, the opacity of the feature point. Default is 1.
feature_legend	Optional. A boolean, should a legend be shown for feature formatting? Default is NULL, where a legend will only appear if the linetype or color format is mapped to.
label_field	Optional. A string specifying the column in feature_distances with which to label the feature lines. Default is NULL, where no labels will be plotted.

label_size	Optional. The font size of the feature labels. Default is 3.
label_alpha	Optional. The opacity of the feature labels. Default is 0.6.
label_pos	Optional. A string specifying the label position on the graph. Must be either "left" or "right". Default is "left".

Details

Input Trajectory Data:

There are two ways to provide data to this function:

- A single or grouped trajectory object. This will use the direct trajectory function at a resolution controlled by timestep. This is simplest, and looks best when zooming in using distance_lims. The only attribute that can be mapped to if using a trajectory is trip_id_performed.
- A distance_df of TIDES AVL data. This will use the distance and time point pairs for plotting, and draw linearly between them. This will look similar to a plot using trajectory when zoomed out. It is most useful if you want to map formatting to attributes other than trip_id_performed, such as a vehicle or operator ID. If starting with a trajectory, but the additional control over formatting is desired, consider using predict() to generate distance and time points to plot, then joining the desired attributes to the trip_id_performed column.

Note that only one of trajectory and distance_df can be used. If both (or neither) are provided, an error will be thrown.

Features and Labels:

Often it is useful to plot the features of a route, such as its stops/stations or the traffic signals it passes through. Use feature_distances to provide information about spatial features to plot. Each row in feature_distances should include at least a distance column. Each of these rows will be plotted as a horizontal line across the graph.

These features can also be labeled. Set label_name to a character string corresponding to a field in feature_distances to generate labels with this field as their text. The color of the label will automatically match that of the feature they describe. The label placement is controlled by label_pos, which can be set to "left" or "right".

Formatting Options:

Once a layer is created on a ggplot2 object, it is difficult to change its formatting. As such, this function attempts to provide as much flexibility in formatting its layers as possible. The resulting plot includes three layers:

- Vehicle trajectories, controlled by traj_color, traj_type, and traj_width, and traj_alpha.
- Features, controlled by feature_color, feature_type, feature_width, and feature_alpha.
- Labels, controlled by label_size, label_alpha, and label_pos.

All of these formats can be controlled by inputting a single string or numeric. The following attributes can also be modified using a dataframe, mapping them to attributes of the layer:

- traj_type and feature_type: A dataframe with one column named linetype, and another column sharing a name with a column in distance_df or feature_distances (or, if using trajectory, a column named trip_id_performed). The values in linetype should be valid ggplot2 linetypes, and the values in the mapping column should match the values in feature or trip column.

- traj_color and feature_color: A dataframe with one column named color, and another column sharing a name with a column in distance_df or feature_distances (or, if using trajectory, a column named trip_id_performed). The values in color should be valid color strings, and the values in the mapping column should match the values in feature or trip column.

Note that if inputting trajectory, instead of distance_df, traj_color and traj_type can only be mapped to trip_id_performed. If using distance_df, they may be mapped to any column in distance_df (e.g., vehicle or operator IDs).

Value

A ggplot2 object.

Examples

```
# Get input data
lineE_traj <- new_transittraj_data("get_trajectory_fun")

# Set my parameters
my_features <- data.frame(name = c("Metro Center"),
                          distance = c(24556))
my_dist_range <- c(24000, 25000)

# Run function
plot_trajectory(trajectory = lineE_traj,
                feature_distances = my_features,
                label_field = "name",
                label_alpha = 0.8,
                distance_lims = my_dist_range,
                traj_color = "indianred3",
                center_trajectories = TRUE,
                timestep = 1)
```

---

predict.avltrajectory_group

Interpolate time or distance points using AVL trajectories

Description

This function uses a fit interpolating curve stored in a grouped or single trajectory object to find new points along each trip's trajectory. Depending on whether new_times or new_distances is provided, the function will utilize the direct or inverse trajectory function.

Usage

```
## S3 method for class 'avltrajectory_group'
predict(
  object,
```

```
  new_times = NULL,
  new_distances = NULL,
  distance_lims = NULL,
  timestep = NULL,
  deriv = 0,
  trips = NULL,
  ...
)
```

Arguments

`object`	The single or grouped trajectory object.
`new_times`	Optional. A vector of numeric timepoints, or a dataframe with at least the column `event_timestamp` of new timepoints to interpolate at. May also contain the column `trip_id_performed`, which will interpolate distances at each trip and time row pair. Default is `NULL`.
`new_distances`	Optional. A vector of numeric distances, or a dataframe with at least the column `distance` of new distances to interpolate at. May also contain the column `trip_id_performed`, which will interpolate times at each trip and distance row pair. Default is `NULL`.
`distance_lims`	Optional. A vector of `(minimum, maximum)` distance bounds over which to interpolate at a given timestep. If provided, `timestep` must also be provided. Default is `NULL`.
`timestep`	Optional. A single numeric indicating the time interval between successive interpolating steps when defining `distance_lims`. If provided, `distance_lims` must also be provided. Default is `NULL`.
`deriv`	Optional. The vector of numeric derivative degrees to calculate at. May only be set if `new_times` or `distance_lims`/`timestep` is provided, and not if `new_distances` is provided. Default is `0` (i.e., position).
`trips`	Optional. A vector of `trip_id_performeds` to interpolate for. Default is `NULL`, which will use all trips found in the trajectory object (or, if include, in the `trip_id_performed` column of `new_times` or `new_distances`).
`...`	Other parameters (not used).

Details

This function is the recommended way to use a fit trajectory function. It has a few key features:

Interpolation:

There are three ways to interpolate: finding distance from times (direct trajectory function), times from distance (inverse trajectory function), or timesteps over a distance range (both inverse and direct trajectory function). For the former two, either a vector or dataframe of `new_times` or `new_distances` may be provided. If a dataframe is provided, it must contain the column `event_timestamp` or `distance`, and all additional columns will be preserved through the interpolation.

Distances from Times:
If `new_times` is provided, the function will find the `distance` of each trip at each point in time. If a dataframe is provided, it must contain the column `event_timestamp`. This will use the trajectory's direct function. When using `new_times`, a `deriv` value can also be set greater than 0. See below for a more detailed discussion.

Times from Distances:
If `new_distances` is provided, the function will find the `event_timestamp` of each trip at each point in space. If a dataframe is provided, it must contain the column `distance`. This will use the trajectory's inverse function. When using `new_distances`, a `deriv` value cannot be set greater than 0. See below for a more detailed discussion.

Time & Distance Pairs from Distance Bounds:
Oftentimes, you may want to interpolate by small timesteps over a defined region of space. This can be done by setting `distance_lims` and `timestep`. The function will use the trajectory's inverse function to find each trip's entrance and exit time through `distance_lims`, then create a sequence between these entrance and exit times with a step of `timestep`. Finally, the trajectory's direct function is used to find the distance at each of these timepoints. A `deriv` value can also be set greater than 0 for the final direct interpolation.
If you have a well-defined region of space, this approach allows you to interpolate vehicle positions at a very tight timescale over a large number of trips efficiently. You could alternatively use `new_times` to interpolate over the entire time range of all trips (which wouldn't require an inverse function), though this may require orders of magnitude more points and would be substantially less efficient.

Finding Derivatives:

Depending on the `interp_method` used when fitting the trajectory object, a derivative may be able to be found:

- `interp_method = "linear"`: This will not allow derivatives. This is because, at each observation, the piecewise linear function is not differentiable.
- `interp_method` is a spline from `stats::splinefun()`: This will typically be differentiable up to the third degree (i.e., `deriv = 0` is position, `deriv = 1` is speed, etc.).

The derivative returned (as column `interp`) is the derivative of distance with respect to time. This means the first derivative is velocity, second is acceleration, and third is jerk. The derivative is taken from the direct trajectory, not the inverse, and the inverse trajectory cannot be used to find derivatives. This means that if `new_distances` is provided, `deriv` must equal 0. If starting from distance values, but derivatives are desired, consider interpolating for timepoints first, then using these as `new_times` to find the derivative.

Prevents Extrapolation:

By default, many interpolating curves provided by R and `stats` will allow extrapolation (i.e., the input of an `event_timestamp` or `distance` beyond the original time or space domain of the trip). In general, this will not be reasonable for transit vehicles: time points should be constrained by the time that a trip has actually been observed, and distances should be constrained to the part of a route a trip actually ran.

This function uses the maximum and minimum time and distance values stored in the trajectory object to identify if an input `new_times` or `new_distances` is beyond the domain/range of each trip individually. The returned output will only include `interp` values for trips within the domain/range of the input.

Accessing the Raw Trajectory Function:

Because of the above features and protections, it is recommend that these `predict()` functions are used to access the fit trajectory and inverse trajectory functions. However, if the raw function itself is desired, it can be accessed using `attr(trajectory, "traj_fun")` or `attr(trajectory, "inv_traj_fun")`. For a group trajectory object, these will return lists of individual trip functions indexed by `trip_id_performed`; for single trajectory objects, these will return the single function for that trip.

Value

The input dataframe, with an additional column `interp` of the interpolated values requested, and an additional `trip_id_performed` column will all trips for which that point is within range. If `new_times` or `distance_lims`/`timestep` are used, a column `deriv` will also be included, indicating which derivative degree each interpolated row corresponds to.

Examples

```
# Set my parameters
my_times = seq(from = 1779890000,
               to = 1779893600,
               by = 180)
my_distances = seq(from = 100,
                   to = 35000,
                   by = 5000)
my_distance_lims = c(500, 600)
my_timestep = 10

# Get input data
lineE_traj <- new_transittraj_data("get_trajectory_fun")

# Run function: get distances from times
interp_dists <- predict(object = lineE_traj,
                        new_times = my_times)
dim(interp_dists)
head(interp_dists)

# Run function: get speeds from times
interp_speeds <- predict(object = lineE_traj,
                         new_times = my_times,
                         deriv = 1)
dim(interp_speeds)
head(interp_speeds)

# Run function: get times from distances
interp_times <- predict(object = lineE_traj,
                        new_distances = my_distances)
dim(interp_times)
head(interp_times)

# Run function: get time & distance pairs given distance bounds
interp_time_dist_pairs <- predict(object = lineE_traj,
                                  distance_lims = my_distance_lims,
```

```
                                    timestep = my_timestep)
dim(interp_time_dist_pairs)
head(interp_time_dist_pairs)

# Run function: vectorized derivatives
interp_vec <- predict(object = lineE_traj,
                      new_times = my_times,
                      deriv = c(0, 1, 2))
dim(interp_vec)
head(interp_vec)
```

`print.avltrajectory_group` *Print an AVL trajectory object*

Description

This function prints a one-line report for grouped or single trajectory objects. For a single trajectory, the trip ID will be printed. For grouped trajectories, the number of trips will be printed.

Usage

```
## S3 method for class 'avltrajectory_group'
print(x, ...)

## S3 method for class 'avltrajectory_single'
print(x, ...)
```

Arguments

`x`	A single or grouped trajectory object.
`...`	Other parameters (not used).

Value

A printing character string.

Examples

```
# Get input data
lineE_traj_grouped <- new_transittraj_data("get_trajectory_fun")
lineE_traj_singles <- new_transittraj_data("get_trajectory_fun_single")

# Print: Grouped trajectory object
print(lineE_traj_grouped)

# Print: Single trajectory object
print(lineE_traj_singles[[2]])
```

`project_onto_route` *Project points to linear distances along a route*

Description

This function takes spatial points and projects them onto a route (i.e., "snaps" them to the nearest point on the shape), returning the linear distance of each point along the route shape, starting from the route's beginning terminal.

Usage

```
project_onto_route(
  shape_geometry,
  points,
  original_crs = 4326,
  project_crs = 4326
)
```

Arguments

`shape_geometry`	The SF object to project onto. Must include the field `shape_id`. See `get_shape_geometry()`.
`points`	Can be either: a dataframe representing point coordinates, with fields `longitude` and `latitude`; or, an SF or SFC point object.
`original_crs`	Optional. A numeric EPSG identifier. If a dataframe is provided for `points`, this will be used to define the coordinate system of the longitude / latitude values. Default is 4326 (WGS 84 ellipsoid).
`project_crs`	Optional. A numeric EPSG identifer indicating the coordinate system to use for spatial calculations. Consider setting to a Euclidian projection, such as the appropriate UTM zone. Default is 4326 (WGS 84 ellipsoid).

Value

The `points` input (either dataframe or SF) with an appended column for the linear distance along the route. If `points` is an SFC, a vector of numeric distances is returned. Units are those of the spatial projection set in `project_crs` (e.g., meters if using WGS UTM).

Examples

```
# Set my parameters
my_crs <- 32611

# Get shape data
lineE_shape <- new_transittraj_data("get_shape_geometry")

# Set points of interest
my_points <- data.frame(longitude = c(-118.270924, -118.230056),
                        latitude = c(34.033895, 34.047884),
```

```
                          poi_name = c("Flower & Washington",
                                       "1st St Viaduct"))

# Run project_onto_route
my_points_proj <- project_onto_route(shape_geometry = lineE_shape,
                                     points = my_points,
                                     project_crs = my_crs)
head(my_points_proj)
```

---

summary.avltrajectory_group *Summarize an AVL trajectory object*

Description

This function creates and prints a list summarizing a single or grouped trajectory object. If the input is a single trajectory, the trip's ID and distance & time range will be printed. If the input is a grouped trajectory, the number of trips and the distance & time range across all trips will be printed. For both, the interpolating curve methods will be printed.

Usage

```
## S3 method for class 'avltrajectory_group'
summary(object, ...)

## S3 method for class 'avltrajectory_single'
summary(object, ...)
```

Arguments

object	A single or grouped trajectory object.
...	Other parameters (not used).

Value

A list summarizing the attributes of a fit trajectory.

Examples

```
# Get input data
lineE_traj_grouped <- new_transittraj_data("get_trajectory_fun")
lineE_traj_singles <- new_transittraj_data("get_trajectory_fun_single")

# Run function: grouped trajectory object
summary(lineE_traj_grouped)

# Run functions: store summary object
lineE_summ <- summary(lineE_traj_grouped)
print(lineE_summ$num_trips)
```

```
# Run function: single trajectory object
summary(lineE_traj_singles[[2]])
```

`trim_trips` *Remove observations occurring before a trip's minimum distance, or after a trip's maximum distance*

Description

Sometimes AVL pings can be recorded under a trip ID while a vehicle is still traveling in the opposite direction. Conversely, a trip may continue recording as it begins traversing the opposite direction. This function attempts to remove these observations by identifying each trip's minimum (beginning) and maximum (ending) distance, then filtering to only observations between these points. For both ends, the first occurrence of the beginning/ending value is used.

Usage

```
trim_trips(distance_df, trim_type = "both", return_removals = FALSE)
```

Arguments

`distance_df`	A dataframe of linearized AVL data. Must include `trip_id_performed`, `event_timestamp`, and `distance`.
`trim_type`	Optional. A string, indicating whether the beginning of trips, end of trips, or both beginning and end of trips should be trimmed. Must be one of `"beginning"`, `"end"`, or `"both"`. Default is `"both"`.
`return_removals`	Optional. A boolean, should the function return a dataframe of points removed and why? Default is `FALSE`.

Value

The input `distance_df` with violating points removed. If `return_removals = TRUE`, a dataframe with observations removed and why.

Examples

```
# Get input data
lineE_clean_trips <- new_transittraj_data("clean_incomplete_trips")
dim(lineE_clean_trips)

# Run function
lineE_trimmed <- trim_trips(distance_df = lineE_clean_trips)
dim(lineE_trimmed)
head(lineE_trimmed)
```

validate_monotonicity *Check if AVL data satisfies assumptions of monotonicity*

Description

This function checks whether the provided AVL dataframe of linearized distances satisfies three conditions:

- Weak monotonicity, either flat or increasing
- Strict monotonicty, increasing only
- Fritsch-Carlson constraints for speeds

See `make_monotonic()` for more information.

Usage

```
validate_monotonicity(distance_df, check_speed = FALSE, return_full = FALSE)
```

Arguments

`distance_df`	A dataframe of linearized AVL data. Must include `trip_id_performed`, `event_timestamp`, and `distance`. If `check_speed = TRUE`, must also include `speed`.
`check_speed`	Optional. A boolean, should the Fritsch-Carlson conditions for slopes be checked? Default is `FALSE`, where the speed check will return `NA`.
`return_full`	Optional. Should a dataframe of each point checked be returned? Default is `FALSE`.

Value

A named vector of booleans indicating whether each of the three conditions are satisfied.

Examples

```
# Get & test non-monotonic dataset
lineE_dists <- new_transittraj_data("get_linear_distances")
lineE_dists_val <- validate_monotonicity(lineE_dists, check_speed = TRUE)
print(lineE_dists_val)

# Get & test monotonic dataset
lineE_mono <- new_transittraj_data("make_monotonic")
lineE_mono_val <- validate_monotonicity(lineE_mono, check_speed = TRUE)
print(lineE_mono_val)
```

validate_tides *Check if AVL data meets TIDES standards*

Description

The transit integrated data exchange standard (TIDES) specifies columns that should be present in AVL data tables and the data types of these columns. This function verifies if those columns are present in the input `avl_df` are of the correct data type. See `Details` for more information.

Usage

```
validate_tides(avl_df)
```

Arguments

`avl_df`	A dataframe of AVL data, either in GPS (longitude/latitude) form or linearized distance form.

Details

The AVL cleaning functions in this package generally require the input dataframes to adhere to the TIDES `vehicle_locations` table schema. The following columns and data types are checked by this validator:

- `location_ping_id`: Should be a `character` string.
- `trip_id_performed`: Should be a `character` string.
- `event_timestamp`: Should be a `POSIXct` date-time.
- `vehicle_id`: Should be a `character` string.
- `operator_id`: Should be a `character` string. This field is not standard in TIDES `vehicle_locations`, and is not a strict requirement for any AVL processing functions.
- `longitude` and `latitude`: Should be `numeric`. These fields are required only to linearize AVL data, and not used afterwards.
- `distance`: Should be `numeric`. This field is not standard in TIDES `vehicle_locations`, and is generated by `get_linear_distances()`. It is required by most other AVL processing functions.
- `speed`: Should be `numeric`. This field is not a strict requirement for any AVL processing functions.

Each AVL processing function in `transittraj` uses specific fields. Each function verifies that the required fields and data types are present before proceeding.

Value

A dataframe of each required field, required data type, whether the field is present, and whether the data type matches expectations.

Examples

```
lacmta_val <- validate_tides(avl_df = lacmta_avl)
print(lacmta_val)
```

Index

∗ **datasets**
    lacmta_avl, 23
    lacmta_gtfs, 24

clean_incomplete_trips, 2
clean_jumps, 4
clean_overlapping_subtrips, 6

export_animation, 7

filter_by_route, 9

get_gtfs_service_dates, 10
get_gtfs_trajectory_fun, 11
get_linear_distances, 15
get_shape_geometry, 17
get_stop_distances, 18
get_trajectory_fun, 19
get_trip_extremes, 21
group_trajectories, 22

lacmta_avl, 23
lacmta_gtfs, 24

make_monotonic, 26

new_transittraj_data, 28

plot.avltrajectory_group, 29
plot.avltrajectory_single
        *(*plot.avltrajectory_group*)*, 29
plot_animated_line, 30
plot_animated_map *(*plot_animated_line*)*,
        30
plot_interactive_gtfs, 35
plot_trajectory, 37
predict.avltrajectory_group, 40
print.avltrajectory_group, 44
print.avltrajectory_single
        *(*print.avltrajectory_group*)*, 44
project_onto_route, 45

summary.avltrajectory_group, 46
summary.avltrajectory_single
        *(*summary.avltrajectory_group*)*,
        46

trim_trips, 47

validate_monotonicity, 48
validate_tides, 49