Package sampley

version 0.0.9

August 2025

Jonathan Syme

Authors

The sampley package has been developed and is maintained by Jonathan Syme[1] with the assistance of Daniel Pendleton[1, 2], Nicholas Record[1], and Benjamin Tupper[1].

Any enquiries or queries should be addressed to Jonathan Syme at jsyme@bigelow.org.

[1]Bigelow Laboratory for Ocean Sciences, East Boothbay, ME, USA

[2]NOAA Northeast Fisheries Science Center, Woods Hole, MA, USA

Description

The sampley package serves to sample survey data (hence 'sampley'; sample + survey). By 'survey data', we refer principally to systematic visual survey data, although the sampley package may also be applicable to other kinds of survey data. By 'sample', we mean process those data to produce distinct samples that can then be input to a model. For more information on, please consult the General Notes section of this User Manual.

Contents

General Notes

The grid, segment, and point approaches

In some studies, surveys are conducted throughout a given area, either along transects or at random, with sampling units created afterwards by dividing the survey effort and detections spatially and temporally. In general, this division is achieved by one of three main approaches — referred to here as the grid approach, the segment approach, and the point approach — each with its own variations.

The grid approach consists of overlaying a grid, typically rectangular or hexagonal, onto the study area and allocating detections and, optionally, survey effort to the cells that they lie within. Additionally, data may be allocated to temporal periods. Each cell within a given period then serves as a sample.

The segment approach involves taking sections of continuous, uniform survey effort and cutting them into segments of standardised lengths which serve as samples.

The point approach consists of using the detections, or a subset thereof, as presences and then sampling absences from absence zones (i.e., areas that were surveyed but where the species was not detected).

Data

The `sampley` package has been developed to process data collected during systematic visual surveys for species great and small (e.g., line-transect surveys) into a form that can be input to a model (e.g., a species distribution model). Nevertheless, it may also be applicable to other kinds of survey data (e.g., non-systematic surveys or acoustic surveys), assuming they meet the following requirements.

At a base level, what is required is a survey track (i.e., the route taken by the observer(s) during a survey) and datapoints that were collected during that survey (e.g., sightings or in-situ recordings of environmental variables). As the data are collected during a survey, they should be 'on-line' (i.e., on the survey track), with the potential exception of sightings, which may be 'off-line' if their location is determined, for example, via distance sampling.

The `sampley` package is not meant for extracting or processing data from other sources (e.g., sea surface temperature data from a satellite database or depth measurements from a bathymetry layer), although it does provide the parameters necessary to facilitate these steps once sampling has been completed.

The `sampley` package does not provide custom functionality to filter or subset data, although standard Python functionality is available. Accordingly, it is recommended to only input those data that are to be processed and to quality check data and remove any unwanted datapoints (e.g., off-effort survey track, sightings with low certainty) beforehand.

Stages

The process followed when implementing the sampley package follows three stages:

Stage 1 – inputting

Stage 2 – delimiting

Stage 3 – sampling

The manual is structured to follow these three stages with more information on each stage found in each corresponding section of the manual.

Classes and Objects

The sampley package contains 9 custom classes. Some or all of these classes may be used during processing, depending on the user's requirements. The classes are listed below according to the stage in which they are made:

Stage 1: DataPoints, Sections

Stage 2: Periods, Cells, Segments, Presences, PresenceZones, Absences

Stage 3: Samples

More detail on each class is given under their respective Stage subheadings.

In this manual, an instance of a class is referred to as an object (e.g., an instance of a DataPoints class is referred to as a DataPoints object). Objects are always written with an initial uppercase, while the values that they contain have the same name but are written with lowercase (e.g., 'Segments' refers to a Segments object, whereas 'segments' refers to the segments themselves).

Attributes

Each object has a *name* attribute and a *parameters* attribute which contain the name of the object and key parameters relating to it, respectively.

Most objects have a third attribute called the same as the class but in lower case (e.g., a DataPoints object has a *datapoints* attribute, a Cells object has a *cells* attribute) that contains the values within a dataframe (specifically, a pandas.DataFrame or a geopandas.GeoDataFrame).

Within this dataframe, each row is a value of that type (e.g., a DataPoints object has a datapoints attribute which is a dataframe where each row is a datapoint, a Segments object has a segments attribute which is a dataframe where each row is a segment).

To access a given attribute, use the object's name, a full stop, and the attribute's name (e.g., to access the parameters attribute of an object called u_datapoints, use u_datapoints.parameters).

Further detail on attributes for each class is given later in the manual.

Methods

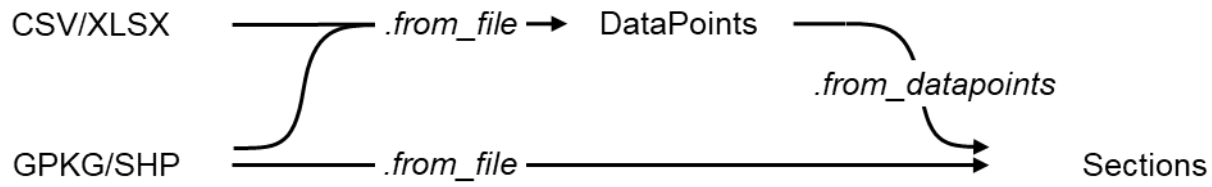
Each class has methods that can be divided into two categories: class methods and standard methods.

Class methods are used to make an object of that class. For example, to make a DataPoints object, the user can use the *from_file* class method. Broadly speaking, the classes in Stage 1 have a *from_file* class method, those in Stage 2 have a *delimit* class method, while the Samples class has various class methods depending on the approach used.

Standard methods are used to apply a function to an object that has already been made. The standard methods vary between classes though most have a *plot* and a *save* method to plot and save the object, respectively.

Paths

Throughout the manual, there are figures that illustrate the possible paths that can be taken to conduct processing, such as the example below:



For all these figures, please note the following.

Objects are written in normal font (e.g., DataPoints, Sections), class methods for making those objects and standard methods for modifying them are written in italics (e.g., *.from_file*, *.from_datapoints*). The methods belong to the object that their arrow points to (e.g., *.from_datapoints* belongs to Sections).

Lines indicate which methods each object is input to or output from. If two lines merge before entering a method, then either of those objects can be input, if they enter separately, then both must be input. In addition to the input objects, each method will require values for several parameters that are not included in the diagram but are detailed in the corresponding sections in the manual.

These paths can be used to determine which objects and methods are required for data processing. For example, the above figure shows, amongst other things, that a GeoPackage (GPKG) can be input to the *from_file* method to make a Sections object.

Output

To facilitate the saving of objects, an output folder should be created where all output objects will be saved. The name of each saved file will be derived automatically from the name of the object.

Stages

Stage 1 — inputting data

To conduct processing, each input file has to be wrangled into one of two classes of objects: Sections and/or DataPoints. In most scenarios, a Sections object and at least one DataPoints object will be necessary to conduct processing. Once any Sections and/or DataPoints objects have been made, they will form the basis from which all subsequent objects are derived. Moreover, any Sections and/or DataPoints objects will not change during processing. This allows for an unlimited number of objects to be derived from them (e.g., if the user wants to process the data via more than one approach).

Input filetypes

The input files can be any of the following types: GeoPackage (GPKG), Shapefile (SHP), Comma-separated values (CSV), or Microsoft Excel (XLSX) — although use of GPKGs and CSVs is recommended. The input process, as well as all subsequent processing, is the same for all filetypes.

Coordinate Reference System (CRS)

For processing, it is necessary for all data to be in a projected CRS, preferably one that preserves distance and uses metres. If the input files are not in an appropriate CRS, they can be reprojected during input. Please note that objects with different CRSs will be incompatible.

Timezone (TZ)

Input files may have datetimes, dates, or no temporal data, although some options will not be possible if temporal data are absent. If input files have datetimes or dates, timezones can be set and, additionally, datetimes can be converted to a different timezone during input. Please note that objects with different timezones will be incompatible.

Paths

A DataPoints object can be made from a CSV, XLSX, GPKG, or SHP file by using the DataPoints.from_file class method (Fig 1). A Sections object can be made from a GPKG or SHP file by using the Sections.from_file class method (Fig 1). Alternatively, a Sections object can be made from a DataPoints object by using the Sections.from_datapoints class method (Fig 1).

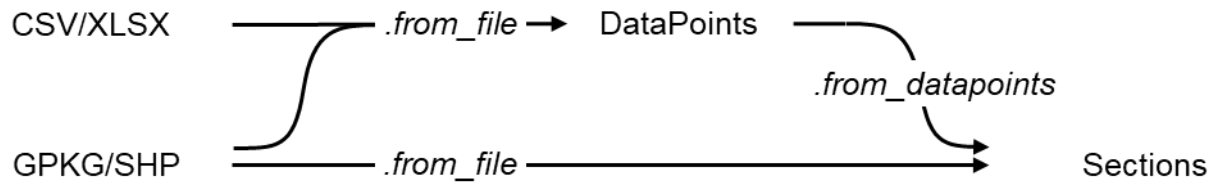


Fig 1. A graphical representation of Stage 1.

DataPoints

A DataPoints object is used for storing datapoints, where each datapoint is simply a point (in space and time) with associated data. For example, each datapoint could be a sighting with information on the number of individuals, a recording of environmental data such as Beaufort sea state and wind speed, or even a simple GPS point with nothing more than latitude and longitude.

The datapoints can be either continuous or sporadic. Continuous datapoints are recorded at frequent, regular intervals and so can be used to construct a trackline. For example, points recorded by a GPS every 10 seconds can be considered continuous. Sporadic datapoints are recorded at infrequent, irregular intervals and so cannot be used to construct a trackline. For example, sightings should typically be considered sporadic, as should environmental datapoints recorded only at the beginning and end of transects. The difference between continuous and sporadic datapoints does not affect how a given DataPoints object is made, but it does affect what can be done with that DataPoints object and so should be kept in mind.

Attributes

datapoints: a dataframe containing the datapoints

A geopandas.GeoDataFrame with the following key columns:

datapoint_id: a unique identifier for each datapoint

geometry: the geometry of each datapoint as a shapely.Point

datetime: the datetime of each point as a pd.Timestamp (or None if no datetimes are entered)

Additionally, the dataframe may contain any number of additional ‘data columns’ containing data of any type.

name: the name of the DataPoints object

The name of the DataPoints object as a string consisting of 'datapoints-' and the name of the input file.

parameters: the parameters of the DataPoints object

Various parameters pertaining to the DataPoints object contained in a dictionary.

Class methods

from_file: make a DataPoints object from a GPKG, SHP, CSV, or XLSX file.

Takes as input a GPKG, SHP, CSV, or XLSX file that contains the datapoints and reformats it for subsequent processing by: renaming and reordering essential columns; if necessary, reprojecting it to a projected CRS; assigning each datapoint a unique ID.

If loading data from a CSV or XLSX, locations of datapoints must be stored in one of two ways:

two columns (x_col and y_col) containing x and y (e.g., longitude and latitude) coordinates

one column (geometry_col) containing points as WKT geometry objects

Parameters:

filepath : str

The path to the file containing the datapoints. Ensure that filepath includes the filename and the extension.

x_col : str, optional, default 'lon'

If inputting a CSV or XLSX with x and y coordinates, the name of the column containing the x coordinate (e.g., longitude) of each datapoint.

y_col : str, optional, default 'lat'

If inputting a CSV or XLSX with x and y coordinates, the name of the column containing the y coordinate (e.g., latitude) of each datapoint.

geometry_col : str, optional, default None

If inputting a CSV or XLSX with points as WKT geometry objects, the name of the column containing the WKT geometry objects.

crs_input : str | int | pyproj.CRS, optional, default None

If inputting a CSV or XLSX, the CRS of the coordinates/geometries. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

datetime_col : str, optional, default None

If applicable, the name of the column containing the datetime or date of each datapoint.

datetime_format : str, optional, default None

Optionally, the format of the datetimes as a string (e.g., "%Y-%m-%d %H:%M:%S").

It is possible to use format="ISO8601" if the datetimes meet ISO8601 (units in greatest to least order, e.g., YYYY-MM-DD) or format="mixed" if the datetimes have different formats (although not recommended as slow and risky).

tz_input : str | timezone | pytz.BaseTzInfo, optional, default None

If datetime_col is specified, the timezone of the datetimes contained within the column. The timezone must be either: a datetime.timezone; a string of a UTC code (e.g., 'UTC+02:00', 'UTC-09:30'); or a string of a timezone name accepted by pytz (e.g., 'Europe/Vilnius' or 'Pacific/Marquesas').

tz_working : str | timezone | pytz.BaseTzInfo, optional, default None

The timezone to be used for the subsequent processing. The timezone must be either: a datetime.timezone; a string of a UTC code (e.g., 'UTC+02:00', 'UTC-09:30'); or a string of a

timezone name accepted by pytz (e.g., ‘Europe/Vilnius’ or ‘Pacific/Marquesas’). Note that tz_input must be specified if tz_working is specified.

datapoint_id_col : str, optional, default None

If applicable, the name of the column containing the datapoint IDs. The datapoint IDs must be unique.

section_id_col : str, optional, default None

If subsequently using Sections.from_datapoints, the name of the column containing the section IDs. Each individual section must have its own unique ID. All the datapoints that make up a given section must have the same value for section ID so that, when Sections.from_datapoints is run, they are grouped together to form a LineString. It is recommended that section IDs be codes consisting of letters and numbers and, optionally, underscores (e.g., ‘s001‘ or 20250710_s01‘).

Returns:

DataPoints

Returns a DataPoints object with three attributes: name, parameters, and datapoints.

open: open a saved DataPoints object

Open a DataPoints object that has previously been saved with DataPoints.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the DataPoints object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., ‘EPSG:4326’); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

tz_working : str | timezone | pytz.BaseTzInfo, optional, default None

The timezone to be used for the subsequent processing. The timezone must be either: a datetime.timezone; a string of a UTC code (e.g., ‘UTC+02:00’, ‘UTC-09:30’); or a string of a timezone name accepted by pytz (e.g., ‘Europe/Vilnius’ or ‘Pacific/Marquesas’).

Methods

plot: plot the datapoints

Makes a basic matplotlib plot of the datapoints in greyscale.

Parameters:

sections : Sections, optional, default None

Optionally, a Sections object with sections to be plotted with the datapoints.

save: save the datapoints

Saves the datapoints GeoDataFrame as a GPKG. The name of the saved file will be the name of the DataPoints object. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the datapoints to before saving (only reprojects the datapoints that are saved and not the DataPoints object).

tz_output : str | timezone | pytz.BaseTzInfo, optional, default None

The timezone to convert the datapoints to before saving (only converts the datapoints that are saved and not the DataPoints object).

Sections

A Sections object is used for storing sections, where each section is a stretch of continuous, uniform survey track.

Defining sections

It is up to the user to define what constitutes a section and to designate the sections before processing with `sampley` (see below for more info on how to designate sections).

The key feature of sections is that they are continuous (i.e., without gaps). For example, a survey may initially result in a single long track, however, portions of this track are then removed as they do not meet certain requirements (e.g., weather conditions are poor or observers are off-watch). Now, what remains is a series of pieces of track with gaps in between. This series of pieces and gaps is not continuous and so cannot constitute a section, however, each individual piece is continuous and so can be a section.

It is also often desirable that sections be uniform, although what constitutes 'uniform' will vary between studies. Broadly speaking, uniformity is often considered in terms of parameters such as speed, observation effort, and weather conditions. It is up to the user to decide which parameters they deem relevant and to use them to define sections before inputting data to `sampley`.

Designating sections (and section IDs)

If the sections are made from a GPKG or a SHP (with Sections.from_file), it is recommended that each section be represented by its own feature (e.g., a single linestring). Nevertheless, having multiple sections within a single feature (e.g., in a multi-linestring) is also allowed. Each individual section can have its own pre-defined unique section ID contained within the GPKG/SHP or, if not, section IDs will be automatically generated during input.

If the sections are made via a DataPoints object (with Sections.from_datapoints), it is necessary to have a section ID column in the input file specifying which section each datapoint belongs to. Each individual section must have its own unique ID. Moreover, all the datapoints that make up a given section must have the same value for section ID so that, when Sections.from_datapoints is run, they are grouped together to form a LineString.

In either case (using Sections.from_file or Sections.from_datapoints), it is recommended that section IDs be codes consisting of letters and numbers and, optionally, underscores (e.g., 's001' or 20250710_s01').

Attributes

sections: a dataframe containing the sections

A geopandas.GeoDataFrame with the following key columns:

section_id: a unique identifier for each section

geometry: the geometry of each section as a shapely.LineString

datetime: the datetime of the section as a pd.Timestamp

name: the name of the Sections object

The name of the Sections object as a string consisting of 'sections-' and the name of the input file.

parameters: the parameters of the Sections object

Various parameters pertaining to the Sections object contained in a dictionary.

Class methods

from_file: make a Sections object from a GPKG or SHP file.

Takes as input a GPKG or SHP file that contains the sections as shapely.LineStrings and reformats it for subsequent processing by: renaming and reordering essential columns; if necessary, reprojecting it to a projected CRS; assigning each section a unique ID.

Sections can be made from CSV or XLSX files containing series of points by first making a DataPoints object and then using Sections.from_datapoints to make a Sections object.

Parameters:

filepath : str

The path to the file containing the sections. Ensure that filepath includes the filename and the extension.

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., ‘EPSG:4326’); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

datetime_col : str, optional, default None

The name of the column containing the datetime of each section.

datetime_format : str, optional, default None

Optionally, the format of the datetimes as a string (e.g., "%Y-%m-%d %H:%M:%S").

It is possible to use format="ISO8601" if the datetimes meet ISO8601 (units in greatest to least order, e.g., YYYY-MM-DD) or format="mixed" if the datetimes have different formats (although not recommended as slow and risky).

tz_input : str | timezone | pytz.BaseTzInfo, optional, default None

If datetime_col is specified, the timezone of the datetimes contained within the column. The timezone must be either: a datetime.timezone; a string of a UTC code (e.g., ‘UTC+02:00’, ‘UTC-09:30’); or a string of a timezone name accepted by pytz (e.g., ‘Europe/Vilnius’ or ‘Pacific/Marquesas’).

tz_working : str | timezone | pytz.BaseTzInfo, optional, default None

The timezone to be used for the subsequent processing. The timezone must be either: a datetime.timezone; a string of a UTC code (e.g., ‘UTC+02:00’, ‘UTC-09:30’); or a string of a

timezone name accepted by pytz (e.g., ‘Europe/Vilnius’ or ‘Pacific/Marquesas’). Note that tz_input must be specified if tz_working is specified.

section_id_col : str, optional, default None

Optionally, the name of the column containing the section IDs. Each individual section must have its own unique ID. It is recommended that section IDs be codes consisting of letters and numbers and, optionally, underscores (e.g., ‘s001‘ or 20250710_s01‘).

Returns:

Sections

Returns a Sections object with three attributes: name, parameters, and sections.

from_datapoints: make a Sections object from a DataPoints object.

Takes as input a DataPoints object that contains sections as continuous series of Points and reformats it for subsequent processing by: converting each series of Points to a LineString; renaming and reordering essential columns. The CRS and timezone will be that of the DataPoints object.

Note that, when making the DataPoints object, it is necessary to specify section_id_col. Please see the documentation for Sections or for the section_id_col parameter under DataPoints.from_file for more information on section IDs and how they should be formatted.

Note that Sections.from_datapoints should only be used with continuous datapoints and not with sporadic datapoints (please see under DataPoints for details on continuous and sporadic datapoints).

Parameters:

datapoints : DataPoints

The DataPoints object that contains sections as series of points.

cols : dict | None, optional, default None

A dictionary whose keys are the names of any columns to keep and whose values are corresponding functions specifying what to do with those columns. For example, if each section has a pre-set period in a column called 'season', cols could be specified as {'season': 'first'} to keep the first value of season for each section.

sortby : str | list, optional, default None

When converting each series of Points to a LineString, the Points are joined one to the next (like a dot-to-dot). If the Points are not in the right order, the resulting LineString will be incorrect. If sortby is not specified, the Points will be joined in the order that they are in. To change this order, specify sortby as the name of a column or columns (e.g., 'datetime') to sort the datapoints by before the Points are converted to LineStrings.

Returns:

Sections

Returns a Sections object with three attributes: name, parameters, and sections.

open: open a saved Sections object

Open a Sections object that has previously been saved with Sections.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the Sections object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

tz_working : str | timezone | pytz.BaseTzInfo, optional, default None

The timezone to be used for the subsequent processing. The timezone must be either: a datetime.timezone; a string of a UTC code (e.g., 'UTC+02:00', 'UTC-09:30'); or a string of a timezone name accepted by pytz (e.g., 'Europe/Vilnius' or 'Pacific/Marquesas').

Methods

plot: plot the sections

Makes a basic matplotlib plot of the sections in greyscale.

Parameters:

datapoints : DataPoints, optional, default None

Optionally, a DataPoints object with datapoints to be plotted with the sections.

save: save the sections

Saves the sections GeoDataFrame as a GPKG. The name of the saved file will be the name of the Sections object. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the sections to before saving (only reprojects the sections that are saved and not the Sections object).

tz_output : str | timezone | pytz.BaseTzInfo, optional, default None

The timezone to convert the sections to before saving (only converts the sections that are saved and not the Sections object).

Stage 2 – delimiting

Stage 2 involves delimiting one or more of the following: periods, cells, segments, presences, presences zones, and absences. The approach(es) used and the data input will determine which of these are necessary to delimit. The result of this stage is one or more objects that are referred to collectively as 'delimiters', i.e., objects that delimit samples (e.g., grid cells set the spatial limits of samples). There is no limit to the number of delimiters that can be made, thus enabling the user to implement different procedures and settings to determine the best option for data processing.

Paths

There are three sets of paths corresponding to the three approaches: grid, segment, and point.

Grid approach: A Cells object can be made from a DataPoints or a Sections object by using the Cells.delimit class method (Fig 2a). Similarly, a Periods object can be made from a DataPoints or a Sections object by using the Periods.delimit class method (Fig 2a).

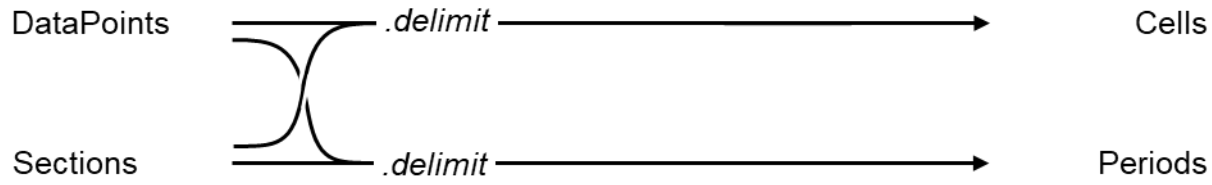


Fig 2a. A graphical representation of Stage 2 – grid approach.

Segment approach: A Segments object can be made from a Sections object by using the Segments.delimit class method (Fig 2b).

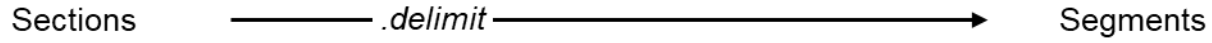


Fig 2b. A graphical representation of Stage 2 – segment approach.

Point approach: A Presences object can be made from a DataPoints object by using the Presences.delimit class method and, additionally, can be thinned with the Presences.thin method (Fig 2c). An PresenceZones object can be made from a Presences object and a Sections object by using the PresenceZones.delimit class method (Fig 2c). An Absences object can then be made from an PresenceZones object and, additionally, can be thinned with the Absences.thin method (Fig 2c).

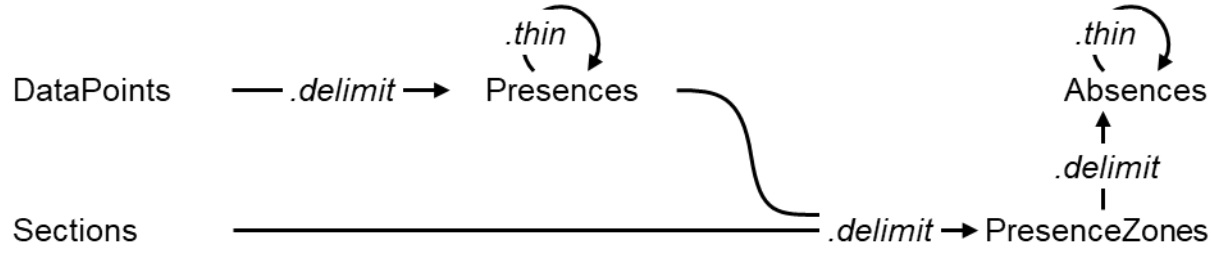


Fig 2c. A graphical representation of Stage 2 – point approach.

Periods

Attributes

periods: a dataframe containing the periods

A pandas.DataFrame with the following key columns:

period_id: a unique identifier for each period consisting of 'p', the begin date, a dash, the number of units, and the unit (e.g., 'p2025-01-22-8d' is an 8 day period beginning the 22nd Jan 2025).

date_beg: the date of the beginning of each period as a pd.Timestamp

date_mid: the date of the midpoint of each period as a pd.Timestamp

date_end: the date of the end of each period as a pd.Timestamp

name: the name of the Periods object

The name of the Periods object as a string consisting of 'periods-', the number of units, and the units (e.g., 'periods-8d', meaning periods of 8 days).

parameters: the parameters of the Periods object

Various parameters pertaining to the Periods object contained in a dictionary.

Class methods

delimit: delimit temporal periods of a set number of units

From a given extent, number of units, and type of units, delimit temporal periods of regular length, e.g., 8 days, 2 months, or 1 year.

Temporal periods of irregular length (e.g., seasons) should be predefined and contained within a column of the input data.

Parameters:

extent : Sections | DataPoints | pandas.DataFrame | tuple[list, str]

An object detailing the temporal extent over which the periods will be limited. Must be one of:

a Sections object whose sections GeoDataFrame has a 'datetime' column

a DataPoints object whose datapoints GeoDataFrame has a 'datetime' column

a pandas.DataFrame that has a 'datetime' column

a tuple containing two elements: a list of two datetimes and a timezone as a string (or None if no timezone is to be used)

num : int | float

The number of temporal units.

unit : str

The temporal units assigned with one of the following strings:

'day': days ('d' also accepted)

'month': months ('m' also accepted)

'year': years ('y' also accepted)

Returns:

Periods

Returns a Periods object with three attributes: name, parameters, and periods.

open: open a saved Periods object

Open a Periods object that has previously been saved with Periods.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the Periods object that was saved (without the extension).

Methods

save: save the periods

Saves the periods DataFrame as a CSV. The name of the saved file will be the name of the Periods object. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

Cells

Attributes

cells: a dataframe containing the cells

A geopandas.GeoDataFrame with the following key columns:

cell_id: a unique identifier for each cell consisting of 'c', a unique number, a dash, 'r'/'h' for rectangular/hexagonal, the side length, and the units (e.g., 'c123-r5000m' is cell number 123 that is rectangular with 5000 metre sides).

polygon: the geometry of each cell as a shapely.Polygon

centroid: the centroid of each cell as a shapely.Point

name: the name of the Cells object

The name of the Cells object as a string consisting of 'cells-', 'r' / 'h' for rectangular / hexagonal and the side length (e.g., 'cells-r5000m', meaning cells that are rectangular with 5000 metre sides).

parameters: the parameters of the Cells object

Various parameters pertaining to the Cells object contained in a dictionary.

Class methods

delimit: delimit grid cells

From a given extent, variation, and side length, delimit rectangular or hexagonal grid cells of a regular size.

Parameters:

extent : Sections | DataPoints | geopandas.GeoDataFrame | tuple[list, str]

An object detailing the spatial extent over which the periods will be limited. Must be one of:

a Sections object

a DataPoints object

a geopandas.GeoDataFrame

a tuple containing two elements: a list containing the x min, y min, x max, and y max and a CRS

var : {'rectangular', 'hexagonal'}

The variation used to generate the cells. Must be one of the following:

'rectangular': make rectangular (square) cells ('r' also accepted)

'hexagonal': make hexagonal cells ('h' also accepted)

side : int | float

The side length of the rectangles/hexagons in the units of the CRS.

buffer : int | float, optional, default 0

The width of a buffer to be created around the extent to enlarge it and ensure that all the surveyed area is covered by the cells.

Returns:

Cells

Returns a Cells object with three attributes: name, parameters, and cells.

open: open a saved Cells object

Open a Cells object that has previously been saved with Cells.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the Cells object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Methods

plot: plot the cells

Makes a basic matplotlib plot of the cells.

Parameters:

datapoints : DataPoints, optional, default None

Optionally, a DataPoints object with datapoints to be plotted with the cells.

sections : Sections, optional, default None

Optionally, a Sections object with sections to be plotted with the cells.

save: save the cells

Saves the cells GeoDataFrame as two GPKGs: one of the polygons and one of the centroids. The names of the saved files will be the name of the Cells object plus '-polygons' and '-centroids', respectively. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the cells to before saving (only reprojects the cells that are saved and not the Cells object). The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Segments

Attributes

segments: a dataframe containing the segments

A geopandas.GeoDataFrame with the following key columns:

segment_id: a unique identifier for each segments

segment_id: the ID of each segment consisting of 's', a unique number, a dash, 's'/'j'/'r' for simple/joining/redistribution variation, the target length, and the units (e.g., 's123-r1000m' is segment 123 made with the redistribution method and a target length of 1000 metres).

line: the geometry of the segment as a shapely.LineString

midpoint: the midpoint of the segment as a shapely.Point

section_id: the ID of the section from which the segment was cut

dfbsec_beg: the distance along the section where the segment begins

dfbsec_end: the distance along the section where the segment ends

name: the name of the Segments object

The name of the Segments object as a string consisting of 'segments-', 's' / 'j' / 'r' for simple / joining / redistribution variation, the target length, and the units (e.g., 'segments-r1000m', meaning segments made with the redistribution method and a target length of 1000 metres).

parameters: the parameters of the Segments object

Various parameters pertaining to the Segments object contained in a dictionary.

Class methods

delimit: delimit segments

With a given variation and target length, cut sections into segments.

Segments can be made with any one of three variations: the simple, joining, and redistribution variations. For all three variations, a target length is set. The variations differ in how they deal with the remainder – the length inevitably left over after dividing a section by the target length. Additionally, for the simple and joining variations, the location of the remainder / joined segment can be randomised (rather than always being at the end).

Parameters:

sections : Sections

The Sections object containing the sections from which the segments will be cut.

var : {'simple', 'joining', 'redistribution'}

The variation to use to make the segments. Must be one of the following:

'simple': the remainder is left as an independent segment ('s' also accepted)

'joining': the remainder, if under half the target length, is joined to another segment, otherwise it is left as an independent segment ('j' also accepted)

'redistribution': the length of the remainder is redistributed among all segments ('r' also accepted)

target : int | float

The target length of the segments in the units of the CRS.

rand : bool, optional, default False

If using the simple or joining variations, whether to randomise the location of the remainder / joined segment or not.

Returns:

Segments

Returns a Segments object with three attributes: name, parameters, and segments.

open: open a saved Segments object

Open a Segments object that has previously been saved with Segments.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the Segments object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Methods

datetimes: get datetimes for the beginning, middle, and end of each segment

Get a datetime value for the beginning, middle, and end of each segment. This is only applicable to segments that were made from sections that were made from continuous datapoints with Sections.from_datapoints. Additionally, it requires that those datapoints have datetime values.

In the (likely) case that a segment begins/ends at some point between two datapoints, the begin/end time for that segment will be interpolated based on the distance from those two datapoints to the point at which the segment begins/ends assuming a constant speed.

Parameters:

datapoints : DataPoints

The DataPoints object, containing datetimes, that was used to make the Sections object that was used to make the Segments object.

plot: plot the segments

Makes a basic matplotlib plot of the segments.

Parameters:

datapoints : DataPoints, optional, default None

Optionally, a DataPoints object with datapoints to be plotted with the segments.

sections : Sections, optional, default None

Optionally, a Sections object with sections to be plotted with the segments.

save: save the segments

Saves the segments GeoDataFrame as two GPKGs: one of the lines and one of the midpoints. The names of the saved files will be the name of the Segments object plus '-lines' and '-midpoints', respectively. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the segments to before saving (only reprojects the segments that are saved and not the Segments object). The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Presences

Attributes

Unlike the other classes, in Presences objects, the presences are not contained within an attribute called presences (as may be expected), but rather they are contained within three separate attributes: full, kept, and removed.

full: a dataframe containing all the presences

A geopandas.GeoDataFrame containing all the presences with the following key columns:

point_id: a unique identifier for each presence

point: the presence as a shapely.Point

date: the date of the presence as a pd.Timestamp

kept: a dataframe of the presences kept after spatiotemporal thinning

A geopandas.GeoDataFrame with the same columns as full but only those presences kept after spatiotemporal thinning.

removed: a dataframe of the presences removed after spatiotemporal thinning

A geopandas.GeoDataFrame with the same columns as full but only those presences removed after spatiotemporal thinning (i.e., the complement to kept).

name: the name of the Presences object

The name of the Presences object as a string consisting of 'presences-' and the name of the DataPoints object from which the Presences object is derived.

parameters: the parameters of the Presences object

Various parameters pertaining to the Presences object contained in a dictionary.

Class methods

delimit: delimit presences

From a DataPoints object, make a Presences object.

There are two options for the datapoints: all rows are presences, in which case there is no need to specify presence_col, or only some rows are presences, in which case presence_col must be specified.

Parameters:

datapoints : DataPoints

The DataPoints object that contains the presences.

presence_col : str, optional, default None

The name of the column containing the values that determine which points are presences (e.g., a column containing a count of individuals). This column must contain only integers or floats. Only needs to be specified if the DataPoints object includes points that are not presences.

Returns:

Presences

Returns a Presences object with three attributes: name, parameters and full.

open: open a saved Presences object

Open a Presences object that has previously been saved with Presences.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the Presences object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Methods

thin: spatiotemporally thin the presences

Spatiotemporally thin the presences so that no two presences are within some spatial threshold and/or within some temporal threshold of each other.

If only a spatial threshold is specified, spatial thinning will be conducted. If only a temporal threshold is specified, temporal thinning will be conducted. If both a spatial and a temporal threshold are specified, spatiotemporal thinning will be conducted.

Adds to the Presences object two attributes — Presences.kept and Presences.removed — that are both geopandas.GeoDataFrame containing the points that were kept and those that were removed after spatiotemporal thinning, respectively.

Parameters:

sp_threshold : int | float, optional, default None

The spatial threshold to use for spatial and spatiotemporal thinning in the units of the CRS.

tm_threshold : int | float, optional, default None

The temporal threshold to use for temporal and spatiotemporal thinning in the units set with tm_unit.

tm_unit : str, optional, default 'day'

The temporal units to use for temporal and spatiotemporal thinning. One of the following:

'year': year (all datetimes from the same year will be given the same value)

'month': month (all datetimes from the same month and year will be given the same value)

'day': day (all datetimes with the same date will be given the same value)

'hour': hour (all datetimes in the same hour on the same date will be given the same value)

'moy': month of the year (i.e., January is 1, December is 12 regardless of the year)

'doy': day of the year (i.e., January 1st is 1, December 31st is 365 regardless of the year

target : int, optional, default None

The target number of presences. If, after thinning, the number of presences is greater than the target, the kept presences will be randomly sampled to reduce their number to the target.

plot: plot the presences

Makes a basic matplotlib plot of the presences.

Parameters:

sp_threshold : int | float, optional, default None

The spatial threshold used for spatial and spatiotemporal thinning in the units of the CRS. If specified, the plot will add a circle around each point to represent the spatial threshold.

which : { 'full', 'kept', 'removed', 'thinned'}, optional, default 'full'

A keyword to indicate which set of points to plot. Must be one of the following:

'full': all the presences (in blue)

'kept': the presences kept after thinning (in blue)

'removed': the presences removed after thinning (in yellow)

'thinned': the presences kept after thinning (in blue) and those removed after thinning (in yellow)

save: save the presences

Saves the full, kept, and removed presences as GPKG files. The name of the saved files will be the name of the Presences object plus '-full', '-kept', and '-removed', respectively. Note that the kept and removed presences will only be saved if they have been made with Presences.thin. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the presences to before saving (only reprojects the presences that are saved and not the Presences object). The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

PresenceZones

Attributes

presencezones: a dataframe containing the presences zones

A geopandas.GeoDataFrame with the following key columns:

section_id: a unique identifier for the section to which the presence zones correspond (in the case that all sections have identical presence zones, this column will contain a single value: 'all')

presencezones: the geometry of each set of presences zones as a shapely.MultiPolygon

name: the name of the PresenceZones object

The name of the PresenceZones object as a string consisting of 'presencezones-', the spatial threshold, its units, and, if used, the temporal threshold and its units.

parameters: the parameters of the PresenceZones object

Various parameters pertaining to the PresenceZones object contained in a dictionary.

Class methods

delimit: delimit presences zones

From the presences, use a spatial and, optionally, temporal threshold to make presences zones.

Presence zones are zones around presences that are deemed to be ‘occupied’ by the animals. Spatial and temporal thresholds determine the extent of these occupied zones.

Additionally, the presence zones correspond to sections – specifically, the sections that they overlap spatially and, optionally, temporally with, as determined by the spatial and temporal thresholds.

Parameters:

presences : Presences

The Presences object containing the presences from which the presences zones are to be made.

sections : Sections

The Sections object containing the sections to which the presences zones correspond.

sp_threshold : int | float, optional, default None

The spatial threshold to use for making the presences zones in the units of the CRS.

tm_threshold : int | float, optional, default None

The temporal threshold to use for making the presences zones in the units set with tm_unit.

tm_unit : str, optional, default 'day'

The temporal units to use for making the presences zones. One of the following:

'year': year (all datetimes from the same year will be given the same value)

'month': month (all datetimes from the same month and year will be given the same value)

'day': day (all datetimes with the same date will be given the same value)

'hour': hour (all datetimes in the same hour on the same date will be given the same value)

'moy': month of the year (i.e., January is 1, December is 12 regardless of the year)

'doy': day of the year (i.e., January 1st is 1, December 31st is 365 regardless of the year

Returns:

PresenceZones

Returns a PresenceZones object with three attributes: name, parameters, and presencezones.

open: open a saved PresenceZones object

Open an PresenceZones object that has previously been saved with PresenceZones.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the PresenceZones object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Methods

plot: plot the presences zones

Makes a basic matplotlib plot of the presences zones.

Parameters:

sections : Sections, optional, default None

Optionally, a Sections object with sections to be plotted with the presences zones.

presences : Presences, optional, default None

Optionally, a Presences object with presences to be plotted with the presences zones.

save: save the presences zones

Saves the presences zones GeoDataFrame as a GPKG. The name of the saved file will be the name of the PresenceZones object. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the presences zones to before saving (only reprojects the presences zones that are saved and not the PresenceZones object). The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Absences

Attributes

Unlike the other classes, in Absences objects, the absences are not contained within an attribute called absences (as may be expected), but rather they are contained within three separate attributes: full, kept, and removed.

full: a dataframe containing all the absences

A geopandas.GeoDataFrame containing all the absences with the following key columns:

point_id: a unique identifier for each absence

point: the absence as a shapely.Point

date: the date of the absence as a pd.Timestamp

Additionally, if the 'from-the-line' variation is used:

point_al: a point corresponding to each point in the 'point' column but along the survey line

kept: a dataframe of the absences kept after spatiotemporal thinning

A geopandas.GeoDataFrame with the same columns as full but only those absences kept after spatiotemporal thinning.

removed: a dataframe of the absences removed after spatiotemporal thinning

A geopandas.GeoDataFrame with the same columns as full but only those absences removed after spatiotemporal thinning (i.e., the complement to kept).

name: the name of the Absences object

The name of the Absences object as a string consisting of 'absences-', 'a' / 'f' for along-the-line / from-the-line variation, and the name of the PresenceZones object.

parameters: the parameters of the Absences object

Various parameters pertaining to the Absences object contained in a dictionary.

Class methods

delimit: delimit the absences

Absences can be generated by one of two variations: the 'along-the-line' variation or the 'from-the-line' variation.

In the along-the-line variation, each absence is generated by randomly placing a point along the survey track, provided it is not within the corresponding presences zones.

In the from-the-line variation, each absence is generated by randomly placing a point along the survey track and then placing a second point a certain distance from the first point perpendicular to the track, provided that this second point is not within the corresponding presences zones. The distance from the

track is selected from a list of candidate distances that can be generated in any way, including from a predefined distribution (e.g., a detection function) by using the function generate_dfls.

Parameters:

sections : Sections

The Sections object containing the sections used to generate the absences.

presencezones : PresenceZones

The PresenceZones object containing the presences zones used to generate the absences.

var : {'along', 'from'}

The variation to use to generate the absences. Must be one of the following:

'along': along-the-line - the absences are generated by randomly placing a point along the surveyed lines ('a' also accepted)

'from': from-the-line - the absences are generated by, firstly, randomly placing a point along the line and then, secondly, placing a point a certain distance from the first point perpendicular to the line ('f' also accepted)

target : int | float

The target number of absences to be generated. Note that, during thinning, some absences may be removed so, to account for this, the target should be set higher than the number desired.

dfls : list[int | float], optional, default None

If using the from-the-line variation, a list of candidate distances from the line to use when generating absences. For each absence, one of these distances will be chosen at random and used to place the absence at that distance from the survey line. These distances can be generated in any way, including from a predefined distribution (e.g., a detection function) with the function generate_dfls.

Returns:

Absences

Returns an Absences object with three attributes: name, parameters and full.

open: open a saved Absences object

Open an Absences object that has previously been saved with Absences.save().

Parameters:

folder : str

The path to the folder containing the saved files.

basename : str

The name of the Absences object that was saved (without the extension).

crs_working : str | int | pyproj.CRS, optional, default None

The CRS to be used for the subsequent processing. In most cases, must be a projected CRS that, preferably, preserves distance and uses metres. The CRS must be either: a pyproj.CRS;

a string in a format accepted by pyproj.CRS.from_user_input (e.g., ‘EPSG:4326’); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Methods

thin: spatiotemporally thin the absences

Spatiotemporally thin the absences so that no two absences are within some spatial threshold and/or within some temporal threshold of each other.

If only a spatial threshold is specified, spatial thinning will be conducted. If only a temporal threshold is specified, temporal thinning will be conducted. If both a spatial and a temporal threshold are specified, spatiotemporal thinning will be conducted.

Adds to the Absences object two attributes — Absences.kept and Absences.removed — that are both geopandas.GeoDataFrame containing the points that were kept and those that were removed after spatiotemporal thinning, respectively.

Parameters:

sp_threshold : int | float, optional, default None

The spatial threshold to use for spatial and spatiotemporal thinning in the units of the CRS.

tm_threshold : int | float, optional, default None

The temporal threshold to use for temporal and spatiotemporal thinning in the units set with tm_unit.

tm_unit : str, optional, default 'day'

The temporal units to use for temporal and spatiotemporal thinning. One of the following:

'year': year (all datetimes from the same year will be given the same value)

'month': month (all datetimes from the same month and year will be given the same value)

'day': day (all datetimes with the same date will be given the same value)

'hour': hour (all datetimes in the same hour on the same date will be given the same value)

'moy': month of the year (i.e., January is 1, December is 12 regardless of the year)

'doy': day of the year (i.e., January 1st is 1, December 31st is 365 regardless of the year

target : int, optional, default None

The target number of absences. If, after thinning, the number of absences is greater than the target, the kept absences will be randomly sampled to reduce their number to the target.

plot: plot the absences

Makes a basic matplotlib plot of the absences.

Parameters:

sp_threshold : int | float, optional, default None

The spatial threshold used for spatial and spatiotemporal thinning in the units of the CRS. If specified, the plot will add a circle around each point to represent the spatial threshold.

which : { 'full', 'kept', 'removed', 'thinned'}, optional, default 'full'

A keyword to indicate which set of points to plot. Must be one of the following:

'full': all the absences (in red)

'kept': the absences kept after thinning (in red)

'removed': the absences removed after thinning (in yellow)

'thinned': the absences kept after thinning (in red) and those removed after thinning (in yellow)

presencezones : PresenceZones, optional, default None

Optionally, an PresenceZones object with presences zones to be plotted with the absences.

save: save the absences

Saves the full, kept, and removed absences as GPKG files. The name of the saved files will be the name of the Absences object plus '-full', '-kept', and '-removed', respectively. Note that the kept and removed absences will only be saved if they have been made with Absences.thin. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

crs_output : str | int | pyproj.CRS, optional, default None

The CRS to reproject the absences to before saving (only reprojects the absences that are saved and not the Absences object). The CRS must be either: a pyproj.CRS; a string in a format accepted by pyproj.CRS.from_user_input (e.g., 'EPSG:4326'); or an integer in a format accepted by pyproj.CRS.from_user_input (e.g., 4326).

Stage 3 – sampling

The delimiters made in Stage 2 set the limits of the samples, however they do not contain any data. In this sense, they can be considered as empty containers waiting to be filled. Stage 3 concerns allocating data contained in one or more DataPoints and/or Sections objects to the delimiters (i.e., filling containers with their corresponding data).

Paths

Grid approach: A Samples object can be made from a DataPoints object, a Cells object, and, optionally, a Periods object by using the Samples.grid class method. Additionally, a Samples object with measures of survey effort can be made from a Sections object, a Cells object, and, optionally, a Periods object by using the Samples.grid_se class method. Finally, a Samples object can be made from multiple other Samples objects by using the Samples.merge class method (Fig 3a).

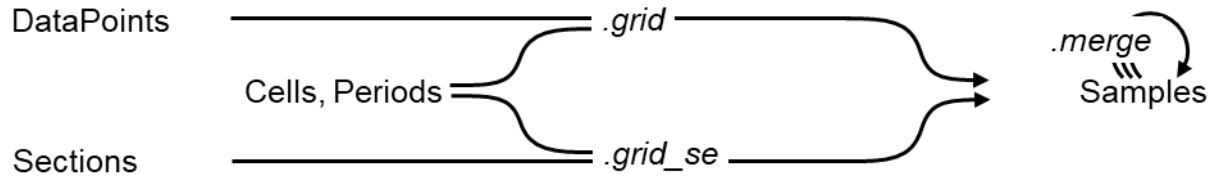


Fig 3a. A graphical representation of Stage 3 – grid approach.

Segment approach: A Samples object can be made from a DataPoints object and a Segments object by using the Samples.segment class method. Additionally, a Samples object with measures of survey effort can be made from a Segments object by using the Samples.segment_se class method. Finally, a Samples object can be made from multiple other Samples objects by using the Samples.merge class method (Fig 3b).

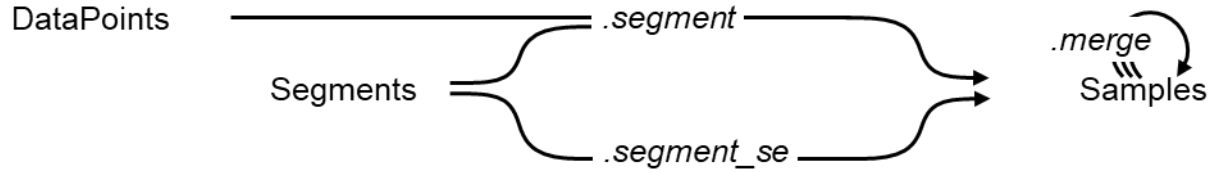


Fig 3b. A graphical representation of Stage 3 – segment approach.

Point approach: A Samples object can be made from a DataPoints object, a Presences object, an Absences object, and, optionally, a Sections object by using the Samples.point class method (Fig 3c).

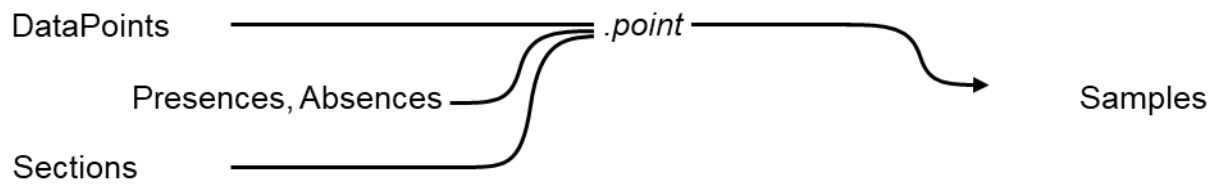


Fig 3c. A graphical representation of Stage 3 – point approach.

Samples

Attributes

samples: a dataframe containing the samples

A geopandas.GeoDataFrame with the key columns of the delimiters (i.e., Periods, Cells, Segments, Presences, or Absences) used to generate the samples as well as any number of data columns.

name: the name of the Samples object

The name of the Samples object as a string consisting of 'samples-' and an amalgamation of the names of the delimiters (i.e., Periods, Cells, Segments, Presences, or Absences) used to generate the samples.

parameters: the parameters of the Samples object

Various parameters pertaining to the Samples object contained in a dictionary.

assigned: the datapoints or sections with their allocated delimiter

Depending on the approach used, a geopandas.GeoDataFrame of the datapoints or sections with an additional column (or columns) indicating the Periods, Cells, or Segments that each datapoint or section has been allocated to. Mainly for the purposes of verifying the allocation process.

Class methods

grid: resample datapoints using the grid approach

Determines which cell and period each datapoint lies within and then groups together datapoints that lie within the same cell and period. As multiple datapoints may lie within the same cell and period, it is necessary to treat them in some way (e.g., average them, sum them). The parameter cols dictates how each column is to be treated.

Parameters:

datapoints : DataPoints

The DataPoints object.

cells : Cells

The Cells object.

periods : Periods | str | None

One of the following:

a Periods object

a string indicating the name of the column in datapoints containing pre-set periods

None

cols : dict

A dictionary indicating how to treat each of the data columns. The dictionary should have the format:

{'COLUMN': FUNCTION,

'COLUMN': FUNCTION}

...where COLUMN is the name of a given column as a string (e.g., 'bss') and FUNCTION is a function to apply to the values in that column when they are grouped together (e.g., 'mean').

Functions include those available for pandas.groupby plus some custom functions provided here. For example:

'mean' - get the mean

'min' - get the minimum

'max' - get the maximum

mode - get the mode

'sum' - sum the values

'count' - count how many have a value (0s counted, NAs ignored)

count_nz - count how many have a value (0s not counted, NAs ignored)

pa - convert numeric values to binary presence-absence (0s not counted, NAs ignored)

list - list the values

Note that some functions have quotation marks, while others do not. Note that some functions differ in how they treat NA values (missing values) and 0s. It is not necessary to specify all columns, but any columns not specified will not be retained. Each column can only be specified once.

full : bool, optional, default False

If False, only those cell-period combinations that have at least one datapoint will be included in samples. If True, all possible cell-period combinations will be included in samples (note that this may result in a large number of samples that have no data).

Returns:

Samples

Returns a Samples object with four attributes: name, parameters, samples, and assigned.

Examples:

For a set of datapoints with a column of counts of individuals, 'individuals', and a column of values for Beaufort sea state (BSS), 'bss', the parameter cols could be set to the following in order to sum the individuals observed per sample and get the mean BSS per sample:

cols={'individuals': 'sum', 'bss': 'mean'}

segment: resample datapoints using the segment approach

Determines which segment each datapoint corresponds to and then groups together datapoints that correspond to the same segment. As multiple datapoints may correspond to the same segment, it is necessary to treat them in some way (e.g., average them, sum them). The parameter cols dictates how each column is to be treated.

Parameters:

datapoints : DataPoints

The DataPoints object.

segments : Segment

The Segments object.

cols : dict

A dictionary indicating how to treat each of the data columns. The dictionary should have the format:

{'COLUMN': FUNCTION,

'COLUMN': FUNCTION}

...where COLUMN is the name of a given column as a string (e.g., 'bss') and FUNCTION is a function to apply to the values in that column when they are grouped together (e.g., 'mean').

Functions include those available for pandas.groupby plus some custom functions provided here. For example:

'mean' - get the mean

'min' - get the minimum

'max' - get the maximum

mode - get the mode

'sum' - sum the values

'count' - count how many have a value (0s counted, NAs ignored)

count_nz - count how many have a value (0s not counted, NAs ignored)

pa - convert numeric values to binary presence-absence (0s not counted, NAs ignored)

list - list the values

Note that some functions have quotation marks, while others do not. Note that some functions differ in how they treat NA values (missing values) and 0s. It is not necessary to specify all columns, but any columns not specified will not be retained. Each column can only be specified once.

how : { 'line', 'midpoint', 'datetime', 'dfb'}

An option specifying how to determine which segment each datapoint corresponds to. Must be one of the following:

line: each datapoint is matched to the nearest segment that has the same date

midpoint: each datapoint is matched to the segment with the nearest midpoint that has the same date

datetime: each datapoint is matched to a segment based on the datetime of the datapoint and the beginning datetimes of the segments (note that Segments.datetimes must be run before; note also that, if multiple surveys are run simultaneously, they will need to be processed separately to avoid datapoints from one survey being allocated to segments from another due to temporal overlap)

dfb: each datapoint is matched to a segment based on the distance it is located from the start of the sections lines (only applicable for matching segments that were made from sections that were made from datapoints with Sections.from_datapoints and those datapoints)

Returns:

Samples

Returns a Samples object with four attributes: name, parameters, samples, and assigned.

Examples:

For a set of datapoints that has a column of counts of individuals, 'individuals', and a column of values for Beaufort sea state (BSS), 'bss', the parameter cols could be set to the following in order to sum the individuals observed per sample and get the mean BSS per sample:

cols={'individuals': 'sum', 'bss': 'mean'}

point: resample datapoints using the point approach

For each presence, gets data from its corresponding datapoint (i.e., the datapoint from which the presence was derived).

Optionally, for each absence, gets the datapoint prior to it and assigns to the absence that datapoint's data. The ID of the prior datapoint is also added to the datapoint_id column. Note that this is only applicable if presences zones were made from sections that were, in turn, made from datapoints with Sections.from_datapoints and those datapoints.

Concatenates the presences and absences and assigns them presence-absence values of 1 and 0, respectively.

Parameters:

datapoints : DataPoints

The DataPoints object.

presences : Presences

The Presences object.

absences : Absences

The Absences object.

cols : list

A list indicating which data columns to add to the presences and, if applicable, the absences.

sections : Sections, optional, default None

If sections is specified, data will be added to the absences. The Sections object must contain the sections from which the absences were derived. Only applicable for absences that were made from sections that were made from datapoints with Sections.from_datapoints and those datapoints.

Returns:

Samples

Returns a Samples object with three attributes: name, parameters, and samples.

grid_se: measure survey effort using the grid approach

Measures the amount of survey track that lies within each cell-period combination to get a measure of survey effort. Survey effort per cell-period can be measured in two ways:

length - length of the survey track in each cell-period

area - area of the buffered survey track in each cell-period

Moreover, each of these ways can be measured using Euclidean or geodesic measurements, as determined by the parameter euc_geo. Geodesic measurements will be more precise but take longer to run. Multiple measures of survey effort can be calculated simultaneously. If the parameter length is True, length will be measured. If the parameter esw is specified, area will be measured.

Parameters:

sections : Sections

The Sections object.

cells : Cells

The Cells object.

periods : Periods | str | None

One of the following:

a Periods object

a string indicating the name of the column in datapoints containing pre-set periods

None

length : bool, optional, default True

If True, the length of survey track in each cell-period combination will be measured.

esw : int | float, optional, default None

Optionally, the one-sided effective stripwidth (ESW). If a value is given, the area of survey track in each cell-period combination will be measured. Note that ESW is one-sided.

euc_geo : {'euclidean', 'geodesic', 'both'}, optional, default 'euclidean'

The type of measurement. Must be one of the following: 'euclidean', 'geodesic', or 'both'.

full : bool, optional, default False

If False, only those cell-period combinations that have at least some survey effort will be included in samples. If True, all possible cell-period combinations will be included in samples (note that this may result in a large number of samples that have no data).

Returns

Samples

Returns a Samples object with four attributes: name, parameters, samples, and assigned. Within the samples attribute, the survey effort measures will be contained in the following columns (if applicable):

se_length: survey effort measured as length with Euclidean distances

se_area: survey effort measured as area with Euclidean distances

se_length_geo: survey effort measured as length with geodesic distances

se_area_geo: survey effort measured as area with geodesic distances

segment_se: measure survey effort using the segment approach

Measures the amount of survey effort per segment. Survey effort per segment can be measured in three ways:

length - length of the segment

area - length of the segment multiplied by a one-sided ESW multiplied by 2

effective area - length of the segment multiplied by a one-sided area under a detection function multiplied by 2

Moreover, each of these ways can be measured using Euclidean or geodesic measurements, as determined by the parameter euc_geo. Geodesic measurements will be more precise but take longer to run. Multiple measures of survey effort can be calculated simultaneously. If the parameter length is True, length will be measured. If the parameter esw is specified, area will be measured.

Parameters:

segments : Segments

The Segments object.

length : bool, optional, default True

If True, the length of each segment will be measured.

esw : int | float, optional, default None

Optionally, the one-sided effective stripwidth (ESW). If a value is given, the area of each segment will be measured. Note that ESW is one-sided.

audf : int | float, optional, default None

Optionally, the one-sided area under detection function (AUDF). If a value is given, the effective area of each segment will be measured. Note that AUDF is one-sided.

euc_geo : {'euclidean', 'geodesic', 'both'}, optional, default 'euclidean'

The type of measurement. Must be one of the following: 'euclidean', 'geodesic', or 'both'.

Returns

Samples

Returns a Samples object with three attributes: name, parameters, and samples. Within the samples attribute, the survey effort measures will be contained in the following columns (if applicable):

se_length: survey effort measured as length with Euclidean distances

se_area: survey effort measured as area with Euclidean distances

se_effective: survey effort measured as effective area with Euclidean distances

se_length_geo: survey effort measured as length with geodesic distances

se_area_geo: survey effort measured as area with geodesic distances

se_effective_geo: survey effort measured as effective area with geodesic distances

merge: merge multiple Samples objects together

Merge multiple Samples objects into a single new Samples object. Each Samples object should be entered as a parameter with a unique name of the user's choosing (note that this name will be used to name the merged Samples object). Only Samples objects made with the grid or segment approach can be merged (i.e., Samples objects must be generated by one or more of Samples.grid, Samples.segment, Samples.grid_se, or Samples.segment_se, but not Samples.point).

Parameters:

**kwargs :

Any number of Samples objects each entered as a parameter with a unique name of the user's choosing.

Returns:

Samples

Returns a Samples object with three attributes: name, parameters, and samples.

Methods

reproject: reproject the samples

Reprojects the samples GeoDataFrame to a target CRS.

Parameters:

crs_target : str | int | pyproj.CRS, optional, default None

The CRS to reproject the samples to.

coords: extracts the coordinates from the centroids, midpoints, or points

Extracts the coordinates from the centroids, midpoints, or points and puts them in two new columns suffixed with '_lon' and '_lat' or '_x' and '_y'.

save: save the samples

Saves the samples GeoDataFrame as a GPKG, a CSV, or both. The name of the saved file(s) will be the name of the Samples object. Additionally, the parameters will be output as a CSV with the same name plus '-parameters'.

Parameters:

folder : str

The path to the output folder where the output files will be saved

filetype : {'gpkg', 'csv', 'both'}, optional, default 'gpkg'

The type of file that the sections will be saved as.

gpkg: GeoPackage

csv: CSV

both: GeoPackage and CSV

crs_output : str | int | pyproj.CRS, optional, default None

Optionally, the CRS to reproject the samples to before saving (only reprojects the samples that are saved and not the Samples object).

coords : bool, optional, default False

If True, x and y coordinates will be extracted from the centroid, midpoint, or point geometries and put in separate columns. This may facilitate subsequent extraction of data from external sources.