Grid cells package Documentation

Release 0.1

Lukas Solanka

June 08, 2015

Contents

CHAPTER 1

Overview

1.1 What are grid cells

Grid cells are a type of cells found in some mammalian species that have very regular spatial firing fields. When these animals forage in experimental arenas, grid cells are active only in certain places and the locations where they are active form a hexagonal lattice. More information can be found in *[MOSER2007]* or in *[HAFTING2005]*.

1.2 What does `gridcells` do

`gridcells` is a simple Python library that aims to provide an open source code repository related to analysis of grid cell related experiments and simulation of models of grid cells.

1.3 Download

`gridcells` can be downloaded from https://github.com/lsolanka/gridcells.

1.4 Dependencies

There are a number of dependencies needed for the python version:

- setuptools (>= 3.6)
- SWIG (ideally >= 3.0; earlier versions not tested)
- numpy (>= 1.8)
- scipy (>= 0.13.3)
- matplotlib (>= 1.3.1)

For Linux, simply install these using the package manager.For Mac OS the easiest way is probably to use homebrew. This package has not been tested on Windows but if you manage to install the dependencies there should be no problems.

1.5 Installation

After installing the dependencies, run:

```
python setup.py install
```

Note: Currently, installing with `pip` through the Python Package Index has some difficulties with the dependency on numpy. Thus it is advisable to have `numpy` installed beforehand.

1.6 License

gridcells is distributed under the GPL license. See LICENSE.txt in the root of the source directory.

1.7 References

CHAPTER 2

Modules

2.1 gridcells.core.common - common/shared definitions

The *common* module is a collection of basic classes used throughout the package:

Pair2D(x, y)	A pair of x and y attributes.
Position2D(x, y, dt)	Positional information with a constant time step.
twisted_torus_distance(a, others, dim)	Calculate a distance between a and others on a twisted torus.

class gridcells.core.common.**Pair2D**(*x*, *y*)
Bases: object

A pair of x and y attributes.

class gridcells.core.common.**Position2D**(*x*, *y*, *dt*)
Bases: *gridcells.core.common.Pair2D*

Positional information with a constant time step.

gridcells.core.common.**twisted_torus_distance**(*a*, *others*, *dim*)
Calculate a distance between a and others on a twisted torus.

Take a which is a 2D position and others, which is a vector of 2D positions and compute the distances between them based on the topology of the twisted torus.

If you just want to remap a function of (X, Y), set a==[[0, 0]].

Parameters **a** : *Pair2D*

Specifies the initial position. a.x and a.y must be convertible to floats

others : *Pair2D*

Positions for which to compute the distance.

dim : *Pair2D*

Dimensions of the torus. dim.x and dim.y must be convertible to floats.

Returns An array of positions, always of the length of others

2.2 gridcells.core.arena - Defining arenas

The *arena* module provides class definitions of arenas. These can subsequently be used as input to process spiking data and generate spatial firing fields/autocorrelations.

2.2.1 These types of arenas are currently defined:

Arena	An abstract class for arenas.
CircularArena(radius, discretisation)	A circular arena.
RectangularArena(size, discretisation)	A rectangular arena.
SquareArena(size, discretisation)	A square arena.

class gridcells.core.arena.**Arena**
Bases: object

An abstract class for arenas.

This class is an interface for obtaining discretisations of the arenas and masks when the shape is not rectangular.

getDiscretisation()
Obtain the discretisation of this arena.

Returns **d** : gridcells.core.Pair2D

A pair of x and y coordinates for the positions in the arena. Units are arbitrary.

getMask()
Return mask (a 2D np.ndarray) of where the positions in the arena are valid.

For isntance with a circular arena, all positions outside its radius are invalid.

class gridcells.core.arena.**CircularArena**(*radius*, *discretisation*)
Bases: *gridcells.core.arena.SquareArena*

A circular arena.

sz
Return the size of the arena. Equivalent to getSize().

class gridcells.core.arena.**RectangularArena**(*size*, *discretisation*)
Bases: *gridcells.core.arena.Arena*

A rectangular arena.

Use RectangularArena when you need to work with rectangular arenas.

Note: The origin (0, 0) of the coordinate system in all the arenas is in the bottom-left corner of the arena.

sz
Return the size of the arena. Equivalent to getSize().

class gridcells.core.arena.**SquareArena**(*size*, *discretisation*)
Bases: *gridcells.core.arena.RectangularArena*

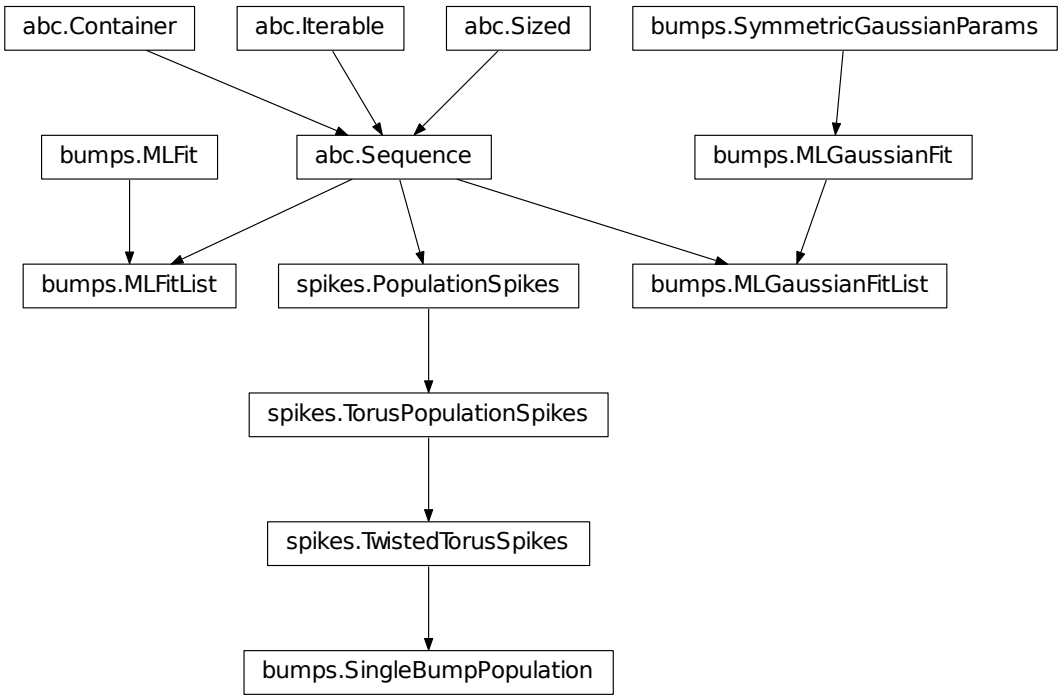
A square arena.

sz
Return the size of the arena. Equivalent to getSize().

2.3 gridcells.analysis.bumps - bump tracking

Classes and functions for processing data related to bump attractors.

2.3.1 Classes



MLFit(mu, sigma2, ln_lh, err2)	Maximum likelihood fit data holer.
MLFitList([mu, sigma2, ln_lh, err2, times])	A container for holding results of maximum likelihood fitting.
MLGaussianFit(amplitude, mu_x, mu_y, sigma, ...)	Gaussian fit performed by applying maximum likelihood estimator.
MLGaussianFitList([amplitude, mu_x, mu_y, ...])	A container for holding maximum likelihood Gaussian fits.
SingleBumpPopulation(senders, times, sheet_size)	A population of neurons that is supposed to form a bump on a twisted toru
SymmetricGaussianParams(amplitude, mu_x, ...)	Parameters for the symmetric Gaussian function.

2.3.2 Functions

fit_gaussian_tt(sig_f, i)	Fit a 2D circular Gaussian function to a 2D signal using a maximum likelihood estimator.
fit_gaussian_bump_tt(sig)	Fit a 2D Gaussian onto a (potential) firing rate bump on the twisted torus.
fit_maximum_lh(sig)	Fit a maximum likelihood solution under Gaussian noise.

class gridcells.analysis.bumps.**MLFit**(*mu*, *sigma2*, *ln_lh*, *err2*)

Bases: object

Maximum likelihood fit data holer.

class gridcells.analysis.bumps.**MLFitList**(*mu=None*, *sigma2=None*, *ln_lh=None*, *err2=None*, *times=None*)
Bases: *gridcells.analysis.bumps.MLFit*, collections.abc.Sequence

A container for holding results of maximum likelihood fitting.

Can be accessed as a Sequence object.

append_data(*d*, *t*)
d must be an instance of *MLFit*

class gridcells.analysis.bumps.**MLGaussianFit**(*amplitude*, *mu_x*, *mu_y*, *sigma*, *err2*, *ln_lh*, *lh_precision*)
Bases: *gridcells.analysis.bumps.SymmetricGaussianParams*

Gaussian fit performed by applying maximum likelihood estimator.

class gridcells.analysis.bumps.**MLGaussianFitList**(*amplitude=None*, *mu_x=None*, *mu_y=None*, *sigma=None*, *err2=None*, *ln_lh=None*, *lh_precision=None*, *times=None*)
Bases: *gridcells.analysis.bumps.MLGaussianFit*, collections.abc.Sequence

A container for holding maximum likelihood Gaussian fits.

Can be accessed as a Sequence.

append_data(*d*, *t*)
d must be an instance of *MLGaussianFit*

class gridcells.analysis.bumps.**SingleBumpPopulation**(*senders*, *times*, *sheet_size*)
Bases: *gridcells.analysis.spikes.TwistedTorusSpikes*

A population of neurons that is supposed to form a bump on a twisted torus.

Parameters **senders** : array_like

A an array of neurons' IDs.

times : array_like

An array of spike times. Length must be the same as as <senders>.

sheet_size : A pair

A pair of X and Y dimensions of the torus.

bump_position(*tstart*, *tend*, *dt*, *win_len*, *full_err=True*)
Estimate bump positions during the simulation time:

1. Estimates population firing rate for each bin.
2. Apply the bump position estimation procedure to each of the population activity items.

Parameters **tstart, tend, dt, win_len** : float

Start and end time, time step, and window length. See also *sliding_firing_rate()*.

full_err : bool

If `True`, save the full error of fit. Otherwise a sum only.

Returns pos:list *MLGaussianFitList*

A list of fitted Gaussian parameters

Notes

This method uses the Maximum likelihood estimator to fit the Gaussian function (*fit_gaussian_bump_tt()*)

uniform_fit(*tstart*, *tend*, *dt*, *win_len*, *full_err=True*)

Estimate the mean firing rate using maximum likelihood estimator (`fit_maximum_lh()`)

1. Uses `sliding_firing_rate()`.
2. Apply the estimator.

Parameters **tstart, tend, dt, win_len**

As in `sliding_firing_rate()`.

full_err : bool

If `True`, save the full error of fit. Otherwise a sum only.

Returns MLFitList

A list of fitted parameters.

class gridcells.analysis.bumps.**SymmetricGaussianParams**(*amplitude*, *mu_x*, *mu_y*, *sigma*, *err2*)

Bases: `object`

Parameters for the symmetric Gaussian function.

gridcells.analysis.bumps.**fit_gaussian_bump_tt**(*sig*)

Fit a 2D Gaussian onto a (potential) firing rate bump on the twisted torus.

Parameters **sig** : np.ndarray

2D firing rate map to fit. Axis 0 is the Y position. This will be passed directly to `fit_gaussian_tt()`.

Returns `analysis.image.MLGaussianFit`

Estimated values of the fit.

Notes

The function initialises the Gaussian fitting parameters to a position at the maximum of *sig*.

gridcells.analysis.bumps.**fit_gaussian_tt**(*sig_f*, *i*)

Fit a 2D circular Gaussian function to a 2D signal using a maximum likelihood estimator.

The Gaussian is not generic: $\sigma_x = \sigma_y = \sigma$, i.e. it is circular only.

The function fitted looks like this:

$$f(\mathbf{X}) = |A| \exp\left\{\frac{-|\mathbf{X} - \mu|^2}{2\sigma^2}\right\}$$

where $|\cdot|$ is a distance metric on the twisted torus.

Parameters **sig_f** : np.ndarray

A 2D array that specified the signal to fit the Gaussian onto. The dimensions of the torus will be inferred from the shape of *sig_f*: (dim.y, dim.x) = *sig_f.shape*.

i : SymmetricGaussianParams

Guassian initialisation parameters. The *err2* field will be ignored.

Returns *MLGaussianFit*

Estimated values, together with maximum likelihood value and precision (inverse variance of noise: *NOT* of the fitted Gaussian).

gridcells.analysis.bumps.**fit_maximum_lh**(*sig*)

Fit a maximum likelihood solution under Gaussian noise.

Parameters **sig** : np.ndarray

A vector containing the samples

Returns

fit : MLFit

Maximum likelihood parameters

2.4 gridcells.analysis.fields - grid field related analysis

The *fields* module contains routines to analyse spiking data either from experiments involoving a rodent running in an arena or simulations involving an animat running in a simulated arena.

2.4.1 Functions

gridnessScore(rateMap, arenaDiam, h, ...)	Calculate gridness score of a spatial firing rate map.
occupancy_prob_dist(arena, pos)	Calculate a probability distribution for animal positions in an arena.
spatialAutoCorrelation(rateMap, arenaDiam, h)	Compute autocorrelation function of the spatial firing rate map.
spatialRateMap(spikeTimes, positions, arena, ...)	Compute spatial rate map for spikes of a given neuron.

gridcells.analysis.fields.**gridnessScore**(*rateMap*, *arenaDiam*, *h*, *corr_cutRmin*)

Calculate gridness score of a spatial firing rate map.

Parameters **rateMap** : np.ndarray

Spatial firing rate map.

arenaDiam : float

The diameter of the arena.

h : float

Precision of the spatial firing rate map.

Returns **G** : float

Gridness score.

crossCorr : np.ndarray

An array containing cross correlation values of the rotated autocorrelations, with the original autocorrelation.

angles : np.ndarray

An array of angles corresponding to the *crossCorr* array.

Notes

This function computes gridness score accoring to *[R1]*. The auto correlation of the firing rate map is rotated in 3 degree steps. The resulting gridness score is the difference between a minimum of cross correlations at 60 and 90 degrees, and a maximum of cross correlations at 30, 90 and 150 degrees.

The center of the auto correlation map (given by corr_cutRmin) is removed from the map.

References

[R1]

gridcells.analysis.fields.**occupancy_prob_dist**(*arena*, *pos*)

Calculate a probability distribution for animal positions in an arena.

Parameters **arena** : *Arena*

Arena the animal was running in.

pos : *Position2D*

Positions of the animal.

Returns **dist** : numpy.ndarray

Probability distribution for the positional data, given the discretisation of the arena. The first dimension is the y axis, the second dimension is the x axis. The shape of the distribution is equal to the number of items in the discretised edges of the arena.

gridcells.analysis.fields.**spatialAutoCorrelation**(*rateMap*, *arenaDiam*, *h*)

Compute autocorrelation function of the spatial firing rate map.

This function assumes that the arena is a circle and masks all values of the autocorrelation that are outside the *arenaDiam*.

Warning: This function will undergo serious interface changes in the future.

Parameters **rateMap** : np.ndarray

Spatial firing rate map (2D). The shape should be *(arenadiam/h+1, arenadiam/2+1)*.

arenaDiam : float

Diameter of the arena.

h : float

Precision of the spatial firing rate map.

Returns **corr** : np.ndarray

The autocorrelation function, of shape *(arenadiam/h*2+1, arenaDiam/h*2+1)*

xedges, yedges : np.ndarray

Values of the spatial lags for the correlation function. The same shape as *corr.shape[0]*.

gridcells.analysis.fields.**spatialRateMap**(*spikeTimes*, *positions*, *arena*, *sigma*)

Compute spatial rate map for spikes of a given neuron.

Preprocess neuron spike times into a smoothed spatial rate map, given arena parameters. Both spike times and positional data must be aligned in time! The rate map will be smoothed by a gaussian kernel.

Parameters **spikeTimes** : np.ndarray

Spike times for a given neuron.

positions : gridcells.core.Position2D

Positional data for these spikes. The timing must be aligned with `spikeTimes`

arena : gridcells.core.Arena

The specification of the arena in which movement was carried out.

sigma : float

Standard deviation of the Gaussian smoothing kernel.

Returns **rateMap** : np.ma.MaskedArray

The 2D spatial firing rate map. The shape will be determined by the arena type.

2.5 `gridcells.analysis.info` - Information-theoretical analysis

The *info* module contains routines related to information-theoretic analysis of data related to grid cells.

2.5.1 Functions

information_rate(rate_map, px)	Compute information rate of a cell given variable x.
information_specificity(rate_map, px)	Compute the 'specificity' of the cell firing rate to a variable X.

gridcells.analysis.info.**information_rate**(*rate_map*, *px*)

Compute information rate of a cell given variable x.

A simple algorithm devised by *[R3]*. This computes the spatial information rate of cell spikes given variable x (e.g. position, head direction) in bits/second.

Parameters **rate_map** : numpy.ndarray

A firing rate map, any number of dimensions. If units are in Hz, then the information rate will be in bits/s.

px : numpy.ndarray

Probability density function for variable `x`. `px.shape` must be equal `rate_maps.shape`

Returns **I** : float

Information rate.

Notes

If you need information in bits/spike, you need to divide the information rate by the average firing rate of the cell.

The firing rate map, in positions that are valid within the arena, may contains NaN numbers. In that case, the firing rate in these positions in `rate_map` will be set to 0.

References

[R3]

gridcells.analysis.info.**information_specificity**(*rate_map*, *px*)
Compute the 'specificity' of the cell firing rate to a variable X.

Compute *information_rate()* for this cell and divide by the average firing rate of the cell. See *[R4]* for more information.

Parameters **rate_map** : numpy.ndarray

A firing rate map, any number of dimensions.

px : numpy.ndarray

Probability density function for variable `x`. `px.shape` must be equal `rate_maps.shape`

Returns **I** : float

Information in bits/spike.

References

[R4]

2.6 gridcells.analysis.registration - Positional data registration.

Use the classes here to align (register) positional data of several recordings with the specified arena coordinates.

2.6.1 Classes

ArenaOriginRegistration([arena])	Register positional data to zero-coordinates of an arena.
OriginRegistrationResult(positions, offsets)	A holder for registered data.

class gridcells.analysis.registration.**ArenaOriginRegistration**(*arena=None*)
Bases: `object`

Register positional data to zero-coordinates of an arena.

The actual positional data recordings are prone to outliers. This registration engine ensures that the positional data from different recordings are "aligned" with respect to the arena coordinates. This is accomplished by optimising the positional offsets with respect to the number of outliers.

Todo

Deal with rotations.

Initialise with an `arena` against which to register the data.

Also use *set_arena()* to change the specific arena.

__init__(*arena=None*)
Initialise with an `arena` against which to register the data.

Also use *set_arena()* to change the specific arena.

register(*positions*)
Register the positional data against the current arena.

Parameters **positions** : *Position2D*

Positional data.

Returns **res** : *OriginRegistrationResult*

The result object, containing new positional data and the determined offsets.

set_arena(*arena*)
Set the arena for registration.

All subsequent calls to *register()* will be performed on this arena.

class gridcells.analysis.registration.**OriginRegistrationResult**(*positions*, *offsets*)
Bases: `object`

A holder for registered data.

Contains two attributes: `positions` and estimated `offsets` in the arena.

2.7 gridcells.analysis.signal - signal analysis

The can be e.g. filtering, slicing, correlation analysis, up/down-sampling, etc.

acorr(sig[, max_lag, norm, mode])	Compute an autocorrelation function of a real signal.
corr(a, b[, mode, lag_start, lag_end])	An enhanced correlation function of two real signals, based on a custom C++ code.
ExtremumTypes	Specifies types of extrema.
local_extrema(sig)	Find all local extrema using the derivative approach.
local_maxima(sig)	Find all local maxima using the derivative approach
local_minima(sig)	Find all local minima using the derivative approach
LocalExtrema(extrema, types)	A class representing local extrema for a particular object.

gridcells.analysis.signal.**acorr**(*sig*, *max_lag=None*, *norm=False*, *mode='onesided'*)
Compute an autocorrelation function of a real signal.

Parameters **sig** : numpy.ndarray

The signal, 1D vector, to compute an autocorrelation of.

max_lag : int, optional

Maximal number of lags. If mode == 'onesided', the range of lags will be [0, max_lag], i.e. the size of the output will be (max_lag+1). If mode == 'twosided', the lags will be in the range [-max_lag, max_lag], and so the size of the output will be 2*max_lag + 1.

If max_lag is None, then max_lag will be set to len(sig)-1

norm : bool, optional

Whether to normalize the auto correlation result, so that res(0) = 1

mode : string, optional

`onesided` or `twosided`. See description of max_lag

output : numpy.ndarray

A 1D array, size depends on `max_lag` and `mode` parameters.

Notes

If the normalisation constant is zero (i.e. the input array is zero), this function will return a zero array.

`gridcells.analysis.signal.`**`corr`**(*a*, *b*, *mode='onesided'*, *lag_start=None*, *lag_end=None*)

An enhanced correlation function of two real signals, based on a custom C++ code.

This function uses dot product instead of FFT to compute a correlation function with range restricted lags.

Thus, for a long-range of lags and big arrays it can be slower than the numpy.correlate (which uses fft-based convolution). However, for arrays in which the number of lags << max(a.size, b.size) the computation time might be much shorter than using convolution to calculate the full correlation function and taking a slice of it.

Parameters:

a, b [ndarray] One dimensional numpy arrays (in the current implementation, they will be converted to dtype=double if not already of that type.

mode [str, optional] A string indicating the size of the output:

`onesided` : range of lags is [0, b.size - 1]

`twosided` : range of lags is [-(a.size - 1), b.size - 1]

`range` : range of lags is [-lag_start, lag_end]

lag_start, lag_end [int, optional] Initial and final lag value. Only used when mode == 'range'

output [numpy.ndarray with shape (1,) and dtype.float] A 1D array of size depending on mode

---

Note: This function always returns a numpy array with dtype=float.

See also:

acorr()

`gridcells.analysis.signal.`**`local_extrema`**(*sig*)

Find all local extrema using the derivative approach.

Parameters **sig** : numpy.ndarray

A 1D numpy array

Returns **extrema** : *LocalExtrema*

An object containing the local extrema of the signal `sig`.

See also:

local_minima Finds local minima.

local_maxima Finds local maxima.

Notes

This method is not suitable to find local extrema of functions where the extremum is flat, i.e. as in square pulses.

gridcells.analysis.signal.**local_minima**(*sig*)
Find all local minima using the derivative approach

Parameters **sig** : numpy.ndarray

A 1D numpy array

Returns **maxima** : np.ndarray

An array of indices into `sig` of the local minima.

See also:

local_extrema Finds local extrema.

local_maxima Finds local maxima.

gridcells.analysis.signal.**local_maxima**(*sig*)
Find all local maxima using the derivative approach

Parameters **sig** : numpy.ndarray

A 1D numpy array

Returns **maxima** : np.ndarray

An array of indices into `sig` of the local maxima.

See also:

local_extrema Finds local extrema.

local_minima Finds local minima.

class gridcells.analysis.signal.**ExtremumTypes**
Bases: `enum.IntEnum`

Specifies types of extrema.

MAX = None
Local maximum.

MIN = None
Local minimum.

UNDEFINED = None
Undefined.

class gridcells.analysis.signal.**LocalExtrema**(*extrema*, *types*)
Bases: `object`

A class representing local extrema for a particular object.

This is only a helper class. Users should not instantiate this class directly

Note: For now only 1D extrema are supported.

Parameters **extrema** : 1D numpy array

Positions of the extrema in a signal. The user must track the source.

types : 1D numpy array

Types of the extrema held in this object. Contains values from *ExtremumTypes*.

get_type(*extremum_type*)

Get all the local extrema that are of type `extremum_type`.

Parameters **extremum_type** : *ExtremumTypes*

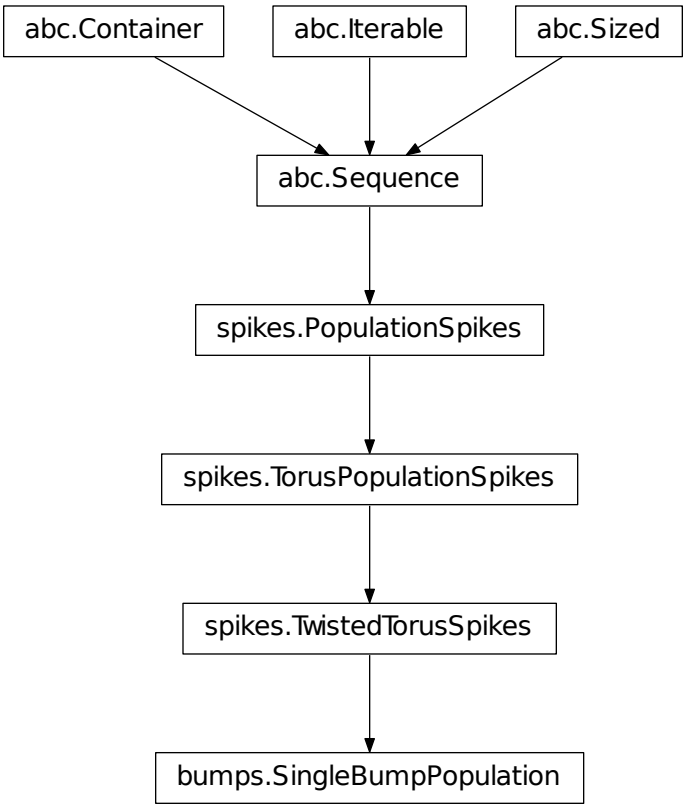
The type of the extremum to retrieve.

Returns **extrema** : iterable

An iterable that will contain the extrema with the specified type. If no extremum of the requested type is present, returns an empty array.

2.8 gridcells.analysis.spikes - spike train analysis

2.8.1 Classes



PopulationSpikes(n, senders, times)	Abstraction of a population of spikes.
TorusPopulationSpikes(senders, times, sheet_size)	Spikes of a population of neurons on a twisted torus.
TwistedTorusSpikes(senders, times, sheet_size)	Spikes arranged on twisted torus.

class gridcells.analysis.spikes.**PopulationSpikes**(*n*, *senders*, *times*)

Bases: collections.abc.Sequence

Abstraction of a population of spikes.

Parameters **n** : int

Number of neurons in the population

senders : 1D array

Neuron numbers corresponding to the spikes

times : 1D array

Spike times. The shape of this array must be the same as for *senders*.

class **CallableHistogram**(***kw*)

Bases: object

Callable class that computes a histogram.

get_bin_edges()

Calculate bin edges.

PopulationSpikes.**avg_firing_rate**(*tstart*, *tend*)

Compute and average firing rate for all the neurons between 'tstart' and 'tend'. Return an array of firing rates, one item for each neuron in the population.

Parameters **tstart** : float (ms)

Start time.

tend : float (ms)

End time.

Returns **output** : numpy array

Firing rate in Hz for each neuron in the population.

PopulationSpikes.**isi**(*n=None*, *reduce_fun=None*)

Return interspike interval of one or more neurons.

Parameters **n** : None, int, or sequence

Neuron numbers. If n is None, then compute ISI stats for all neurons in the population. If n is an int, compute ISIs for just neuron indexed by n. Otherwise n is expected to be a sequence of neuron indices.

reduce_fun : callable or None

A reduction function (callable object) that performs an operation on all the ISIs of the population. If None, nothing is done. The callable has to take one input parameter, which is the sequence of ISIs. This allows to cascade data processing without the need for duplicating spike timing data.

Returns output: list

A list of outputs (depending on parameters) for each neuron, even if n is an int.

PopulationSpikes.**isi_cv**(*n=None*, *win_len=None*)
Coefficients of variation of inter-spike intervals of one or more neurons in the population. For the description of parameters and outputs and their semantics see also ISI().

Parameters **win_len** : float, list of floats, or None

Specify the maximal ISI value, i.e. use windowed coefficient of variation. If None, use the whole range.

PopulationSpikes.**isi_neuron**(*n*)
Compute all interspike intervals of one neuron with ID n. If the number of spikes is less than 2, returns an empty array.

Todo

Works on sorted spike trains only!

Note: If you get negative interspike intervals, you will need to sort your spike times (per each neuron).

PopulationSpikes.**n**
Number of neurons in the population

PopulationSpikes.**raster_data**(*neuron_list=None*)
Extract the senders and corresponding spike times for a raster plot.

Todo

implement neuron_list

Parameters **neuron_list** : list, optional

Extract only neurons given in this list

Returns **output** : a tuple

A pair containing (senders, times).

PopulationSpikes.**sliding_firing_rate**(*tstart*, *tend*, *dt*, *win_len*)
Compute a sliding firing rate over the population of spikes, by taking a rectangular window of specified length.

Parameters **tstart** : float

Start time of the firing rate analysis.

tend : float

End time of the analysis

dt : float

Firing rate window time step

win_len : float

Lengths of the windowing function (rectangle)

Returns **output** : a tuple

A pair (F, t), specifying the vector of firing rates and corresponding times. F is a 2D array of the shape (n, Ntimes), in which n is the number of neurons and Ntimes is the number of time steps. 't' is a vector of times corresponding to the time windows taken.

PopulationSpikes.**spike_train_difference**(*idx1*, *idx2=None*, *full=True*, *reduce_fun=None*)

Compute time differences between pairs of spikes of two neurons or a list of neurons.

Parameters **idx1** : int, or a sequence of ints

Index of the first neuron or a list of neurons for which to compute the correlation histogram.

idx2 : int, or a sequence of ints, or None

Index of the second neuron or a list of indexes for the second set of spike trains.

full : bool, optional

Not fully implemented yet. Must be set to True.

reduce_fun : callable, optional

Any callable object that computes a function over an array of each spike train difference. The function must take one input argument, which will be the array of spike time differences for a pair of neurons. The output of this function will be stored instead of the default output.

Returns **output** : A 2D or 1D array

Spike train autocorrelation histograms for all the pairs of neurons.

The computation takes the following steps:

- If `idx1` or `idx2` are integers, they will be converted to a list of size 1.
- If `idx2` is None, then the result will be a list of lists of pairs of cross-correlations between the neurons. Even if there is only one neuron. If `full == True`, the output will be an upper triangular matrix of all the pairs, i.e. it will exclude the duplicated. Otherwise there will be cross correlation histograms between all the pairs.
- if `idx2` is not None, then `idx1` and `idx2` must be arrays of the same length, specifying the pairs to compute autocorrelation for

PopulationSpikes.**spike_train_xcorr**(*idx1*, *idx2*, *lag_range*, *bins=50*, ***kw*)

Compute the spike train crosscorrelation function for all pairs of spike trains in the population.

For explanation of how `idx1` and `idx2` are treated, see *spike_train_difference()*.

Parameters **idx1** : int, or a sequence of ints

Index of the first neuron or a list of neurons for which to compute the correlation histogram.

idx2 : int, or a sequence of ints, or None

Index of the second neuron or a list of indexes for the second set of spike trains.

lag_range : (lag_start, lag_end)

Limits of the cross-correlation function. The bins will always be **centered** on the values.

bins : int, optional

Number of bins

kw : dict

Keyword arguments passed on to the numpy.histogram function

Returns **output** : a 2D or 1D list

See *spike_train_difference()*.

PopulationSpikes.**windowed**(*tlimits*)
Return population spikes restricted to tlimits.

Parameters **tlimits** : a pair

A tuple (tstart, tend). The spikes in the population must satisfy tstart >= t <= tend.

Returns **output** : PopulationSpikes instance

A copy of self with only a subset of spikes, limited by the time window.

class gridcells.analysis.spikes.**TorusPopulationSpikes**(*senders*, *times*, *sheet_size*)
Bases: *gridcells.analysis.spikes.PopulationSpikes*

Spikes of a population of neurons on a twisted torus.

dimensions
Dimensions of the torus (X, Y)

get_dimensions()
Size of the torus.

get_x_size()
Horizontal size of the torus.

get_y_size()
Vertical size of the torus.

nx
Horizontal size of the torus

ny
Vertical size of the torus

population_vector(*tstart*, *tend*, *dt*, *win_len*)
Compute the population vector on a torus, from the spikes present. Note that this method will have a limited functionality on a twisted torus, but can be used if the population activity translates in the X dimension only.

Parameters **tstart** : float

Start time of analysis

tend : float

End time of analysis

dt : float

Time step of the (rectangular) windowing function

win_len : float

Length of the windowing function

Returns **output** : tuple

A pair (r, t) in which r is a 2D vector of shape (int((tend-tstart)/dt)+1), 2), corresponding to the population vector for each time step of the windowing function, and t is a vector of times, of length the first dimension of r.

sliding_firing_rate(*tstart*, *tend*, *dt*, *win_len*)
Compute a sliding firing rate over the population of spikes, by taking a rectangular window of specified

length. However, unlike the ancestor method (PopulationSpikes.sliding_firing_rate), return a 3D array, a succession of 2D population firing rates in time.

Parameters **tstart** : float

Start time of the firing rate analysis.

tend : float

End time of the analysis

dt : float

Firing rate window time step

win_len : float

Lengths of the windowing function (rectangle)

Returns **output** : a tuple

A pair (F, t), specifying the vector of firing rates and corresponding times. F is a 3D array of the shape (nx, ny, Ntimes), in which nx/ny are the number of neurons in X and Y dimensions, respectively, and Ntimes is the number of time steps. 't' is a vector of times corresponding to the time windows taken.

class `gridcells.analysis.spikes.`**`TwistedTorusSpikes`**(*senders*, *times*, *sheet_size*)

Bases: *gridcells.analysis.spikes.TorusPopulationSpikes*

Spikes arranged on twisted torus. The torus is twisted in the X direction.

2.9 `gridcells.plotting.fields` - grid field related plotting

The *fields* module contains routines to create matplotlib plots of spatial firing fields and similar commonly used structures.

2.9.1 How to plot

The plotting is currently realized as a subclass of `matplotlib.axes.Axes` and is used via a `projection="gridcells_arena"` keyword argument. Since a custom `Axes` class is not part of standard matplotlib, before using the `projection`, you have to first register the plotting class with matplotlib by importing the plotting module:

```
import matplotlib.pyplot as plt
from gridcells.plotting import fields
from gridcells.core import SquareArena, Pair2D
```

Next, create an Axes object that you can plot to:

```
fig = plt.figure()
arena = SquareArena(100., Pair2D(1., 1.))
ax = fig.add_subplot(111, projection="gridcells_arena", arena=arena)
```

The `add_subplot` method takes a keyword argument `projection="gridcells_arena"` that specifies the type of Axes to use. Here, we also have to specify the `arena` parameter in the form of an *Arena* instance. In our case we have created a square arena with size 100x100 and a discretisation of 1x1 (in arbitrary units).

Next, for illustration purposes, we create a random spatial rate map with size compatible with the current arena, and plot to the axes, by calling *spatial_rate_map()*:

```
sz = arena.getDiscretisation()
rate_map = np.random.rand(len(sz.x), len(sz.y))
ax.spatial_rate_map(rate_map)
```

2.9.2 Custom grid cell plotting Axes

GridArenaAxes(*args, **kwargs)	A custom matplotlib Axes that allows to plot figures in the shape of arenas.

class gridcells.plotting.fields.**GridArenaAxes**(**args*, ***kwargs*)

Bases: Axes

A custom matplotlib Axes that allows to plot figures in the shape of arenas.

Methods:

fft2(rate_map[, scalebar, scaletext, fftn, ...])	Plot a 2D Fourier transform (power) of a spatial rate map.
spatial_rate_map(rate_map[, scalebar, ...])	Plot the spatial rate map in the specified arena.
spikes(spike_times, pos[, dotsize, ...])	Plot spike positions.

Create the axes.

Should not be used directly, but via the matplotlib `projection` keyword argument.

Parameters **arena** : *Arena*

Are that will be used for plotting. This has to be specified as a keyword argument to matplotlib's `Figure.add_subplot()` or `Figure.add_axes()`.

default_margin = 0.0

Default margin in axis coordinates

fft2(*rate_map*, *scalebar=None*, *scaletext='cm^{-1}'*, *fftn=None*, *subtractmean=True*, ***kwargs*)

Plot a 2D Fourier transform (power) of a spatial rate map.

Parameters **rate_map** : np.ndarray

The rate map as 2D array. Rows determine the Y coordinate, columns the X coordinate. Masked items will be ignored.

scalebar : float, optional

The length of the scale bar that will be plotted as horizontal line. Must be in data units.

scaletext : str, optional

Text after the scale bar number, i.e. units.

fftn : int, optional

Size of the array that the Fourier transform is actually computed from. If `None` it will be `max(rate_map.shape)`. Otherwise the `rate_map` will be padded with zeros.

subtractmean : bool, optional

Whether to subtract the mean of the signal before computing the FFT. This will remove any constant component in the centre of the spectrogram.

kwargs : kwargs

Optional kwargs that will be passed to matplotlib's pcolormesh.

name = 'gridcells_arena'
Name of the class used to register with matplotlib

spatial_rate_map(*rate_map*, *scalebar=None*, *scaletext='cm'*, *maxrate=True*, *g_score=None*, *maxrate_fs='xx-small'*, ***kwargs*)
Plot the spatial rate map in the specified arena.

Parameters **rate_map** : np.ndarray

The rate map as 2D array. Rows determine the Y coordinate, columns the X coordinate. Masked items will be ignored.

scalebar : float, optional

The length of the scale bar that will be plotted as horizontal line. Must be in data units.

scaletext : str, optional

Text after the scale bar number, i.e. units.

maxrate : bool, optional

Whether to print the max firing rate (top right corner)

g_score : float, optional

Grid score for this spatial rate map. If `None`, plot nothing.

maxrate_fs : matplotlib font size identifier

Font size for maxrate.

kwargs : kwargs

Optional kwargs that will be passed to matplotlib's pcolormesh.

spikes(*spike_times*, *pos*, *dotsize=5*, *scalebar=None*, *scaletext='cm'*, ***kwargs*)
Plot spike positions.

Both positions and spikes must be aligned!

Parameters **spike_times** : np.ndarray

Spike times to plot on top of the trajectories

pos : gridcells.Position2D

Positional data for the spike times.

dotsize : float

Size of spike dots.

scalebar : float, optional

The length of the scale bar that will be plotted as horizontal line. Must be in data units.

scaletext : str, optional

Text after the scale bar number, i.e. units.

kwargs : kwargs

Optional kwargs that will be passed to matplotlib's plot functions (for the trajectories and spike dots).

[MOSER2007] Edvard Moser and May-Britt Moser (2007). Grid cells. Scholarpedia, 2(7):3394.

[HAFTING2005] Hafting, T. et al., 2005. Microstructure of a spatial map in the entorhinal cortex. Nature, 436(7052), pp.801–806.

[R1] Hafting, T. et al., 2005. Microstructure of a spatial map in the entorhinal cortex. Nature, 436(7052), pp.801-806.

[R3] Skaggs, W.E. et al., 1993. An Information-Theoretic Approach to Deciphering the Hippocampal Code. In Advances in Neural Information Processing Systems 5. pp. 1030-1037.

[R4] Skaggs, W.E. et al., 1993. An Information-Theoretic Approach to Deciphering the Hippocampal Code. In Advances in Neural Information Processing Systems 5. pp. 1030-1037.

g

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

W