resonate Documentation

Release v1.0.9

Kenneth Reitz

Jul 12, 2023

Contents

resonATe is the Ocean Tracking Network's acoustic telemetry analysis toolkit. It can be used to filter, compress, visualize and analyze acoustic detection extracts from OTN and other marine telemetry data.

CHAPTER 1

Abacus Plot

The abacus plot is a way to plot annimal along time. The function uses Plotly to place your points on a scatter plot. `ycolumn` is used as the y axis and `datecollected` is used as the x axis. `color_column` is used to group detections together and assign them a color. Details are in *Abacus Plot*.

CHAPTER 2

Bubble Plot

The bubble plot function returns a Plotly scatter plot layered ontop of a map. The color of the markers will indicate the number of detections at each location. Alternatively, you can indicate the number of individuals seen at each location by using `type = 'individual'`. Details are in *Bubble Plot*.

CHAPTER 3

Cohort

The tool takes a file of compressed detections and a time parameter in minutes. It identifies groups of animals traveling together. Each station an animal visits is checked for other animals detected there within the specified time period. Details are in *Cohort Tool*.

CHAPTER 4

Compressing Detections

Compressing detections is done by looking at the detection times and locations of an animal. Any detections that occur successively in time, in the same location, and the time between detections does not exceed the `timefilter`, are combined into a single detection with a start and end time. The result is a compressed detections Pandas DataFrame.

Compression is the first step of the Mihoff Interval Data Tool. Compressed detection DataFrames are needed for the tools, such as interval and cohort. Details are in *Compression Tool*.

CHAPTER 5

Filtering

(White, E., Mihoff, M., Jones, B., Bajona, L., Halfyard, E. 2014. White-Mihoff False Filtering Tool)

OTN has developed a tool which will assist with filtering false detections. The first level of filtering involves identifying isolated detections. The original concept came from work done by Easton White. He was kind enough to share his research database with OTN. We did some preliminary research and developed a proposal for a filtering tool based on what Easton had done. This proof of concept was presented to Steve Kessel and Eddie Halfyard in December 2013 and a decision was made to develop a tool for general use.

This is a very simple tool. It will take an input file of detections and based on an input parameter will identify suspect detections. The suspect detections will be put into a dataframe which the user can examine. There will be enough information for each suspect detection for the user to understand why it was flagged. There is also enough information to be able to reference the detection in the original file if the user wants to see what was happening at the same time.

The input parameter is a time in minutes. We used 3600 seconds as the default as this is what was used in Easton's code. This value can be changed by the user. The output contains a record for each detection for which there has been more than xx minutes since the previous detection (of that tag/animal) and more than the same amount of time until the next detection. It ignores which receiver the detection occurred at. That is all it does, nothing more and nothing less. Details are in *Filter Tool*.

Two other filtering tools are available as well, one based on distance alone and one based on velocity. They can be found at *Filter Tools* as well.

CHAPTER 6

Distance Matrix

This takes a DataFrame created by the White-Mihoff False Filtering tool. The file contains rows of station pairs with the straight line distance between them calculated in metres. A station pair will only be in the file if an animal traveled between the stations. If an animal goes from stn1 to stn2 and then to stn3, pairs stn1-stn2 and stn2-stn3 will be in the file. If no animal goes between stn1 and stn3, that pair will not be in the file. The tool also takes a file that the researcher provides of 'real distances'. The output will be a file which looks like the first file with the 'real distance' column updated. Details are in *Distance Matrix Tool*

CHAPTER 7

Interval Data

(Mihoff, M., Jones, B., Bajona, L., Halfyard, E. 2014. Mihoff Interval Data Tool)

This tool will take a DataFrame of compressed detections and a distance matrix and output an interval DataFrame. The Interval DataFrame will contain records of the animal id, the arrival time at stn1, the departure time at stn1, the detection count at stn1, the arrival time at stn2, time between detections at the two stations, the interval in seconds, the distance between stations, and the velocity of the animal in m/s. Details are in *Interval Data Tool*.

CHAPTER 8

Residence Index

This residence index tool will take a compressed or uncompressed detection file and caculate the residency index for each station/receiver in the detections. A CSV file will be written to the data directory for future use. A Pandas DataFrame is returned from the function, which can be used to plot the information. The information passed to the function is what is used to calculate the residence index, make sure you are only passing the data you want taken into consideration for the residence index (i.e. species, stations, tags, etc.). Details in *Residence Index Tool*.

CHAPTER 9

Receiver Efficiency Index

(Ellis, R., Flaherty-Walia, K., Collins, A., Bickford, J., Walters Burnsed, Lowerre-Barbieri S. 2018. Acoustic telemetry array evolution: from species- and project-specific designs to large-scale, multispecies, cooperative networks)

The receiver efficiency index is number between `0` and `1` indicating the amount of relative activity at each receiver compared to the entire set of receivers, regardless of positioning. The function takes a set detections and a deployment history of the receivers to create a context for the detections. Both the amount of unique tags and number of species are taken into consideration in the calculation. For the exact method, see the details in *Receiver Efficiency Index*.

CHAPTER 10

Unique Id

This tool will add a column to any file. The unique id will be sequential integers. No validation is done on the input file. Details in *Unique Detections ID*.

CHAPTER 11

Visual Timeline

This tool takes a detections extract file and generates a Plotly animated timeline, either in place in an iPython notebook or exported out to an HTML file. Details in *Visual Timeline*.

CHAPTER 12

Contents:

12.1 Installation

resonATe can be installed using Pip or through a Conda environment.

12.1.1 Conda

```
conda config --add channels conda-forge
conda install resonate
```

12.1.2 Pip

```
pip install resonate
```

12.2 Preparing Data

resonATe requires your acoustic telemetry data to have specific column headers. The column headers are the same ones used by the Ocean Tracking Network for their detection extracts.

The columns you need are as follows:

- **catalognumber** - A unique identifier assigned to an animal.
- **station** - A unique identifier for the station or mooring where the receiver was located. This column is used in resonATe for grouping detections which should be considered to have occurred in the same place.
- **datecollected** - Date and time of release or detection, all of which have the same timezone (example format: `2018-02-02 04:09:45`).
- **longitude** - The receiver location at time of detection in decimal degrees.

- **latitude** - The receiver location at time of detection in decimal degrees.
- **scientificname** - The taxonmoic name for the animal detected.
- **fieldnumber** - The unique number for the tag/device attached to the animal.
- **unqdetecid** - A unique value assigned to each record in the data. resonATe includes a function to generate this column if needed. Details in *Unique Detections ID*.

The *Receiver Efficiency Index* also needs a deployment history for stations. The columns for deployments are as follows:

- **station_name** - A unique identifier for the station or mooring where the receiver was located. This column is used in resonATe for grouping detections which should be considered to have occurred in the same place.
- **deploy_date** - A date of when the receiver was placed in a water or is active (example format: `2018-02-02`).
- **recovery_date** - A date of when the receiver was removed from the water or became inactive (example format: `2018-02-02`).
- **last_download** - A date of the last time data was retrieved from the receiver (example format: `2018-02-02`).

All other columns are not required and will not affect the functions; however, they may be used in some functions. For example, `receiver_group` can be used color code data in the *Abacus Plot*.

Warning: Detection records from mobile receivers, i.e. from receivers attached to gliders or animals, as well as satellite transmitter detections, will not necessarily be appropriate or compatible for use with all of these tools.

12.2.1 Renaming Columns

Pandas provides a `rename()` function that can be implemented as follows:

```
import pandas as pd

df = pd.read_csv('/path/to/detections.csv')

df.rename(index=str, columns={
  'your_animal_id_column':'catalognumber',
  'your_station_column':'station',
  'your_date_time_column':'datecollected',
  'your_longitude_column':'longitude',
  'your_latitude_column':'latitude',
  'your_unique_id_column':'unqdetecid'
}, inplace=True)
```

12.2.2 Example Dataset

catalognumber	scientificname	commonname	receiver_group	station	datecollected	timezone	longitude	latitude	unqdetecid
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX248	2014-06-08 20:10	UTC	-63.50002	42.89487	HFX-A69-9001-26655-170932
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX248	2014-06-08 20:12	UTC	-63.50002	42.89487	HFX-A69-9001-26655-170933
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX249	2014-06-08 20:12	UTC	-63.50002	42.88764	HFX-A69-9001-26655-170934
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX248	2014-06-08 20:14	UTC	-63.50002	42.89487	HFX-A69-9001-26655-170935
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX248	2014-06-08 20:16	UTC	-63.50002	42.89487	HFX-A69-9001-26655-170936
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX248	2014-06-08 20:17	UTC	-63.50002	42.89487	HFX-A69-9001-26655-170937
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX248	2014-06-08 20:27	UTC	-63.50002	42.89487	HFX-A69-9001-26655-170938
NSBS-Sophie	Prionace glauca	blue shark	HFX	HFX247	2014-06-08 20:28	UTC	-63.49995	42.90203	HFX-A69-9001-26655-170939
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX119	2014-06-20 11:36	UTC	-63.3331	43.79986	HFX-A69-9001-26667-173924
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX118	2014-06-20 11:37	UTC	-63.32552	43.8043	HFX-A69-9001-26667-171528
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX119	2014-06-20 11:38	UTC	-63.3331	43.79986	HFX-A69-9001-26667-173925
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX118	2014-06-20 11:38	UTC	-63.32552	43.8043	HFX-A69-9001-26667-171529
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX119	2014-06-20 11:40	UTC	-63.3331	43.79986	HFX-A69-9001-26667-173926
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX118	2014-06-20 11:41	UTC	-63.32552	43.8043	HFX-A69-9001-26667-171530
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX119	2014-06-20 11:42	UTC	-63.3331	43.79986	HFX-A69-9001-26667-173927
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX118	2014-06-20 11:43	UTC	-63.32552	43.8043	HFX-A69-9001-26667-171531
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX119	2014-06-20 11:44	UTC	-63.3331	43.79986	HFX-A69-9001-26667-173928
NSBS-Lola	Prionace glauca	blue shark	HFX	HFX119	2014-06-20 11:46	UTC	-63.3331	43.79986	HFX-A69-9001-26667-173929

12.3 Abacus Plot

The abacus plot is a way to plot annimal along time. The function uses Plotly to place your points on a scatter plot. `ycolumn` is used as the y axis and `datecollected` is used as the x axis. `color_column` is used to group detections together and assign them a color.

> **Warning:** Input files must include `datecollected` as a column.

```
from resonate.abacus_plot import abacus_plot
import pandas as pd

df = pd.read_csv('/path/to/detections.csv')
```

To display the plot in iPython use:

```
abacus_plot(df, ycolumn='catalognumber', color_column='receiver_group')
```

Or use the standard plotting function to save as HTML:

```
abacus_plot(df, ipython_display=False, filename='example.html')
```

Below is the sample output for blue sharks off of the coast of Nova Scotia.

12.3.1 Abacus Plot Function

`abacus_plot.`**`abacus_plot`**(*detections*, *ycolumn='catalognumber'*, *color_column=None*, *ipython_display=True*, *title='Abacus Plot'*, *filename=None*)

Creates a plotly abacus plot from a pandas dataframe

Parameters

- **`detections`** – detection dataframe
- **`ycolumn`** – the series/column for the y axis of the plot
- **`color_column`** – the series/column to group by and assign a color
- **`ipython_display`** – a boolean to show in a notebook
- **`title`** – the title of the plot
- **`filename`** – Plotly filename to write to

Returns A plotly scatter plot

12.4 Bubble Plot

The bubble plot function returns a Plotly scatter plot layered ontop of a map. The color of the markers will indicate the number of detections at each location. Alternatively, you can indicate the number of individuals seen at each location by using `type = 'individual'`.

> **Warning:** Input files must include `station` , `catalognumber`, `unqdetecid`, `latitude`, `longitude`, and `datecollected` as columns.

```
from resonate.bubble_plot import bubble_plot
import pandas as pd
import plotly.offline as py

df = pd.read_csv('/path/to/detections.csv')
```

To display the plot in iPython use:

```
bubble_plot(df)
```

Or use the standard plotting function to save as HTML:

```
bubble_plot(df,ipython_display=False, filename='/path_to_plot.html')
```

You can also do your count by number of individuals by using `type = 'individual`:

```
bubble_plot(df, type='individual')
```

12.4.1 Mapbox

Alternatively you can use a Mapbox access token plot your map. Mapbox is much for responsive than standard Scattergeo plot.

Example Code

```
mapbox_access_token = 'ADD_YOUR_TOKEN_HERE'
bubble_plot(df, mapbox_token=mapbox_access_token)
```

Below is the sample output for blue sharks off of the coast of Nova Scotia.

12.4.2 Bubble Plot Function

`bubble_plot.`**`bubble_plot`**(*detections*, *type='detections'*, *ipython_display=True*, *title='Bubble Plot'*, *height=700*, *width=1000*, *plotly_geo=None*, *filename=None*, *mapbox_token=None*, *marker_size=10*, *colorscale='Viridis'*)

Creates a plotly abacus plot from a pandas dataframe

Parameters

- **`detections`** – detection dataframe
- **`ipython_display`** – a boolean to show in a notebook
- **`title`** – the title of the plot
- **`height`** – the height of the plotl
- **`width`** – the width of the plotly
- **`plotly_geo`** – an optional dictionary to controle the geographix aspects of the plot
- **`filename`** – Plotly filename to write to
- **`mapbox_token`** – A string of mapbox access token
- **`marker_size`** – An int to indicate the diameter in pixels

- **colorscale** – A string to indicate the color index

Returns A plotly geoscatter plot or mapbox plot

12.5 Cohort

The tool takes a dataframe of compressed detections and a time parameter in minutes. It identifies groups of animals traveling together. Each station an animal visits is checked for other animals detected there within the specified time period.

The function returns a dataframe which you can use to help identify animal cohorts. The cohort is created from the compressed data that is a result from the `compress_detections()` function. Pass the compressed dataframe into the `cohort()` function along with a time interval in seconds (default is 3600) to create the cohort dataframe.

Warning:

Input files must include `station`, `catalognumber`, `seq_num`, `unqdetecid`, and `datecollected` as columns.

```
from resonate.cohorts import cohort
from resonate.compress import compress_detections
import pandas as pd

time_interval = 3600 # in seconds

data = pd.read_csv('/path/to/detections.csv')

compressed_df = compress_detections(data)

cohort_df = cohort(compressed_df, time_interval)

cohort_df
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```
# Saves the cohort file
cohort_df.to_csv('/path/to/output.csv', index=False)
```

12.5.1 Cohort Functions

`cohorts.`**`cohort`**(*compressed_df*, *interval_time=3600*)

Creates a dataframe of cohorts using a compressed detection file

Parameters

- **compressed_df** – compressed dataframe
- **interval_time** – cohort detection time interval (in seconds)

Returns

cohort dataframe with the following columns

- anml_1
- anml_1_seq

- station
- anml_2
- anml_2_seq
- anml_2_arrive
- anml_2_depart
- anml_2_startunqdetecid
- anml_2_endunqdetecid
- anml_2_detcount

12.6 Compressing Detections

Compressing detections is done by looking at the detection times and locations of an animal. Any detections that occur successively in time, and the time between detections does not exceed the `timefilter`, in the same location are combined into a single detection with a start and end time. The result is a compressed detections Pandas DataFrame.

Compression is the first step of the Mihoff Interval Data Tool. Compressed detection DataFrames are needed for the tools, such as interval and cohort.

Warning: Input files must include `datecollected`, `catalognumber`, and `unqdetecid` as columns.

```
from resonate.compress import compress_detections
import pandas as pd

detections = pd.read_csv('/path/to/data.csv')

compressed = compress_detections(detections=detections)
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```
compressed.to_csv('/path/to/output.csv', index=False)
```

12.6.1 Compression Functions

`compress.`**`compress_detections`**(*detections*, *timefilter=3600*)

Creates compressed dataframe from detection dataframe

Parameters

- **`detections`** – detection dataframe
- **`timefilter`** – A maximum amount of time in seconds that can pass before a new detction event is started

Returns A Pandas DataFrame matrix of detections events

12.7 Filtering Detections on Distance / Time

12.7.1 White/Mihoff Filter

(White, E., Mihoff, M., Jones, B., Bajona, L., Halfyard, E. 2014. White-Mihoff False Filtering Tool)

OTN has developed a tool which will assist with filtering false detections. The first level of filtering involves identifying isolated detections. The original concept came from work done by Easton White. He was kind enough to share his research database with OTN. We did some preliminary research and developed a proposal for a filtering tool based on what Easton had done. This proof of concept was presented to Steve Kessel and Eddie Halfyard in December 2013 and a decision was made to develop a tool for general use.

This is a very simple tool. It will take an input file of detections and based on an input parameter will identify suspect detections. The suspect detections will be put into a dataframe which the user can examine. There will be enough information for each suspect detection for the user to understand why it was flagged. There is also enough information to be able to reference the detection in the original file if the user wants to see what was happening at the same time.

The input parameter is a time in seconds. We used 3600 seconds as the default as this is what was used in Easton's code. This value can be changed by the user. The output contains a record for each detection for which there has been more than xx seconds since the previous detection (of that tag/animal) and more than the same amount of time until the next detection. It ignores which receiver the detection occurred at. That is all it does, nothing more and nothing less.

Below the interval is set to 3600 seconds and is not using a a user specified suspect file. The function will also create a distance matrix.

Warning: Input files must include `datecollected`, `catalognumber`, `station` and `unqdetecid` as columns.

```python
from resonate.filters import get_distance_matrix
from resonate.filters import filter_detections
import pandas as pd

detections = pd.read_csv('/path/to/detections.csv')

time_interval = 3600 # in seconds

SuspectFile = None

CreateDistanceMatrix = True

filtered_detections = filter_detections(detections,
                                        suspect_file=SuspectFile,
                                        min_time_buffer=time_interval,
                                        distance_matrix=CreateDistanceMatrix)
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```python
filtered_detections['filtered'].to_csv('/path/to/output.csv', index=False)

filtered_detections['suspect'].to_csv('/path/to/output.csv', index=False)

filtered_detections['dist_mtrx'].to_csv('/path/to/output.csv', index=False)
```

12.7.2 Distance Filter

The distance filter will separate detections based only on distance. The `maximum_distance` argument defaults to 100,000 meters (or 100 kilometers), but can be adjusted. Any detection where the succeeding and preceding detections are more than the `maximum_distance` away will be considered suspect.

Warning: Input files must include `datecollected`, `catalognumber`, `station` and `unqdetecid` as columns.

```
from resonate.filters import distance_filter
import pandas as pd

detections = pd.read_csv('/path/to/detections.csv')


filtered_detections = distance_filter(detections)
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```
filtered_detections['filtered'].to_csv('/path/to/output.csv', index=False)

filtered_detections['suspect'].to_csv('/path/to/output.csv', index=False)
```

12.7.3 Velocity Filter

The velocity filter will separate detections based on the animal's velocity. The `maximum_velocity` argument defaults to 10 m/s, but can be adjusted. Any detection where the succeeding and preceding velocities of an animal are more than the `maximum_velocity` will be considered suspect.

Warning: Input files must include `datecollected`, `catalognumber`, `station` and `unqdetecid` as columns.

```
from resonate.filters import velocity_filter
import pandas as pd

detections = pd.read_csv('/path/to/detections.csv')


filtered_detections = velocity_filter(detections)
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```
filtered_detections['filtered'].to_csv('/path/to/output.csv', index=False)

filtered_detections['suspect'].to_csv('/path/to/output.csv', index=False)
```

12.7.4 Filtering Functions

`filters.`**`distance_filter`**`(`*detections*, *maximum_distance=100000*`)`

Parameters

- **`detections`** – a Pandas DataFrame of acoustic detection
- **`maximum_distance`** – a umber in meters, default is 100000

Returns A list of Pandas DataFrames of filtered detections and suspect detections

`filters.`**`filter_detections`**(*detections*, *suspect_file=None*, *min_time_buffer=3600*, *distance_matrix=False*)

Filters isolated detections that are more than min_time_buffer apart from other dets. for a series of detections in detection_file. Returns Filtered and Suspect dataframes. suspect_file can be a file of existing suspect detections to remove before filtering. dist_matrix is created as a matrix of between-station distances from stations defined in the input file.

Parameters

- **`detections`** – A Pandas DataFrame of acoustic detections
- **`suspect_file`** – Path to a user specified suspect file, same format as the detections
- **`min_time_buffer`** – The minimum of time required for outlier detections in seconds
- **`distance_matrix`** – A boolean of whether or not to generate the distance matrix

Returns A list of Pandas DataFrames of filtered detections, suspect detections, and a distance matrix

`filters.`**`get_distance_matrix`**(*detections*)

Creates a distance matrix of all stations in the array or line.

Parameters **`detections`** – a Pandas DataFrame of detections

Returns A Pandas DataFrame matrix of station to station distances

`filters.`**`velocity_filter`**(*detections*, *maximum_velocity=10*)

Parameters

- **`detections`** –
- **`maximum_velocity`** –

Returns A list of Pandas DataFrames of filtered detections and suspect detections

12.8 Interval Data

`interval_data()` takes a compressed detections DataFrame, a distance matrix, and a detection radius DataFrame and creates an interval data DataFrame.

Intervals are lengths of time in which a station detected an animal. Many consecutive detections of an animal are replaced by one interval.

Warning: Input files must include `datecollected`, `catalognumber`, and `unqdetecid` as columns.

```python
from resonate.filters import get_distance_matrix
from resonate.compress import compress_detections
from resonate.interval_data_tool import interval_data
import pandas as pd
import geopy

input_file = pd.read_csv("/path/to/detections.csv")
```

(continues on next page)

(continued from previous page)

```
compressed = compress_detections(input_file)
matrix = get_distance_matrix(input_file)
```

Set the station radius for each station name.

```
detection_radius = 400

station_det_radius = pd.DataFrame([(x, geopy.distance.Distance(detection_radius/1000.
↪0))
                                   for x in matrix.columns.tolist()], columns=[
↪'station','radius'])

station_det_radius.set_index('station', inplace=True)

station_det_radius
```

You can modify individual stations if needed by using `DatraFrame.set_value()` from Pandas.

```
station_name = 'HFX001'

station_detection_radius = 500

station_det_radius.at[station_name, 'radius'] = geopy.distance.Distance( station_
↪detection_radius/1000.0 )
```

Create the interval data by passing the compressed detections, the matrix, and the station radii.

```
interval = interval_data(compressed_df=compressed, dist_matrix_df=matrix, station_
↪radius_df=station_det_radius)

interval
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```
interval.to_csv('/path/to/output.csv', index=False)
```

12.8.1 Interval Data Functions

`interval_data_tool.`**`interval_data`**(*compressed_df*, *dist_matrix_df*, *station_radius_df=None*)

Creates a detection interval file from a compressed detection, distance matrix and station detection radius DataFrames

Parameters

- **`compressed_df`** – compressed detection dataframe
- **`dist_matrix_df`** – station distance matrix dataframe
- **`station_radius`** – station distance radius dataframe

Returns interval detection Dataframe

12.9 Residence Index

Kessel et al. Paper https://www.researchgate.net/publication/279269147

This residence index tool will take a compressed or uncompressed detection file and caculate the residency index for each station/receiver in the detections. A CSV file will be written to the data directory for future use. A Pandas DataFrame is returned from the function, which can be used to plot the information. The information passed to the function is what is used to calculate the residence index, **make sure you are only passing the data you want taken into consideration for the residence index (i.e. species, stations, tags, etc.)**.

detections: The CSV file in the data directory that is either compressed or raw. If the file is not compressed please allow the program time to compress the file and add the rows to the database. A compressed file will be created in the data directory. Use the compressed file for any future runs of the residence index function.

calculation_method: The method used to calculate the residence index. Methods are:

- kessel
- timedelta
- aggregate_with_overlap
- aggregate_no_overlap.

project_bounds: North, South, East, and West bounding longitudes and latitudes for visualization.

The calculation methods are listed and described below before they are called. The function will default to the Kessel method when nothing is passed.

Below is an example of inital variables to set up, which are the detection file and the project bounds.

Warning: Input files must include `datecollected`, `station`, `longitude`, `latitude`, `catalognumber`, and `unqdetecid` as columns.

```
from resonate import residence_index as ri
import pandas as pd

detections = pd.read_csv('/Path/to/detections.csv')
```

12.9.1 Kessel Residence Index Calculation

The Kessel method converts both the startdate and enddate columns into a date with no hours, minutes, or seconds. Next it creates a list of the unique days where a detection was seen. The size of the list is returned as the total number of days as an integer. This calculation is used to determine the total number of distinct days (T) and the total number of distinct days per station (S).

$RI = \frac{S}{T}$

RI = Residence Index

S = Distinct number of days detected at the station

T = Distinct number of days detected anywhere on the array

Warning: Possible rounding error may occur as a detection on `2016-01-01 23:59:59` and a detection on `2016-01-02 00:00:01` would be counted as two days when it is really 2-3 seconds.

Kessel RI Example Code

```
kessel_ri = ri.residency_index(detections, calculation_method='kessel')

ri.plot_ri(kessel_ri)
```

12.9.2 Timedelta Residence Index Calculation

The Timedelta calculation method determines the first startdate of all detections and the last enddate of all detections. The time difference is then taken as the values to be used in calculating the residence index. The timedelta for each station is divided by the timedelta of the array to determine the residence index.

$RI = \frac{\Delta S}{\Delta T}$

RI = Residence Index

ΔS = Last detection time at a station - First detection time at the station

ΔT = Last detection time on an array - First detection time on the array

Timedelta RI Example Code

```
timedelta_ri = ri.residency_index(detections, calculation_method='timedelta')

ri.plot_ri(timedelta_ri)
```

12.9.3 Aggregate With Overlap Residence Index Calculation

The Aggregate With Overlap calculation method takes the length of time of each detection and sums them together. A total is returned. The sum for each station is then divided by the sum of the array to determine the residence index.

RI = $\frac{AwOS}{AwOT}$

RI = Residence Index

AwOS = Sum of length of time of each detection at the station

AwOT = Sum of length of time of each detection on the array

Aggregate With Overlap RI Example Code

```
with_overlap_ri = ri.residency_index(detections, calculation_method='aggregate_with_
↪overlap')

ri.plot_ri(with_overlap_ri)
```

12.9.4 Aggregate No Overlap Residence Index Calculation

The Aggregate No Overlap calculation method takes the length of time of each detection and sums them together. However, any overlap in time between one or more detections is excluded from the sum.

For example, if the first detection is from **2016-01-01 01:02:43** to **2016-01-01 01:10:12** and the second detection is from **2016-01-01 01:09:01** to **2016-01-01 01:12:43**, then the sume of those two detections would be 10 minutes.

A total is returned once all detections of been added without overlap. The sum for each station is then divided by the sum of the array to determine the residence index.

RI = $\frac{AnOS}{AnOT}$

RI = Residence Index

AnOS = Sum of length of time of each detection at the station, excluding any overlap

AnOT = Sum of length of time of each detection on the array, excluding any overlap

Aggregate No Overlap RI Example Code

```
no_overlap_ri = ri.residency_index(detections, calculation_method='aggregate_no_
↪overlap')

ri.plot_ri(no_overlap_ri, title="ANO RI")
```

12.9.5 Mapbox

Alternatively you can use a Mapbox access token plot your map. Mapbox is much for responsive than standard Scattergeo plot.

Mapbox Example Code

```
mapbox_access_token = 'YOUR MAPBOX ACCESS TOKEN HERE'
kessel_ri = ri.residency_index(detections, calculation_method='kessel')
ri.plot_ri(kessel_ri, mapbox_token=mapbox_access_token,marker_size=40, scale_
↪markers=True)
```

12.9.6 Residence Index Functions

`residence_index.`**`aggregate_total_no_overlap`**(*detections*)

The function below aggregates timedelta of startdate and enddate, excluding overlap between detections. Any overlap between two detections is converted to a new detection using the earlier startdate and the latest enddate. If the startdate and enddate are the same, a timedelta of one second is assumed.

Parameters **`detections`** – pandas DataFrame pulled from the compressed detections CSV

Returns An float in the number of days

`residence_index.`**`aggregate_total_with_overlap`**(*detections*)

The function below aggregates timedelta of startdate and enddate of each detection into a final timedelta then returns a float of the number of days. If the startdate and enddate are the same, a timedelta of one second is assumed.

Parameters **`detections`** – Pandas DataFrame pulled from the compressed detections CSV

Returns An float in the number of days

`residence_index.`**`get_days`**(*dets*, *calculation_method='kessel'*)

Determines which calculation method to use for the residency index.

Wrapper method for the calulation methods above.

Parameters

- **`dets`** – A Pandas DataFrame pulled from the compressed detections CSV
- **`calculation_method`** – determines which method above will be used to count total time and station time

Returns An int in the number of days

`residence_index.`**`get_station_location`**`(`*station*, *detections*`)`

Returns the longitude and latitude of a station/receiver given the station and the table name.

Parameters

- **`station`** – String that contains the station name
- **`detections`** – the table name in which to find the station

Returns A Pandas DataFrame of station, latitude, and longitude

`residence_index.`**`plot_ri`**`(`*ri_data*, *ipython_display=True*, *title='Bubble Plot'*, *height=700*, *width=1000*, *plotly_geo=None*, *filename=None*, *marker_size=6*, *scale_markers=False*, *colorscale='Viridis'*, *mapbox_token=None*`)`

Parameters

- **`ri_data`** – A Pandas DataFrame generated from `residency_index()`
- **`ipython_display`** – a boolean to show in a notebook
- **`title`** – the title of the plot
- **`height`** – the height of the plotly
- **`width`** – the width of the plotly
- **`plotly_geo`** – an optional dictionary to control the geographic aspects of the plot
- **`filename`** – Plotly filename to write to
- **`mapbox_token`** – A string of mapbox access token
- **`marker_size`** – An int to indicate the diameter in pixels
- **`scale_markers`** – A boolean to indicate whether or not markers are scaled by their value
- **`colorscale`** – A string to indicate the color index. See here for options: https://community.plot.ly/t/what-colorscales-are-available-in-plotly-and-which-are-the-default/2079

Returns A plotly geoscatter

`residence_index.`**`residency_index`**`(`*detections*, *calculation_method='kessel'*`)`

This function takes in a detections CSV and determines the residency index for reach station.

Residence Index (RI) was calculated as the number of days an individual fish was detected at each receiver station divided by the total number of days the fish was detected anywhere on the acoustic array. - Kessel et al.

Parameters

- **`detections`** – CSV Path
- **`calculation_method`** – determines which method above will be used to count total time and station time

Returns

A residence index DataFrame with the following columns

- days_detected

- latitude
- longitude
- residency_index
- station

`residence_index.`**`total_days_count`**(*detections*)

The function below takes a Pandas DataFrame and determines the number of days any detections were seen on the array.

The function converst both the startdate and enddate columns into a date with no hours, minutes, or seconds. Next it creates a list of the unique days where a detection was seen. The size of the list is returned as the total number of days as an integer.

* **NOTE** ** Possible rounding error may occur as a detection on 2016-01-01 23:59:59 and a detection on 2016-01-02 00:00:01 would be counted as days when it is really 2-3 seconds.

Parameters **`detections`** – Pandas DataFrame pulled from the compressed detections CSV

Returns An int in the number of days

`residence_index.`**`total_days_diff`**(*detections*)

Determines the total days difference.

The difference is determined by the minimal startdate of every detection and the maximum enddate of every detection. Both are converted into a datetime then subtracted to get a timedelta. The timedelta is converted to seconds and divided by the number of seconds in a day (86400). The function returns a floating point number of days (i.e. 503.76834).

Parameters **`detections`** – Pandas DataFrame pulled from the compressed detections CSV

Returns An float in the number of days

12.10 Receiver Efficiency Index

The receiver efficiency index is number between `0` and `1` indicating the amount of relative activity at each receiver compared to the entire set of receivers, regardless of positioning. The function takes a set detections and a deployment history of the receivers to create a context for the detections. Both the amount of unique tags and number of species are taken into consideration in the calculation.

The receiver efficiency index implement is implemented based on the paper Acoustic telemetry array evolution: From species- and project-specific designs to large-scale, multispecies, cooperative networks. Each receiver's index is calculated on the formula of:

REI = $\frac{T_r}{T_a} \times \frac{S_r}{S_a} \times \frac{DD_r}{DD_a} \times \frac{D_a}{D_r}$

- REI = Receiver Efficiency Index
- T_r = The number of tags detected on the receievr
- T_a = The number of tags detected across all receivers
- S_r = The number of species detected on the receiver
- S_a = The number of species detected across all receivers
- DD_a = The number of unique days with detections across all receivers
- DD_r = The number of unique days with detections on the receiver
- D_a = The number of days the array was active

- D_r = The number of days the receiver was active

Each REI is then normalized against the sum of all considered stations. The result is a number between `0` and `1` indicating the relative amount of activity at each receiver.

Warning: Detection input files must include `datecollected`, `fieldnumber`, `station`, and `scientificname` as columns and deployment input files must include `station_name`, `deploy_date`, `last_download`, and `recovery_date` as columns.

`REI()` takes two arguments. The first is a dataframe of detections the detection timstamp, the station identifier, the species, and the tag identifier. The next is a dataframe of deployments for each station. The station name should match the stations in the detections. The deployments need to include a deployment date and recovery date or last download date. Details on the columns metnioned see the preparing data section.

Warning: This function assumes that no deployments for single station overlap. If deployments do overlap, the overlapping days will be counted twice.

```
from resonate.receiver_efficiency import REI

detections = pd.read_csv('/path/to/detections.csv')
deployments = pd.read_csv('/path/to/deployments.csv')

station_REIs = REI(detections = detections, deployments = deployments)
```

12.10.1 Residence Index Functions

`receiver_efficiency.`**`REI`**`(`*detections*, *deployments*`)`

Calculates a returns a list of each station and the REI (defined here):

Parameters

- **`detections`** – a pandas DataFrame of detections
- **`deployments`** – a pandas DataFrame of station deployment histories

Returns a pandas DataFrame of station, REI, latitude, and longitude

12.11 Subsetting Data

Sometimes there is too much data for a visualization tool to handle, or you wish to only take a certain subset of your input data and apply it elsewhere.

These examples, written in Python and leveraging the Pandas data manipulation package, are meant as a starting point. More complex operations are possible in Pandas, but these should form a baseline of understanding that will cover the most common operations.

```
import pandas as pd
filename = "/path/to/data.csv"
data = pd.read_csv(directory+filename)
```

12.11.1 Subsetting data by date range

Provide a date field, as well as starting and ending date range. By default, the detection date column of a detection extract file is provided.

```
# Enter the column name that contains the date you wish to evaluate
datecol = 'datecollected'
# Enter the start date in the following format
startdate = "YYYY-MM-DD"

# Enter the end date in the following format
enddate = "YYYY-MM-DD"

# Subsets the dat between the two indicated dates uding the datecollected column
data_date_subset = data[(data[datecol] > startdate) & (data[datecol] < enddate)]

# Output the subset data to a new CSV in the indicated directory
data_date_subset.to_csv(directory+startdate+"_to_"+enddate+"_"+filename, index=False)
```

12.11.2 Subsetting on column value

Provide the column you expect to have a certain value and the value you'd like to create a subset from.

```
# Enter the column you want to subset
column=''

# Enter the value you want to find in the above column
value=''

# The following pulls the new data subset into a Pandas DataFrame
data_column_subset=data[data[column]==value]

# Output the subset data to a new CSV in the indicated directory
data_column_subset.to_csv(directory+column+"_"+value.replace(" ", "_")+"_"+filename,␣
↪index=False)
```

12.12 Unique Detections ID

Adds the **uniquecid** column to your input file. The uniquecid column assigns every detection record a unique numerical value. This column is needed in order to perform operations, such as filter and compression functions.

The code below will add a unique detection ID column and return the Pandas dataframe.

```
from resonate.uniqueid import add_unqdetecid

input_file = '/path/to/detections.csv'

unqdet_det = add_unqdetecid(input_file);
```

You can use the Pandas `DataFrame.to_csv()` function to output the file to a desired location.

```
unqdet_det.to_csv('/path/to/output.csv', index=False)
```

12.12.1 Unique Detections Function

`uniqueid.add_unqdetecid(input_file, encoding='utf-8-sig')`

Adds the unqdetecid column to an input csv file. The resulting file is returned as a pandas DataFrame object.

Parameters

- **input_file** – Path to the input csv file.
- **encoding** – source encoding for the input file (Default utf8-bom)

Returns padnas DataFrame including unqdetecid column.

12.13 Visual Timeline

This tool takes a detections extract file and generates a Plotly animated timeline, either in place in an iPython notebook or exported out to an HTML file.

Warning: Input files must include `datecollected`, `catalognumber`, `station`, `latitude`, and `longitude` as columns.

```python
from resonate.visual_timeline import timeline
import pandas as pd
detections = pd.read_csv("/path/to/detection.csv")
timeline(detections, "Timeline")
```

12.13.1 Exporting to an HTML File

You can export the map to an HTML file by setting `ipython_display` to `False`.

```python
from resonate.visual_timeline import timeline
import pandas as pd
detections = pd.read_csv("/path/to/detection.csv")
timeline(detections, "Timeline", ipython_display=False)
```

12.13.2 Mapbox

Alternatively you can use a Mapbox access token plot your map. Mapbox is much for responsive than standard Scattergeo plot.

```python
from resonate.visual_timeline import timeline
import pandas as pd

mapbox_access_token = 'YOUR MAPBOX ACCESS TOKEN HERE'
detections = pd.read_csv("/path/to/detection.csv")
timeline(detections, "Title", mapbox_token=mapbox_access_token)
```

12.13.3 Example Output

Below is the sample output for blue sharks off of the coast of Nova Scotia, without using Mapbox.

12.13.4 Visual Timeline Functions

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index