Attractors and Oscillation

Benedikt Rank

Mar 31, 2022

CONTENTS:

repository: https://github.com/bnra/attractors_and_oscillation

Emergence of Attractor Dynamics in Stochastically Synchronized Networks and Interplay between Attractor Dynamics and Oscillatory Dynamics

- Hopfield-like Conductance Scaling in E-I Networks with EIF neurons
- Stochastic Synchronization
- Gamma-breadth snapshots of network activity as network state

Ever since Hopfield's seminal paper on auto-associative memory models in 1982, Hopfield Net- works have garnered significant interest, not least because of their powerful functional properties and their ability to explain high-level phenomena in cognition, such as pattern completion. More recently more biologically plausible neuron and network models have been applied successfully to implement comparable network state evolutions by making use of network dynamics arising from complex spiking neuron models and E-I networks. However, these mechanisms rely on highly synchronous oscillation generated by an E-I network which is linked to pathological conditions and requires cell rates for the excitatory population that are far in excess of empirical findings. In stochastic oscillation on the other hand high, frequency oscillations arise as an emergent property while individual cells spike irregularly at low, biologically plausible rates. In this project the emer- gence of attractor dynamics in a stochastically synchronized network and the interaction between the attractor dynamics and oscillation are explored.

CHAPTER

ONE

API REFERENCE

This page contains auto-generated API reference documentation[1].

1.1 attractor

1.1.1 Module Contents

Functions

resolve_time_interval(stimulus_onset, inter_presentation_interval, stimulus_length)	compute time intervals relative to the onset of stimulus presentations [stimulus_onset - inter_presentation_interval + stimulus_length, stimulus_onset + inter_presentation_interval],
resolve_spike_times(stimulus_onset, inter_presentation_interval, stimulus_length)	resolve spike times relative to the onset of stimulus presentations [stimulus_onset - inter_presentation_interval + stimulus_length, stimulus_onset + inter_presentation_interval],
resolve_snapshots(troughs, inter_presentation_interval, first_stimulus_onset)	resolve snapshots relative to the onset of stimulus presentations,
separate_presentation_cycles(troughs, peaks, stimulus_onset)	separate presentation cycles according to the distance of the stimulus onset to the trough or peak, respectively
separate_snapshots(troughs, stimulus_end, stimulus_length, inter_presentation_interval)	separate snapshots according to the distance of the time point of ceasure of the stimulus presentation to the time point at which the next snapshot starts
fraction_significant_snapshots(pvalue, stimulus_pattern, pattern, significance = 0.05)	compute fraction of significant snapshots
indices_snapshots_blocked_stimulus(pvalue, stimulus_pattern, pattern, troughs, peaks, stimulus_onset, stimulus_length)	compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for windows
fraction_significant_snapshots_blocked_stimulus(pvalue, stimulus_pattern, pattern, troughs, peaks, stimulus_onset, stimulus_length, significance = 0.05)	compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for windows
fraction_significant_snapshots_blocked_stimulus_alternative(pvalue, stimulus_pattern, pattern, troughs, peaks, stimulus_onset, stimulus_length, significance = 0.05)	compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for windows

continues on next page

[1] Created with sphinx-autoapi

Table 1 – continued from previous page

indices_snapshots_blocked_stimulus_sliding_window(pvalue, stimulus_pattern, pattern, troughs, peaks, stimulus_onset, t_end, inter_onset_interval, window_length, window_step)	compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for a sliding window
fraction_significant_snapshots_blocked_stimulus_sliding_window(pvalue, stimulus_pattern, pattern, troughs, peaks, stimulus_onset, t_end, inter_onset_interval, window_length, window_step, significance = 0.05)	compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for a sliding window
fraction_significant_snapshots_across_intervals(pvalue, stimulus_pattern, pattern, troughs, peaks, interval, significance = 0.05)	compute fraction of significant snapshots across a set of intervals
indices_snapshots_across_intervals(pvalue, stimulus_pattern, pattern, troughs, peaks, interval)	extract idx of significant snapshots across a set of intervals
accuracy(snapshot, pattern)	accuracy between snapshot and pattern
log_fac(n)	logarithm of factorial approximated by Stirling's approximation
log_choose(n, k)	log(n choose k) := log(n! / (k! * (n-k)!)) = log(n!) - log(k!) -log((n-k)!)
choose(n, k)	n choose k := n! / (k! * (n-k)!)
p_value_snapshot_same_sparsity(similarity, sparsity, pattern)	p_value of drawing a pattern X = {x_i}, i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l achieving a higher or equal similarity (by chance ~ H0) to parameter similarity
pvalue_snapshot_sparsity_missmatch(similarity, sparsity, pattern, spike_count, num_cycles)	pvalue snapshot sparsity missmatch - probability of a flip ~ probability of spiking in the snapshot in a given cycle: ps
pvalue_snapshot_sparsity_missmatch_single_cycle(similarity, sparsity, pattern, spike_count)	pvalue snapshot sparsity missmatch with p snapshot-specific - probability of a flip ~ probability of spiking in the snapshot in a given cycle: ps
pvalue_snapshot(l, k, s, p)	pvalue of snapshot given similarity s with reference pattern ~ probability of observing a more or equally extreme (high) similarity assuming data is distriubted randomly (~ H0)
similarity(snapshot, pattern)	similarity between snapshot and pattern (dot_product)
p_value_snapshot_med(similarity, sparsity, pattern, num_spikes_snapshot, num_cycles)	**param similarity** similarity of a snapshot and pattern pair
p_value_snapshot_dot_product(similarity, sparsity, pattern, spike_count, num_cycles)	**param similarity** similarity of a snapshot and pattern pair (dot_product)
similarity_threshold(sparsity, pattern, spike_count, num_cycles, significance = 0.05, similarity = None)	critical threshold, a similarity value in the given range param similarity, whose p value is the tightest lower bound on the given significance level
similarity_conductance_scaling(pattern)	similarity rule for conductance scaling ~ pair-wise similarity btw two neurons (ie. per synapse) averaged across patterns
sim_vec(matrix)	similarity of row vectors
compute_conductance_scaling(patterns, sparsity)	compute the scaling factor of the conductance according to Battaglia, Treves 1998

continues on next page

Table 1 – continued from previous page

compute_conductance_scaling_single_clip(patterns, sparsity)	compute the scaling factor of the conductance by summing over patterns and clipping the result
compute_conductance_scaling_unclipped(patterns, sparsity)	compute the scaling factor of the conductance by summing over patterns
normalize(matrix, frm = None, to = None)	normalize (here squash) all values in matrix to [0,1] and if specified rescale to [frm,to]
z_score(matrix)	compute z score: z := (x - mu) / sigma
extract_snapshot_masks(pop_rate, t_start, t_end, dt)	extract snapshot masks of oscillation cycles,
extract_snapshots(spike_train, pop_size, pop_rate, t_start, t_end, dt)	**spike_train** spike_train per neuron [ms]

attractor.**resolve_time_interval**(*stimulus_onset*, *inter_presentation_interval*, *stimulus_length*)

compute time intervals relative to the onset of stimulus presentations [stimulus_onset - inter_presentation_interval + stimulus_length, stimulus_onset + inter_presentation_interval], given stimulus_onset, inter_presentation_interval and stimulus_length

Parameters

- **stimulus_onset** (*numpy.ndarray*) – set of onsets of stimulus presentations [ms]
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets or ceasures equivalently [ms]
- **stimulus_length** (*float*) – length of stimulus [ms]

Returns time intervals relative to the onset of stimulus presentations [stimulus_onset - inter_presentation_interval + stimulus_length, stimulus_onset + inter_presentation_interval]

attractor.**resolve_spike_times**(*stimulus_onset*, *inter_presentation_interval*, *stimulus_length*)

resolve spike times relative to the onset of stimulus presentations [stimulus_onset - inter_presentation_interval + stimulus_length, stimulus_onset + inter_presentation_interval], given stimulus_onset, inter_presentation_interval and stimulus_length

Parameters

- **stimulus_onset** (*numpy.ndarray*) – set of onsets of stimulus presentations [ms]
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets or ceasures equivalently [ms]
- **stimulus_length** (*float*) – length of stimulus [ms]

Returns spike times relative to stimulus onset

attractor.**resolve_snapshots**(*troughs*, *inter_presentation_interval*, *first_stimulus_onset*)

resolve snapshots relative to the onset of stimulus presentations, a snapshot is boolean spike mask over all cells indicating which cells spike over the interval between two troughs - the mid point between two troughs will be used as standin for the spike train of all spiking cells during the snapshot

Parameters

- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets or ceasures equivalently [ms]
- **first_stimulus_onset** (*float*) –

Returns resolved spike times relative to stimulus onset

attractor.**separate_presentation_cycles**(*troughs*, *peaks*, *stimulus_onset*)

separate presentation cycles according to the distance of the stimulus onset to the trough or peak, respectively ideally trough group: [t_trough-w/4, t_trough + w/4) or ii) peak group:[t_peak-w/4, t_peak + w/4). effectively: assign presentation cycle to group trough, peak for distance from stimulus onset to trough < peak, peak < trough, respectively where w refers to the wavelength, t_trough is the time point of any trough and therefore marks the beginning and ending of a snapshot, respectively.

Parameters

- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] (C+1,) where C is # snapshots
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms]
- **stimulus_onset** (*numpy.ndarray*) – time points of stimulus onset [ms](S,) where S is the number of stimulus presentations
- **stimulus_length** – length of stimulus_presentation [ms]
- **inter_presentation_interval** – interval between two stimulus presentation onsets or ceasures equivalently [ms]

Returns indices of trough cycles and peak cycles (S,) where S is the number of stimulus presentations

Return type Tuple[numpy.ndarray, numpy.ndarray]

attractor.**separate_snapshots**(*troughs*, *stimulus_end*, *stimulus_length*, *inter_presentation_interval*)

separate snapshots according to the distance of the time point of ceasure of the stimulus presentation to the time point at which the next snapshot starts - a snapshot is boolean spike mask over all cells indicating which cells spike over the interval between two troughs -into three groups: i) trough group - w/4 around any trough [t_trough-w/4, t_trough + w/4) ii) peak group - w/4 around any peak [t_trough - 3/4*w, t_trough-1/4*w) iii) null group - all snapshots not in i) and not in ii) ie. snapshots during which stimulus presentation occurs and does not end before t_snap_0 + w/4 (with t_snap_0 the time point at which snapshot starts) where w refers to the wavelength, t_trough is the time point of any trough and therefore marks the beginning and ending of a snapshot, respectively.

Parameters

- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] (C+1,) where C is # snapshots - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **stimulus_end** (*numpy.ndarray*) – time points of ceasure of the stimulus presentations [ms](S,) where S is the number of stimulus presentations
- **stimulus_length** (*float*) – length of stimulus_presentation [ms]
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets or ceasures equivalently [ms]

Returns indices of peak group, indices of trough group, indices of null group for the vector of snapshots (C,) and indices of peak cycle, trough cycle and null cycle (S,) where S is the number of stimulus presentations

Return type Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]

attractor.**fraction_significant_snapshots**(*pvalue*, *stimulus_pattern*, *pattern*, *significance=0.05*)

compute fraction of significant snapshots

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns

- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **significance** (*float*) – significance level at which the fration of significant snapshots is computed

Returns fraction of significant snapshots and corresponding pvalues (C,), where C is the number of snapshots

Return type Tuple[float, numpy.ndarray]

attractor.**indices_snapshots_blocked_stimulus**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *stimulus_onset*, *stimulus_length*)

compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for windows [-stimulus_length,t0],[t0,t0+stimulus_length],[t0+stimulus_length, t0+2*stimulus_length], note that the stimulus occurs in interval t_beg = t0 and t_end = t0 + stimulus_length - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **stimulus_onset** (*numpy.ndarray*) – onset times of the stimulus [ms]
- **stimulus_length** (*float*) – length of the stimulus [ms]
- **significance** – significance level at which the fration of significant snapshots is computed

Returns fraction of significant snapshots within time windows and corresponding pvalues (C,), where C is the number of snapshots ([t0-stimulus_length,t0], [t0,t0+stimulus_length], [t0+stimulus_length, t0+2*stimulus_length])

Return type Tuple[Tuple[float, float, float], Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]]

attractor.**fraction_significant_snapshots_blocked_stimulus**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *stimulus_onset*, *stimulus_length*, *significance=0.05*)

compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for windows [-stimulus_length,t0],[t0,t0+stimulus_length],[t0+stimulus_length, t0+2*stimulus_length], note that the stimulus occurs in interval t_beg = t0 and t_end = t0 + stimulus_length - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **stimulus_onset** (*numpy.ndarray*) – onset times of the stimulus [ms]
- **stimulus_length** (*float*) – length of the stimulus [ms]
- **significance** (*float*) – significance level at which the fration of significant snapshots is computed

Returns fraction of significant snapshots within time windows and corresponding pvalues (C,), where C is the number of snapshots ([t0-stimulus_length,t0], [t0,t0+stimulus_length], [t0+stimulus_length, t0+2*stimulus_length])

Return type Tuple[Tuple[float, float, float], Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray]]

attractor.**fraction_significant_snapshots_blocked_stimulus_detailed**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *stimulus_onset*, *stimulus_length*, *significance=0.05*)

compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for windows ([t0-stimulus_length,t0-stimulus_length/2], [t0-stimulus_length/2, t0], [t0,t0+stimulus_length], [t0+stimulus_length, t0+1.5*stimulus_length],[t0+1.5*stimulus_length,t0+2*stimulus_length])], note that the stimulus occurs in interval t_beg = t0 and t_end = t0 + stimulus_length - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)

- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **stimulus_onset** (*numpy.ndarray*) – onset times of the stimulus [ms]
- **stimulus_length** (*float*) – length of the stimulus [ms]
- **significance** (*float*) – significance level at which the fration of significant snapshots is computed

Returns fraction of significant snapshots within time windows and corresponding pvalues (C,), where C is the number of snapshots

Return type Tuple[Tuple[float, float, float, float, float], Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]]

([t0-stimulus_length,t0-stimulus_length/2], [t0-stimulus_length/2, t0], [t0,t0+stimulus_length], [t0+stimulus_length, t0+1.5*stimulus_length],[t0+1.5*stimulus_length,t0+2*stimulus_length])

attractor.**indices_snapshots_blocked_stimulus_sliding_window**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *stimulus_onset*, *t_end*, *inter_onset_interval*, *window_length*, *window_step*)

compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for a sliding window of length window_length and with step window_step note that the stimulus occurs in interval t_beg = t0 and t_end = t0 + stimulus_length - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **stimulus_onset** (*float*) – onset time of the first stimulus presentation [ms]
- **t_end** (*float*) – end of the simulation
- **inter_onset_interval** (*float*) – interval time between the onsets of any two subsequent stimulus presentations [ms]
- **window_length** (*float*) – length of the sliding window
- **window_step** (*float*) – step size of the sliding window

Returns indices of significant snapshots within time windows [stimulus_onset, stimulus_onset + window_step, stimulus_onset + 2 * window_step, ..., stimulus_onset + inter_onset_interval]

Return type Tuple[Tuple[float, Ellipsis], Tuple[numpy.ndarray, Ellipsis]]

attractor.**fraction_significant_snapshots_blocked_stimulus_sliding_window**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *stimulus_onset*, *t_end*, *inter_onset_interval*, *window_length*, *window_step*, *significance=0.05*)

compute fraction of significant snapshots for blocked stimulus given a stimulus length and stimulus onset times for a sliding window of length window_length and with step window_step note that the stimulus occurs in interval t_beg = t0 and t_end = t0 + stimulus_length - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] - allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **stimulus_onset** (*float*) – onset time of the first stimulus presentation [ms]
- **t_end** (*float*) – end of the simulation
- **inter_onset_interval** (*float*) – interval time between the onsets of any two subsequent stimulus presentations [ms]
- **window_length** (*float*) – length of the sliding window
- **window_step** (*float*) – step size of the sliding window
- **significance** (*float*) – significance level at which the fration of significant snapshots is computed

Returns fraction of significant snapshots within time windows and corresponding pvalues (C,), where C is the number of snapshots [stimulus_onset, stimulus_onset + window_step, stimulus_onset + 2 * window_step, …, stimulus_onset + inter_onset_interval]

Return type Tuple[Tuple[float, Ellipsis], Tuple[numpy.ndarray, Ellipsis]]

attractor.**fraction_significant_snapshots_across_intervals**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *interval*, *significance=0.05*)

compute fraction of significant snapshots across a set of intervals - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] (C+1,)- allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **interval** (*List[Tuple[float, float]]*) – set of intervals defining which snapshots are considered for the computation: a snapshot ([t_snap_beg, t_snap_end] - def. by parameter 'troughs') enclosing peak (t_peak - given in peaks) is considered if there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)
- **significance** (*float*) – significance level at which the fration of significant snapshots is computed

Returns fraction of significant snapshots within intervals and corresponding pvalues (C,), where C is the number of snapshots

Return type Tuple[float, numpy.ndarray]

attractor.**indices_snapshots_across_intervals**(*pvalue*, *stimulus_pattern*, *pattern*, *troughs*, *peaks*, *interval*)

extract idx of significant snapshots across a set of intervals - a snapshot ([t_snap_beg, t_snap_end]) is considered if for its enclosed peak (t_peak)

there is an interval ([t_beg, t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg<= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Parameters

- **pvalue** (*numpy.ndarray*) – pvalue for each snapshot (N, C) where C is the number of snapshots and N the number of patterns
- **stimulus_pattern** (*numpy.ndarray*) – stimulus pattern used to stimulate the network (pattern_length,) and one of the patterns in parameter 'pattern'
- **pattern** (*numpy.ndarray*) – patterns from which the scaling matrix (weights) is computed (N x pattern_length,)
- **troughs** (*numpy.ndarray*) – troughs of population rate [ms] (C+1,)- allow computing the start and end time of each snapshot (see *attractor.extract_snapshots()*)
- **peaks** (*numpy.ndarray*) – peaks of population rate [ms] used for determining the inclusion of snapshots that start before or end after an interval (edge case)
- **interval** (*List[Tuple[float, float]]*) – set of intervals defining which snapshots are considered for the computation: a snapshot ([t_snap_beg, t_snap_end] - def. by parameter 'troughs') enclosing peak (t_peak - given in peaks) is considered if there is an interval ([t_beg,

t_end]) such that t_peak < t_end and t_peak >= t_beg or t_snap_beg <= t_beg and t_snap_end >= t_end (ie snapshot encloses interval)

Returns set of boolean mask of pvalues (N,C) indexing all pvalues of stimulus_pattern of snapshots within an interval for each

Return type Tuple[float, numpy.ndarray]

attractor.**accuracy**(*snapshot*, *pattern*)
accuracy between snapshot and pattern

Parameters

- **snapshot** (*numpy.ndarray*) – population vector within one oscillatory cycle
- **pattern** (*numpy.ndarray*) – original pattern used for learning

attractor.**log_fac**(*n*)
logarithm of factorial approximated by Stirling's approximation ln(n!) ~ n ln(n) - n + 1/2 * ln(2*pi*n)

Parameters

- **precision** – precision of the approximation (affects number of terms of sterling's series used in approx)
- **n** (*int*) –

attractor.**log_choose**(*n*, *k*)
log(n choose k) := log(n! / (k! * (n-k)!)) = log(n!) - log(k!) -log((n-k)!)

Parameters

- **n** (*int*) – size of set of elements to choose from
- **k** (*int*) – size subset of elements tb chosen from set paramter n - applies elementwise if array of k values is provided

attractor.**choose**(*n*, *k*)
n choose k := n! / (k! * (n-k)!)

Parameters

- **n** (*int*) – size of set of elements to choose from
- **k** (*Union[int, numpy.ndarray]*) – size subset of elements tb chosen from set paramter n - applies elementwise if array of k values is provided

attractor.**p_value_snapshot_same_sparsity**(*similarity*, *sparsity*, *pattern*)
p_value of drawing a pattern X = {x_i}, i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l achieving a higher or equal similarity (by chance ~ H0) to parameter similarity pmf(p,k)= p^k (1-p)^l-k - unlike for a binomial distribution the order of trials matters

probability of an exact match with pattern: p('exact_match') = p^k (1-p)^l-k any match that is not an exact match can be expressed starting from the probability of an 'exact_match' and adjusting for n flips, where n-flipped vector is a vector received from pattern by flipping n elements: - all combinations of n elements distributed over 1s (p) and 0s (1-p) and chosen from k (#1s) and (l-k) (#0s) respectively - adjust k coefficient of probability to compensate for the respective deviations from 'exact_match' probability of n-flip: p(n) = sum_{i=max(0,n-(l-k))..min(k,n)} (k choose i) * (l-k choose n-i) * p^(k-i+(n-i)) (1-p)^(l-k+i-(n-i)) (i flips of 1 -> 0 - max number is k, n-i flips of 0->1

l-k >= n-i <-> i >= n - (l-k), which is a non-trivial constraint if l-k < n)

probability of achieving a higher or equal similarity s: p(N<=l-s) = sum_{n=0..l-s} sum_{i=0..min(k,n)} (k choose i) * (l-k choose n-i) * p^(k-i+(n-i)) (1-p)^(l-k+i-(n-i))

rewriting the products as sums of logarithms: log((k choose i) * (l-k choose n-i) * p^(k-i+(n-i)) (1-p)^(l-k+i-(n-i))) ~ log(k choose i) + log(l-k choose n-i) + (k-i+(n-i)) * log(p) + (l-k+i-(n-i)) * log(1-p)

Parameters

- **similarity** (*int*) – similarity of a snapshot and pattern pair
- **sparsity** (*float*) – sparsity parameter used to draw the pattern X = {x_i} i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l
- **pattern** (*numpy.ndarray*) – original pattern

attractor.**pvalue_snapshot_sparsity_missmatch**(*similarity*, *sparsity*, *pattern*, *spike_count*, *num_cycles*)

pvalue snapshot sparsity missmatch - probability of a flip ~ probability of spiking in the snapshot in a given cycle: ps

p(N<=l-s) = sum_{n=0..l-s} sum_{i=0..min(k,n)} (k choose i) * (l-k choose n-i) * p^(k-i+(n-i)) (1-p)^(l-k+i-(n-i)) where p = ps see pvalue_snapshot, pvalue_snapshost_same_sparsity for derivation

Note that the maximum number of possible flips n=0..l-s and 1-fips i=0..min(k,n) and their permutations do not change. Only the probability with which a snapshot exhibits these flips changes. So an exact match is still possible even though it is extremely unlikely.

On the unchangedness of flips: as the similarity measurement remains the same: s is unchanged, therefore also n=l-s. k is also unchanged as an exact match is theoretically possible even for very small ps, even though unlikely, therefore a similarity s of l (s==l) is possible and this allows for k 1-flips from this exact match.

Parameters

- **similarity** (*int*) – similarity of a snapshot and pattern pair
- **sparsity** (*float*) – sparsity parameter used to draw the pattern X = {x_i} i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l
- **pattern** (*numpy.ndarray*) – original pattern
- **spike_count** (*int*) – number of spikes that occurred across the population and simulation time
- **num_cycles** (*int*) – number of cycles that occurred during simulation (given oscillatory regime)

attractor.**pvalue_snapshot_sparsity_missmatch_single_cycle_count**(*similarity*, *sparsity*, *pattern*, *spike_count*)

pvalue snapshot sparsity missmatch with p snapshot-specific - probability of a flip ~ probability of spiking in the snapshot in a given cycle: ps

Parameters

- **similarity** (*int*) – similarity of a snapshot and pattern pair
- **sparsity** (*float*) – sparsity parameter used to draw the pattern X = {x_i} i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l
- **pattern** (*numpy.ndarray*) – original pattern
- **spike_count** (*int*) – number of spikes that occurred across the population and the snapshot

attractor.**pvalue_snapshot**(*l*, *k*, *s*, *p*)

pvalue of snapshot given similarity s with reference pattern ~ probability of observing a more or equally extreme (high) similarity assuming data is distriubted randomly (~ H0) note that spiking probability / sparsity can be considered as drawing a RV vector X = [x_i], i in [0,l], x_i ~ BER(p=sparsity), x_i in {0,1}

P(X >= s) = P(N <= l-s) where X is RV of similarity and N RV of number of flips from exact match between pattern and snapshot. To achieve similarity s we must observe exactly l-s flips. Any flip is either a flip from

1->0 or from 0->1. Given n flips, observing i, max(0, n-(l-k))<= i <= min(k,n), 1-flips implies also observing n-i 0-flips where k: #1s and l-k: #0s. There are k choose i * l-k choose n-i possible permutations to observe this. Bounds on i the 1-flips/distribution of n flips over 1s and 0s: - lower bound: max(0, n-(l-k))<= i ~ dictated by available # of 0s l-k so for n > l-k we will have at least i=n-(l-k) 1s - i <= min(k,n) ~ i upper bounded by # 1s in pattern k and number flips n

Parameters

- **l** (*int*) – pattern length
- **k** (*int*) – # 1s in pattern
- **s** (*int*) – similarity (: accuracy of snapshot given pattern ~ sum(snap==pattern))
- **p** (*float*) – probability of a flip ~ probability of spiking in the snapshot

attractor.**similarity**(*snapshot*, *pattern*)

similarity between snapshot and pattern (dot_product)

Parameters

- **snapshot** (*numpy.ndarray*) – population vector within one oscillatory cycle
- **pattern** (*numpy.ndarray*) – original pattern used for learning

attractor.**p_value_snapshot_med**(*similarity*, *sparsity*, *pattern*, *num_spikes_snapshot*, *num_cycles*)

Parameters

- **similarity** (*int*) – similarity of a snapshot and pattern pair
- **sparsity** (*float*) – sparsity parameter used to draw the pattern X = {x_i} i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l
- **pattern** (*numpy.ndarray*) – original pattern

attractor.**p_value_snapshot_dot_product**(*similarity*, *sparsity*, *pattern*, *spike_count*, *num_cycles*)

Parameters

- **similarity** (*int*) – similarity of a snapshot and pattern pair (dot_product)
- **sparsity** (*float*) – sparsity parameter used to draw the pattern X = {x_i} i in [0,l], x_i ~ BER(p=sparsity), xi in {0,1}^l
- **pattern** (*numpy.ndarray*) – original pattern

attractor.**similarity_threshold**(*sparsity*, *pattern*, *spike_count*, *num_cycles*, *significance=0.05*, *similarity=None*)

critical threshold, a similarity value in the given range param similarity, whose p value is the tightest lower bound on the given significance level ~ inverse cdf of p_value_snapshot

Parameters

- **sparsity** (*float*) – sparsity used to sample pattern from binomial distribution
- **pattern** (*numpy.ndarray*) – pattern for which critical threshold is computed
- **significance** (*float*) – significance level for which the critical similarity threshold is computed
- **similarity** (*numpy.ndarray*) – range of similarity values from which the critical threshold, a similarity value, whose p value is the tightest lower bound on the given significance level, is selected - assumption: the similarity array

is sorted in descending order wrt. the p values of the similarity values as returned by `p_value_snapshot()`

– default: full search assuming integer interval of similarity values [0, pattern.size - 1]

- **spike_count** (*int*) –
- **num_cycles** (*int*) –

Returns critical threshold and corresponding p value which is the tightest lower bound on the significance level (w/in given range, parameter similarity), parameter significance, returns (None, None) if no lower bound found ie p_value of similarity[n-1] > significance level

attractor.**similarity_conductance_scaling**(*pattern*)
similarity rule for conductance scaling ~ pair-wise similarity btw two neurons (ie. per synapse) averaged across patterns

Parameters **pattern** (*numpy.ndarray*) – patterns (N x pop_size) from which the weights are computed

attractor.**sim_vec**(*matrix*)
similarity of row vectors

r_ij = dot product of (row) vectors i,j of length l divided by l, where matrix is of dimensions (n,l)

Parameters **matrix** (*numpy.ndarray*) –

attractor.**compute_conductance_scaling**(*patterns*, *sparsity*)
compute the scaling factor of the conductance according to Battaglia, Treves 1998 (https://pubmed.ncbi.nlm.nih.gov/9472489/) original process: i) g_ij := 0 ii) for each pattern do:

a) delta_g_ij = g_EE / C_EE * (n_i^p / sparsity - 1) (n_j^p / sparsity - 1) b) g_ij = max(0, g_ij + delta_g_ij)

where g_ij is the conductance of synapse from neuron with index i to j

Here scaling factor s is computed: - g = g_EE / C_EE * s - process: i) s_ij := 0 ii) for each pattern do:

a) delta_s_ij = (n_i^p / sparsity - 1) (n_j^p / sparsity - 1) b) s_ij = max(0, s_ij + delta_s_ij)

(clipping equivalent to original process as g_EE/C_EE is a positive constant term therefore crossing of 0 (clipping) remains unchanged)

Parameters

- **patterns** (*numpy.ndarray*) – patterns tb used in compution shape: (p,size) where size is the size of the pattern (= size of E population) and p is the number of patterns
- **sparsity** (*float*) – sparsity of the patterns

Returns scaling factor s for conductances (shape: (size,size))

attractor.**compute_conductance_scaling_single_clip**(*patterns*, *sparsity*)
compute the scaling factor of the conductance by summing over patterns and clipping the result

Here scaling factor s is computed: - g = g_EE / C_EE * s - process: i) for each pattern compute s^p_ij = (n_i^p / sparsity - 1) (n_j^p / sparsity - 1)

ii) s_ij = sum_p s^p_ij

iii) s_ij = max(0, s_ij)

Parameters

- **patterns** (*numpy.ndarray*) – patterns tb used in compution shape: (p,size) where size is the size of the pattern (= size of E population) and p is the number of patterns
- **sparsity** (*float*) – sparsity of the patterns

Returns scaling factor s for conductances (shape: (size,size))

attractor.**compute_conductance_scaling_unclipped**(*patterns*, *sparsity*)

compute the scaling factor of the conductance by summing over patterns

Here scaling factor s is computed: - g = g_EE / C_EE * s - process: i) for each pattern compute s^p_ij = (n_i^p / sparsity - 1) (n_j^p / sparsity - 1)

ii) s_ij = sum_p s^p_ij

Parameters

- **patterns** (*numpy.ndarray*) – patterns tb used in compution shape: (p,size) where size is the size of the pattern (= size of E population) and p is the number of patterns
- **sparsity** (*float*) – sparsity of the patterns

Returns scaling factor s for conductances (shape: (size,size))

attractor.**normalize**(*matrix*, *frm=None*, *to=None*)

normalize (here squash) all values in matrix to [0,1] and if specified rescale to [frm,to]

Parameters

- **matrix** (*numpy.ndarray*) – matrix tb normalized
- **frm** (*float*) – lower bound of interval to which matrix is tb rescaled (requires setting to)
- **to** (*float*) – upper bound of interval to which matrix is tb rescaled (requires setting frm)

Returns matrix normalized to [0,1] or [frm,to] if specified

attractor.**z_score**(*matrix*)

compute z score: z := (x - mu) / sigma where mu = mean(matrix), sigma = std(matrix) (over all values in matrix), for all values x in matrix

Parameters **matrix** (*numpy.ndarray*) – matrix tb normalized

Returns z_score of the matrix

attractor.**extract_snapshot_masks**(*pop_rate*, *t_start*, *t_end*, *dt*)

extract snapshot masks of oscillation cycles, where one snapshot mask is a boolean mask for a specific cylce (trough-to-trough) with value at index i True iff neuron i spiked in the respective cycle

Parameters

- **pop_rate** (*numpy.ndarray*) – population rate from which snapshot masks are generated for interval [t_start, t_end] - resolution must be the simulation timestep, parameter dt
- **t_start** (*float*) – start time for analysis and extraction
- **t_end** (*float*) – end time for analysis and extraction
- **dt** (*float*) – time step of the simulation and resolution of snapshot masks

Returns snapshot masks of oscillation cycles as csr matrix (sparse): (C,T), where number of cycles C = troughs.size -1 and time bins T = (t_end-t_start)//dt + 1

attractor.**extract_snapshots**(*spike_train*, *pop_size*, *pop_rate*, *t_start*, *t_end*, *dt*)

Spike_train spike_train per neuron [ms]

Parameters

- **pop_size** (*int*) – size of the population - assumes neuron indices in [0, pop_size) whose str representations are keys of spike_train
- **t_start** (*float*) – start time for snapshot extraction [ms]
- **t_end** (*float*) – end time for snapshot extraction [ms]
- **dt** (*float*) – step size of simulation
- **spike_train** (*Dict[str, numpy.ndarray]*) –
- **pop_rate** (*numpy.ndarray*) –

Returns snapshots (C x pop_size), where C is the number of cylces

1.2 parse_equations

1.2.1 Module Contents

Classes

EquationEvaluator	A node visitor base class that walks the abstract syntax tree and calls a
VariableExtractor	A node visitor base class that walks the abstract syntax tree and calls a

Functions

evaluate_node(node, parameters)	evaluate abstract syntax tree node recurviely using params to resolve
extract_node(node, variables)	
evaluate_equations(equations, parameters)	
extract_variables_from_equations(equations)	

parse_equations.**evaluate_node**(*node*, *parameters*)
evaluate abstract syntax tree node recurviely using params to resolve external variables

parse_equations.**extract_node**(*node*, *variables*)

Parameters **variables** (*Dict[str, dict]*) –

Return type Tuple[str, int]

class parse_equations.**EquationEvaluator**(*params*)
Bases: ast.NodeVisitor

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found.

This function may return a value which is forwarded by the *visit* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

Per default the visitor functions for the nodes are `'visit_'` + class name of the node. So a *TryFinally* node visit function would be *visit_TryFinally*. This behavior can be changed by overriding the *visit* method. If no visitor function exists for a node (return value *None*) the *generic_visit* visitor is used instead.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversing. For this a special visitor exists (*NodeTransformer*) that allows modifications.

visit_Assign(*self*, *node*)

visit_AugAssign(*self*, *node*)

property report(*self*)

parse_equations.**evaluate_equations**(*equations*, *parameters*)

Parameters

- **equations** (*str*) –
- **parameters** (*Dict[str, Any]*) –

class parse_equations.**VariableExtractor**

Bases: ast.NodeVisitor

A node visitor base class that walks the abstract syntax tree and calls a visitor function for every node found. This function may return a value which is forwarded by the *visit* method.

This class is meant to be subclassed, with the subclass adding visitor methods.

Per default the visitor functions for the nodes are `'visit_'` + class name of the node. So a *TryFinally* node visit function would be *visit_TryFinally*. This behavior can be changed by overriding the *visit* method. If no visitor function exists for a node (return value *None*) the *generic_visit* visitor is used instead.

Don't use the *NodeVisitor* if you want to apply changes to nodes during traversing. For this a special visitor exists (*NodeTransformer*) that allows modifications.

visit_AugAssign(*self*, *node*)

property report(*self*)

parse_equations.**extract_variables_from_equations**(*equations*)

Parameters **equations** (*str*) –

1.3 BrianExperiment

1.3.1 Module Contents

Classes

TqdmCallBack	Provide progress bar updatable via callback based on `tqdm` via `tqdm.update()`
TimeTracker	track time durations of sequential stages of a process

continues on next page

Table 4 – continued from previous page

BrianExperiment	Implements Context Manager Interface for Brian2 Experiments especially setup and teardown as well as handling persistence

Attributes

_Data

class BrianExperiment.**TqdmCallBack**(*report_freq*, **args*, ***kwargs*)

Bases: tqdm.tqdm

Provide progress bar updatable via callback based on tqdm via tqdm.update()

Parameters **report_freq** (*brian2.units.fundamentalunits.Quantity*) –

update_cb(*self*, *elapsed*, *completed*, *start*, *duration*)

update progress bar

Parameters

- **elapsed** (*brian2.units.fundamentalunits.Quantity*) – total real time since start of the experiment
- **completed** (*float*) – fraction in [0,1] indicating completion
- **start** (*brian2.units.fundamentalunits.Quantity*) – start of the experiment in biological time
- **duration** (*brian2.units.fundamentalunits.Quantity*) – total duration of the experiment in biological time

class BrianExperiment.**TimeTracker**(*verbose=False*)

track time durations of sequential stages of a process

Parameters **verbose** (*bool*) –

add_timing(*self*, *process*)

add a new stage - this marks the end of the previous stage if it exists

Parameters **process** (*str*) –

finalize(*self*)

end the previous stage without beginnning a new stage

property timings(*self*)

timings of all tracked and ended processes

property verbose(*self*)

indicates whether the TimeTracker is used in verbose mode, where TimeTracker will print progress (verbose is set in TimeTracker.__init__())

BrianExperiment.**_Data**

class BrianExperiment.**BrianExperiment**(*dt=0.01 * ms*, *report_progress=False*, *progress_bar=False*, *persist=False*, *path=''*, *object_path='/'*, *neuron_eqs=[]*, *neuron_params=[]*, *neuron_eq_module='differential_equations.eif_equations'*, *neuron_param_module='differential_equations.eif_parameters'*)

Implements Context Manager Interface for Brian2 Experiments especially setup and teardown as well as handling persistence All data monitored by NeuronPopulation instances is automatically persisted as well as time steps of defaultclock. Any additional data can be persisted by adding it to a dict passed to *BrianExperiment*. It relies on using the default clock for all network components - we save time array only once.

It is crucial that all network definitions (instances of `NeuronPopulation`, `Synapse`,...) are bound to a unique name, as logic in this class makes use of these names, eg. for persisting.

Note if neuron equations and parameters reside elsewhere (see `neuron_eq_module`, `neuron_param_module`) then pass the corresponding modules to `__init__()`

Example

```
with TestEnv():
    with BrianExperiment(persist=True, path="file.h5", object_path="/run_1/data")␣
↪as exp:
        exp.persist_data["mon"] = np.arange(10)
    with FileMap("file.h5") as f:
        print(f["run_1"]["data"]["persist_data"]["mon"])
```

```
[0 1 2 3 4 5 6 7 8 9]
```

Parameters

- **dt** (*brian2.units.fundamentalunits.Quantity*) –
- **report_progress** (*bool*) –
- **progress_bar** (*bool*) –
- **persist** (*bool*) –
- **path** (*str*) –
- **object_path** (*str*) –
- **neuron_eqs** (*List[str]*) –
- **neuron_params** (*List[str]*) –
- **neuron_eq_module** (*str*) –
- **neuron_param_module** (*string*) –

class **PersistData**(*persist*, *exp*)

Bases: `dict`

Dictionary-like class whose setability can be switched on or off based on whether persist is set on *BrianExperiment* and whether the instance of the class is accessed within the context of *BrianExperiment*

Parameters

- **persist** (*bool*) – whether or not persist is set on the instance of *BrianExperiment*
- **exp** – instance of *BrianExperiment*

__setitem__(*self*, *key*, *value*)
Set self[key] to value.

__repr__(*self*)
Return repr(self).

static resolve_module_name(*mod*)

Parameters **mod** (*types.ModuleType*) –

Return type str

property time_elapsed(*self*)
str representing time elapsed during simulation, None if *BrianExperiment.run()* not executed yet

property persist_data(*self*)
special dictionary (*BrianExperiment.PersistData*) that may be populated within the context and whose entries will be persisted on exit if persist is set

property path(*self*)
path to underlying h5 file - especially useful when no path passed in BrianExperiment.__init__() and it is autogenerated

property dt(*self*)
timestep to be used in simulation

_retrieve_callers_frame(*self*)

_retrieve_callers_context(*self*)

_save_context(*self*)

_collect_devices(*self*)

_reset_context(*self*)
delete the underlying brian2 objects of the wrapper classes defined within the context of the experiment

_get_namespace(*self*)

run(*self*, *t=0.01 * ms*, *report_freq=100 * ms*)
run brian2 network via brian2.network.run()

Parameters

- **t** (*brian2.units.fundamentalunits.Quantity*) – time for which the simulation is tb run
- **report_freq** – frequency at which the report is updated, irrelevant if progress_report=False in BrianExperiment.__init__()

__enter__(*self*)

static _destructure_persist(*items*)

Parameters **items** (*List[Tuple[persistence.Writer, _Data]]*) –

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

1.4 opath

Functions for transforming object paths (path-like strings)

1.4.1 Module Contents

Functions

verify(opath, path_type = 'abs_path')	verify path in object tree of hdf5 file
split(opath)	split path in object tree of hdf5 file into path components
join(path, head, *args)	join arbitrarily many path components in the object tree

exception opath.**OpathError**
Bases: Exception

Common base class for all non-exit exceptions.

opath.**verify**(*opath*, *path_type='abs_path'*)
verify path in object tree of hdf5 file

Parameters

- **opath** (*str*) – object path in object tree of hd5 file
- **path_type** (*str*) – path type tb verified choose from single_component | abs_path | rel_path | any_path, where any_path is the superset of the other options, abs_path and rel_path are supersets of single_component paths

Returns error msg, empty if valid

Return type str

opath.**split**(*opath*)
split path in object tree of hdf5 file into path components will append root component '/' for absolute paths

Parameters **opath** (*str*) – object path in object tree of hd5 file

Returns list of object path components

Return type List[str]

opath.**join**(*path*, *head*, **args*)
join arbitrarily many path components in the object tree (at least two)

Parameters

- **path** (*str*) – base path in the object tree
- **head** (*str*) – single path component
- **args** (*str*) – (opt.) further single path components

Returns compound path

Return type str

1.5 analysis

1.5.1 Module Contents

Functions

compute_stimulus_characteristics(stimulus_block_interval)	
detect_peaks(signal, dt)	detect peaks of a signal given sampling interval dt with a minimum distance between peaks
gaussian_smoothing(instantaneous_rate, window_size, one_sigma_window, dt)	smoothes the instantaneous rate by window_size around any point ([-window_size, window_size]) using a gaussian window
instantaneous_rate_from_spike_train(t, dt, spike_train)	computes instantaneous rate from spike train
cell_rate_from_spike_train(t_start, t_end, ids, spike_train)	computes cell rate from spike train for each neuron individually
mt_spectrum(rate, dt, nfft = None)	spectrum of the population rate computed with a multi taper method
mt_psd(rate, dt, nfft = None)	Power spectral density of the population rate computed with a multi taper method
multitaper_power_spectral_density(rate, dt, w_sliding = None, w_step = 0.1, nfft = None)	Power spectral density of the population rate computed with a multi taper method
population_rate_avg_over_time(rate)	population rate average over time
synchronization_frequency(frequency, power_spectral_density)	synchronization frequency : peak frequency of the population rate power spectral density
snr(psd, frequency, bin_size = 10.0)	signal-to-noise ratio from power spectral density
cross_power_spectral_density()	
restrict_frequency(frequency, psd, f_lower_bound = None, f_upper_bound = None)	restrict frequencies and corresponding psd to those for which f_lower_bound <= frequency <= f_upper_bound
compute_synaptic_input(source_ids, target_ids, syn_const)	synaptic input for each distinct id in source_ids by target_id
synaptic_conductance(target_ids, source_ids, cell_rate, synaptic_input, conductance)	total synaptic conductance for a specific synaptic input type for a group of synapses on a per target neuron basis
effective_total_synaptic_conductance(source_ids_e_e, target_ids_e_e, source_ids_i_e, target_ids_i_e, cell_rate_e, cell_rate_i, syn_const_e_e, syn_const_i_e, conductance_e_e, conductance_i_e)	effective total synaptic conductance for a group of synapses on a per target neuron basis,

analysis.**compute_stimulus_characteristics**(*stimulus_block_interval*)

Parameters **stimulus_block_interval** (*numpy.ndarray*) –

analysis.**detect_peaks**(*signal*, *dt*)

detect peaks of a signal given sampling interval dt with a minimum distance between peaks of half the wavelength of the fundamental frequency of the signal

Parameters

- **signal** (*numpy.ndarray*) –
- **dt** (*float*) –

analysis.**gaussian_smoothing**(*instantaneous_rate*, *window_size*, *one_sigma_window*, *dt*)
smoothes the instantaneous rate by window_size around any point ([-window_size, window_size]) using a gaussian window with one_sigma_window as sigma

Parameters

- **instantaneous_rate** (*numpy.ndarray*) – instantaneous rate of the population
- **window_size** (*float*) – size of the window for smoothing in [ms]
- **one_sigma_window** (*float*) – size of the window encompassing one sigma [ms]
- **dt** (*float*) – step size of the simulation in [ms] and duration between recordings

gaussian window implemented acc to https://www.mathworks.com/help/signal/ref/gausswin.html

(equivalent to brian2.PopulatonRateMonitor.smooth_rate(window='gaussian', width=w) for window_size=2*w and one_sigma_window=w)

analysis.**instantaneous_rate_from_spike_train**(*t*, *dt*, *spike_train*)
computes instantaneous rate from spike train

Parameters

- **t** (*float*) – simulation time [ms]
- **dt** (*float*) – simulation time step [ms]
- **spike_train** (*Dict[str, numpy.ndarray]*) – spike trains of individual neurons

Returns instantaneous firing rate - population average rate for each time step

analysis.**cell_rate_from_spike_train**(*t_start*, *t_end*, *ids*, *spike_train*)
computes cell rate from spike train for each neuron individually

Parameters

- **t_start** (*float*) – start time for analysis[ms]
- **t_end** (*float*) – end time for analysis [ms]
- **ids** (*numpy.ndarray*) – neuron ids of the entire population
- **spike_train** (*Dict[str, numpy.ndarray]*) – spike trains of individual neurons

Returns ids and corresponding cell rate for each cell (time averaged)

Return type Tuple[numpy.ndarray, numpy.ndarray]

analysis.**mt_spectrum**(*rate*, *dt*, *nfft=None*)
spectrum of the population rate computed with a multi taper method

Parameters

- **rate** (*numpy.ndarray*) – population rate
- **dt** (*float*) – time step / interval of successive measures of the population rate
- **nfft** (*int*) – length of the output of fft (n-point discrete, where n = nft) - set only if you desire a specific nfft - defaults to 2 ** sp, where sp is smallest num for which rate.size <= 2 ** sp

Returns frequencies, complex spectrum, weights, eigenvalues of multitaper method

analysis.**mt_psd**(*rate*, *dt*, *nfft=None*)
Power spectral density of the population rate computed with a multi taper method

Parameters

- **rate** (*numpy.ndarray*) – population rate
- **dt** (*float*) – time step / interval of successive measures of the population rate
- **nfft** (*int*) – length of the output of fft (n-point discrete, where n = nft) - set only if you desire a specific nfft - defaults to 2 ** sp, where sp is smallest num for which rate.size <= 2 ** sp

Returns frequencies and the power spectral density (at the respective frequencies)

analysis.**multitaper_power_spectral_density**(*rate*, *dt*, *w_sliding=None*, *w_step=0.1*, *nfft=None*)
Power spectral density of the population rate computed with a multi taper method computed over the entire time series or for separate (yet overlapping) time intervals using a sliding window without padding when parameter w_sliding is set

Parameters

- **rate** (*numpy.ndarray*) – population rate
- **dt** (*float*) – time step / interval of successive measures of the population rate
- **w_sliding** (*int*) – sliding window used for computing psd discretized over time (without padding) - when not set, defaults to computing psd over entire time series
- **w_step** (*float*) – step size of the sliding window as a fraction of the sliding window size (param w_sliding) - irrelevant when w_sliding not set
- **nfft** (*int*) – length of the output of fft (n-point discrete, where n = nfft) - set only if you desire a specific nfft, eg to increase the resolution

Returns frequencies and the power spectral density (for entire time series psd shape: (nfft/2,1); for separate intervals psd shape: (nfft/2, intervals) (at the respective frequencies)

analysis.**population_rate_avg_over_time**(*rate*)
population rate average over time

Parameters **rate** (*numpy.ndarray*) – population rates over time (shape: (number samples,))

Returns time average of the population rate

analysis.**synchronization_frequency**(*frequency*, *power_spectral_density*)
synchronization frequency : peak frequency of the population rate power spectral density

Parameters

- **frequency** (*numpy.ndarray*) – frequencies whose power is given by value at respective index of parameter power_spectral_density
- **power_spectral_density** (*numpy.ndarray*) – power of respective frequencies in parameter frequency

Returns synchronization frequency and its power

Return type Tuple[numpy.ndarray, numpy.ndarray]

analysis.**snr**(*psd*, *frequency*, *bin_size=10.0*)
signal-to-noise ratio from power spectral density

snr = P_signal / P_noise, where P_signal is the total power in the bin around the peak frequency and P_noise is the total power across the remainder of the spectrum

Parameters

- **psd** (*numpy.ndarray*) – power spectral density of the signal
- **frequency** (*numpy.ndarray*) – corresponding frequencies of the psd [Hz]

- **bin_size** (*float*) – size of the bin [Hz] around the peak frequency used to compute the signal

analysis.**cross_power_spectral_density**()

analysis.**restrict_frequency**(*frequency*, *psd*, *f_lower_bound=None*, *f_upper_bound=None*)
restrict frequencies and corresponding psd to those for which f_lower_bound <= frequency <= f_upper_bound

Parameters

- **frequency** (*numpy.ndarray*) – frequencies which are tb restricted
- **psd** (*numpy.ndarray*) – power spectral density that is tb restricted based on the associated frequency value, expects the power across frequencies to be on axis 0
- **f_lower_bound** (*float*) – lower bound on frequency
- **f_upper_bound** (*float*) – upper bound on frequency

Returns frequencies and corresponding psd restricted to the interval defined by the bounds

analysis.**compute_synaptic_input**(*source_ids*, *target_ids*, *syn_const*)
synaptic input for each distinct id in source_ids by target_id

Parameters

- **source_ids** – ids representing the source of synaptic connections - each id represents the source neuron of a synapse
- **target_ids** – ids representing the target of synaptic connections - each id represents the target neuron of a synapse
- **syn_const** (*Union[float, numpy.ndarray]*) – synaptic input constant, input to the target neuron when the source neuron spikes - either a scalar value (same for all synapses) or one value per synaptic connection

analysis.**synaptic_conductance**(*target_ids*, *source_ids*, *cell_rate*, *synaptic_input*, *conductance*)
total synaptic conductance for a specific synaptic input type for a group of synapses on a per target neuron basis

Parameters

- **target_ids** (*numpy.ndarray*) – target ids for which the total conductance is computed (unique targets) sorted in ascending order
- **source_ids** (*numpy.ndarray*) – source ids (unique sources) sorted in ascending order
- **cell_rate** (*numpy.ndarray*) – cell rate of the source population
- **synaptic_input** (*numpy.ndarray*) – synaptic input per target and source neuron (targets x source)
- **conductance** (*float*) – conductance of the respective synaptic input type

Returns target_ids and corresponding total conductance of the respective synaptic input (for this synapse group)

analysis.**effective_total_synaptic_conductance**(*source_ids_e_e*, *target_ids_e_e*, *source_ids_i_e*, *target_ids_i_e*, *cell_rate_e*, *cell_rate_i*, *syn_const_e_e*, *syn_const_i_e*, *conductance_e_e*, *conductance_i_e*)
effective total synaptic conductance for a group of synapses on a per target neuron basis, e refers to the excitatory population and i to the inhibitory population in an EI-Network

Parameters

- **source_ids_e_e** (*numpy.ndarray*) – ids representing the source of e-e synaptic connections - each id represents the source neuron of a synapse

- **target_ids_e_e** (*numpy.ndarray*) – ids representing the target of e-e synaptic connections - each id represents the target neuron of a synapse
- **source_ids_i_e** (*numpy.ndarray*) – ids representing the source of i-e synaptic connections - each id represents the source neuron of a synapse
- **target_ids_i_e** (*numpy.ndarray*) – ids representing the target of i-e synaptic connections - each id represents the target neuron of a synapse
- **cell_rate_e** (*numpy.ndarray*) – cell rate of the excitatory population
- **cell_rate_i** (*numpy.ndarray*) – cell rate of the inhibitory population
- **syn_const_e_e** (*float*) – synaptic input (constant) to the target neurons when the source neuron spikes for e-e synapses - either a scalar value (same for all synapses) or one value per synaptic connection
- **syn_const_i_e** (*float*) – synaptic input (constant) to the target neurons when the source neuron spikes for i-e synapses - either a scalar value (same for all synapses) or one value per synaptic connection
- **conductance_e_e** (*float*) – conductance for e-e synapses
- **conductance_i_e** (*float*) – conductance for i-e synapses

Returns effective total synaptic conductance per neuron of the entire e population

1.6 mp

1.6.1 Module Contents

Classes

Environment	Implements Context Manager Interface in a functional style
Logger	Implements Context Manager Interface for a logger
CaptureStandardStreams	Implements Context Manager Interface for capturing standard streams stdout and stderr
MultiPipeCommunication	create two linked communication objects with a set of pipes available as attributes under the provided stream names
ProcessExperiment	Process executing a target function (parameter target) and communicating stdout, stderr and current file path being processed
Progress	
Pool	Pool of instances of *ProcessExperiment*

Functions

float_to_path_component(x)	
path_component_to_float(pc)	
file_name_generator(instance)	Generate a file name from key value pairs
file_name_parser(fname)	Parse key value pairs from a file name

Attributes

log_path

mp.**log_path**

class mp.**Environment**(*enter=None*, *on_error=None*, *exit=None*)
Implements Context Manager Interface in a functional style

Parameters

- **enter** (*Callable*) –
- **on_error** (*Callable*) –
- **exit** (*Callable*) –

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

class mp.**Logger**(*path*, *stream_names*, *levels=None*)
Implements Context Manager Interface for a logger

Parameters

- **path** (*str*) –
- **stream_names** (*List[str]*) –
- **levels** (*List[str]*) –

logall(*self*, *process*, *stream_name*, *msgs*, *level=None*)

Parameters

- **process** (*int*) – process id
- **stream_name** (*str*) – name of the stream
- **msgs** (*List[Tuple[str, float]]*) – messages with associated timestamps (POSIX timestamp from eg. `time.time()`)
- **level** (*str*) – logging level

log(*self*, *process*, *stream_name*, *tstamp*, *msg*, *level=None*)

Parameters

- **process** (*int*) – process id
- **stream_name** (*str*) – name of the stream
- **tstamp** (*float*) – tstamp of the event (POSIX timestamp from eg. `time.time()`)
- **msg** (*str*) – msg
- **level** (*str*) – logging level

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

class mp.**CaptureStandardStreams**(*stdout=True*, *stderr=False*)

Implements Context Manager Interface for capturing standard streams stdout and stderr

Parameters

- **stdout** (*Union[bool, str]*) –
- **stderr** (*Union[bool, str]*) –

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

class mp.**MultiPipeCommunication**

create two linked communication objects with a set of pipes available as attributes under the provided stream names eg. for a stream name 'name' parent_com_obj.name and child_com_obj.name will return either end of the associated pipe

class **Communication**(*control_pipe*, *streams*)

Parameters

- **control_pipe** (*multiprocessing.connection.Connection*) –
- **streams** (*Dict[str, multiprocessing.connection.Connection]*) –

send(*self*, *msg*, *stream*)

Parameters

- **msg** (*str*) –
- **stream** (*multiprocessing.connection.Connection*) –

poll(*self*, *stream*, *wait=0*)

Parameters

- **stream** (*multiprocessing.connection.Connection*) –
- **wait** (*int*) –

recv(*self*, *stream*)

Parameters **stream** (*multiprocessing.connection.Connection*) – stream tb received from

Returns messages with associated timestamps

Return type List[Tuple[str, float]]

recvlines(*self*, *stream*, *keep_empty=True*)

Receive by line

Parameters

- **stream** (*multiprocessing.connection.Connection*) – stream tb received from

- **keep_empty** (*bool*) – whether or not to keep empty lines

Returns messages line by line with associated timestamps
Return type List[Tuple[List[str], float]]

closed(*self*, *stream*)

Parameters **stream** (*multiprocessing.connection.Connection*) –

close(*self*)

send_terminate(*self*)

should_terminate(*self*)

recv_control(*self*)

class mp.**ProcessExperiment**(*idx*, *num_procs*, *target*, *kwargs*, *com=None*)
Bases: multiprocessing.Process

Process executing a target function (parameter target) and communicating stdout, stderr and current file path being processed via dedicated pipes wrapped in an instance of *MultiPipeCommunication*

Parameters

- **idx** (*int*) –
- **num_procs** (*int*) –
- **target** (*Callable*) –
- **kwargs** (*Dict[str, Any]*) –
- **com** (MultiPipeCommunication) –

supported_stream_names = ['stdout', 'stderr', 'curfile']

close(*self*)
Close the Process object.

This method releases resources held by the Process object. It is an error to call this method if the child process is still running.

run(*self*)
execute target function (parameter target) for each nth instance with offset p of the cartesian product of the value ranges in parameter parameters, where p is the index of the process and n is the number of processors available

mp.**float_to_path_component**(*x*)

Parameters **x** (*float*) –

mp.**path_component_to_float**(*pc*)

Parameters **pc** (*str*) –

mp.**file_name_generator**(*instance*)
Generate a file name from key value pairs (reversed by *file_name_parser()*)

mp.**file_name_parser**(*fname*)
Parse key value pairs from a file name (reversed by *file_name_generator()*)

Parameters **fname** (*str*) –

class mp.**Progress**(*n*)

Parameters **n** (*int*) –

update(*self*, *it*)

static format_duration(*values*, *labels*, *unit_lengths*)
create string representation of time duration given the time increments (labels), their respective values (values) and their unit lengths (unit_lengths)

Returns string representation of time with format: y, d, h, m, s - where only duration components whose first increment is reached are used

class mp.**Pool**(*base_path*, *parameters*, *target*, *kwargs*, *log_path=log_path*, *num_procs=None*, *file_name_generator=file_name_generator*, *progress=True*)
Pool of instances of *ProcessExperiment*

Parameters

- **base_path** (*str*) –
- **parameters** (*Dict[str, List[Any]]*) –
- **target** (*Callable*) –
- **kwargs** (*Dict[str, Any]*) –
- **log_path** (*str*) –
- **num_procs** (*int*) –
- **file_name_generator** (*Callable[[List[Tuple[str, Any]]], str]*) –
- **progress** (*bool*) –

next_process(*self*, *idx*)

update_signal_handler(*self*, *processes*)

run(*self*)

static create_signal_handler(*processes*)

Parameters **processes** (*List[ProcessExperiment]*) –

1.7 experiments_eif

1.7.1 Module Contents

Functions

run_exp_eif(simtime, path, rpe, rpi, esize)	
generate_patterns(esize, sparsity = 0.2, numpatterns = 20)	
generate_fixed_patterns(esize, sparsity = 0.2, numpatterns = 20)	

continues on next page

Table 11 – continued from previous page

run_exp_eif_attr(simtime, path, rpe, rpi, esize, sparsity, pattern, weighted = True, norm = 1.0)	**param weighted** whether or not to use weighted synapses - synaptic scaling according to patters
run_exp_eif_attr_blocked_stimulus(simtime, path, rpe, rpi, esize, sparsity, pattern, stimuluspatternidx, perturbation, beta, minusbeta, continuousstim = False, weighted = True, norm = 1.0)	**param stimuluspatternidx** index of the pattern in parameter 'pattern' tb used for stimulus presentation after opt. perturbation

experiments_eif.**run_exp_eif**(*simtime*, *path*, *rpe*, *rpi*, *esize*)

Parameters

- **simtime** (*float*) –
- **path** (*str*) –
- **rpe** (*float*) –
- **rpi** (*float*) –
- **esize** (*int*) –

experiments_eif.**generate_patterns**(*esize*, *sparsity=0.2*, *numpatterns=20*)

Parameters

- **esize** (*int*) –
- **sparsity** (*float*) –
- **numpatterns** (*int*) –

experiments_eif.**generate_fixed_patterns**(*esize*, *sparsity=0.2*, *numpatterns=20*)

Parameters

- **esize** (*int*) –
- **sparsity** (*float*) –
- **numpatterns** (*int*) –

experiments_eif.**run_exp_eif_attr**(*simtime*, *path*, *rpe*, *rpi*, *esize*, *sparsity*, *pattern*, *weighted=True*, *norm=1.0*)

Parameters

- **weighted** (*bool*) – whether or not to use weighted synapses - synaptic scaling according to patters
- **simtime** (*float*) –
- **path** (*str*) –
- **rpe** (*float*) –

- **rpi** (*float*) –
- **esize** (*int*) –
- **sparsity** (*float*) –
- **pattern** (*numpy.ndarray*) –
- **norm** (*float*) –

experiments_eif.**run_exp_eif_attr_blocked_stimulus**(*simtime*, *path*, *rpe*, *rpi*, *esize*, *sparsity*, *pattern*, *stimuluspatternidx*, *perturbation*, *beta*, *minusbeta*, *continuousstim=False*, *weighted=True*, *norm=1.0*)

Parameters

- **stimuluspatternidx** (*numpy.ndarray*) – index of the pattern in parameter 'pattern' tb used for stimulus presentation after opt. perturbation
- **perturbation** (*float*) – percentage of perturbation used for computing the number of indices to be perturbated in the pattern
- **beta** (*float*) – additional excitation to 1s in stimulus_pattern as multiple of synaptic input (picked up by brian)
- **minus_beta** – additional inhibition to 0s in stimulus_pattern as multiple of synaptic input (picked up by brian)
- **weighted** (*bool*) – whether or not to use weighted synapses - synaptic scaling according to patters
- **simtime** (*float*) –
- **path** (*str*) –
- **rpe** (*Union[float, Dict[str, float]]*) –
- **rpi** (*Union[float, Dict[str, float]]*) –
- **esize** (*int*) –
- **sparsity** (*float*) –
- **pattern** (*numpy.ndarray*) –
- **minusbeta** (*float*) –
- **continuousstim** (*bool*) –
- **norm** (*float*) –

1.8 persistence

Entrypoint for file persistence with h5 files

Additionally exports:

- persistence.opath - *opath* provides utilities for dealing with object paths
- persistence.validate_file_path - *utils.validate_file_path()* validates file paths
- persistence.generate_sequential_file_name - *utils.generate_sequential_file_name()* generates file name sequentially

1.8.1 Module Contents

Classes

Array	Placeholder for `tables.array.Array()`
VArray	Placeholder for `tables.vlarray.VlArray()`
Node	Placeholder for `tables.groups.Group()`
Reader	Implements a Mapping Interface for the passed h5 file enabling indexing by key
Writer	Implements a Mutable Mapping Interface for the passed h5 file enabling indexing by key,
FileMap	Implements Context Manager Interface for *Reader* and *Writer* which on entering opens the

Functions

get_nodes(file, object_path)	Retrieves nodes and leaves attached below the node specified by the arg path in the

class persistence.**Array**(*obj*, **args*, ***kwargs*)
Placeholder for `tables.array.Array()` used by Reader and Writer class to enable Mapping Interface while also allowing arbitrary nesting

Parameters **obj** (*numpy.ndarray*) – array to be stored

class persistence.**VArray**(**args*, *obj=None*, ***kwargs*)
Placeholder for `tables.vlarray.VlArray()` used by Reader and Writer class to enable Mapping Interface while also allowing arbitrary nesting

Parameters **obj** (*Union[numpy.ndarray, None]*) – array to be stored (optional)

class persistence.**Node**
Placeholder for `tables.groups.Group()` used by Reader and Writer class to enable Mapping Interface while also allowing arbitrary nesting

class persistence.**Reader**(*file*, *object_path*)
Implements a Mapping Interface for the passed h5 file enabling indexing by key

Parameters

- **file** (*tables.file.File*) –
- **object_path** (*str*) –

__len__(*self*)

__iter__(*self*)

up(*self*)

_extract_value(*self*, *key*, *nodes*, *leaves*, *recursive=False*)
extract values

Parameters

- **recursive** (*bool*) – whether to read all descendants into memory recursively
- **key** (*str*) –

- **nodes** (*Tuple[Dict[str, tables.group.Group]]*) –
- **leaves** (*Dict[str, Union[tables.array.Array, tables.vlarray.VLArray]]*) –

Returns instance of *persistence.Reader* or a terminal node read into memory, if recursive set will return dictionary with all descendants read into memory or a terminal node read into memory

__getitem__(*self*, *key*)

Parameters **key** (*str*) –

keys(*self*)

items(*self*)

Note that items will extract all values of terminal nodes (arrays) into memory at the current level

This is not memory efficient! Avoid!

values(*self*)

Note that this will extract all values of terminal nodes (arrays) into memory at the current level

This is not memory efficient! Avoid!

load(*self*)

convert instance of this class to a dictionary - fully loads all descendants recursively

Returns dictionary containing all descendants of the instance of this class recursively

_as_dict(*self*, *slice_length=10*, *full_load=False*)

Create a dictionary from .h5 file abstraction creating string representations of leaf nodes and slicing arrays and strings

Parameters

- **slice_length** – length of slices used to represent arrays in leaf nodes as str and twice the length is used for slicing strings
- **full_load** – if True reads the entire array from the underlying as `numpy.ndarray`

full_load(*self*)

__repr__(*self*)

Return repr(self).

class persistence.**Writer**(*file*, *object_path*)

Bases: *Reader*

Implements a Mutable Mapping Interface for the passed h5 file enabling indexing by key, setting key value pairs as well as deleting key value pairs Leaf nodes are stored as `numpy.ndarray`.

Parameters

- **file** (*tables.file.File*) –
- **object_path** (*str*) –

_create_opath(*self*, *file*, *object_path*)

Parameters

- **file** (*tables.file.File*) –
- **object_path** (*str*) –

__delitem__(*self*, *key*)

Parameters **key** (*str*) –

__setitem__(*self*, *key*, *value*)

Parameters

- **key** (*str*) –
- **value** (*Union[numpy.ndarray, Array, VArray, Node, Dict, List, Tuple, str]*) –

class persistence.**FileMap**(*path*, *mode='read'*, *object_path='/'*)

Implements Context Manager Interface for *Reader* and *Writer* which on entering opens the h5 file and returns an instance of *Writer* or *Reader* tb used within the context as well as closes the h5 file when the context is left

supported modes are "write" [open file truncating and read & write] "modify" : open file and read & write "read" : open file in read-only

indexing just like nested dictionary: getting, setting and deleting items is supported

analagous to file system tree with inner nodes/directories (nodes: Node) and leaves/files (arrays: ndarray, Array, VArray)

Example

```
with TestEnv():
    with FileMap("file.h5", mode="write") as f:
        f["mydata"] = Node()
        md = f["mydata"]
        md["run_x"] = Node()
        m = md["run_x"]
        m["spikes"] = np.arange(10)
    with FileMap("file.h5", mode="read") as f:
        print(f["mydata"]["run_x"]["spikes"])
```

```
[0 1 2 3 4 5 6 7 8 9]
```

file structure: /mydata/run_x/spikes -> array([...]) nested dictionary: {"mydata":{"run_x":{"spikes":array([...])}}}

Assignment of nested dictionary of type XDict := Union[np.ndarray, Dict[str, XDict]]

```
with TestEnv():
    with FileMap("file.h5", mode="write") as f:
        f["mydata"] = { "run_x" : { "spikes": np.arange(10) }, "array": np.
↪arange(5) }
    with FileMap("file.h5", mode="read") as f:
        print(f)
```

```
{
  "mydata": {
    "run_x": {
      "spikes": "array([0 1 2 3 4 5 6 7 8 9]) (10,) dtype:int64"
    },
```

(continues on next page)

(continued from previous page)

```
    "array": "array([0 1 2 3 4]) (5,) dtype:int64"
  }
}
```

Basic Navigation

```
with TestEnv():
    with FileMap("file.h5", mode="write") as f:
        f["mydata"] = { "run_x" : { "spikes": np.arange(10) }, "array": np.
↪arange(5) }
    with FileMap("file.h5", mode="read") as f:
        md = f["mydata"]            # move down the object tree / index nested␣
↪dictionary
        g = md.up()                 # move up the object tree
        print(f.opath)
        print(g.opath)
```

```
/
/
```

Parameters

- **path** (*str*) –
- **mode** (*str*) –
- **object_path** (*str*) –

__enter__(*self*)

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

persistence.**get_nodes**(*file*, *object_path*)

Retrieves nodes and leaves attached below the node specified by the arg path in the object tree of the arg file object

Parameters

- **file** (*tables.file.File*) – representation of the underlying h5 file
- **path** – path to the current node within the object tree
- **object_path** (*str*) –

Returns a list of *Node* and a list of arrays `np.ndarray`, *Array*, `VlArray` attached at the current path

Return type Tuple[Dict[str, tables.group.Group], Dict[str, Union[tables.array.Array, tables.vlarray.VLArray]]]

1.9 utils

1.9.1 Module Contents

Classes

TestEnv	Implements Context Manager Interface to setup and teardown a test environment

Functions

run_cmd(cmd)	
validate_file_path(path, ext = '')	Validate file path - whether
generate_sequential_file_name(base_path, base_name, ext)	Generate the next file name sequentially given a directory name and base file name
retrieve_callers_frame(condition)	retrieve the frame satisfying a condition - if no such frame raises Exception
retrieve_callers_context(frame_info)	retrieve the context for a frame
clean_brian2_quantity(x)	clean quantity of its unit
convert_and_clean_brian2_quantity(x)	convert quantity to base unit and clean of its base unit
unwrap_brian2_variable_view(x)	unwrap instance of `brian2.core.variables.VariableView`
get_brian2_unit(x)	get brian2 unit of quantity
get_brian2_base_unit(x)	get brian2 base unit of quantity - basic unscaled unit
split_into_temporal_components(t, full=False)	split into significant temporal components (significant up until the largest non-zero component)
format_duration_ns(t, drop_zeros=True)	create string representation of time duration in y, d, h, m, s, ms, mu_s, ns (y:years ~ 365 days)
unique_idx(x)	compute unique values and all indices for each unique value (efficient)
values_in_interval(t0, t1, dt)	compute the number of values in the interval [t0,t1)
next_power_of_two(x)	next power of two implemented using bit length of integer
round_to_res(x, res)	round to a resolution of the most significant digit of parameter res
compute_time_interval(t, dt, t_start = None, t_end = None)	compute a time interval [t_start, t_end] (closed bounds)
restrict_to_interval(x, dt, t_start = None, t_end = None)	restrict data to interval [t_start, t_end] given data is sampled at equidistant intervals of dt
logical_and(*args)	

class utils.**TestEnv**(*path='.tmp'*)

Implements Context Manager Interface to setup and teardown a test environment for file i/o

__enter__(*self*)

makes the tmp_dir and makes it the cwd

__exit__(*self*, *exc_type*, *exc_value*, *traceback*)
makes the initial_dir the cwd and deletes the tmp_dir

utils.**run_cmd**(*cmd*)

utils.**validate_file_path**(*path*, *ext=''*)
Validate file path - whether base path exists, file name has correct extension [verified only in case ext passed], enforces naming conventions on basename only containing characters [**a-zA-Z0-9_**-] yet may start with '.' (hidden files) and has a maximal length of 255 (https://www.ibm.com/docs/en/aix/7.1?topic=files-file-naming-conventions)

Parameters

- **path** (*str*) – path whose validity is tb verified
- **ext** (*str*) – file extension - validity of the extension not verified

Returns error message - empty if path valid

utils.**generate_sequential_file_name**(*base_path*, *base_name*, *ext*)
Generate the next file name sequentially given a directory name and base file name If directory does not exist the directory is created.

Parameters

- **base_path** (*str*) – directory where the files are tb created
- **base_name** (*str*) – basis of the file name used in sequential generation, file name is base_name + '_' + current increment
- **ext** (*str*) –

utils.**retrieve_callers_frame**(*condition*)
retrieve the frame satisfying a condition - if no such frame raises Exception

Parameters **condition** (*Callable[[inspect.FrameInfo], bool]*) – condition to test for the frame in question

Returns first frame in stack passing the condition

utils.**retrieve_callers_context**(*frame_info*)
retrieve the context for a frame - context: globals updated with locals

Parameters **frame_info** (*inspect.FrameInfo*) – the information object of a frame for which context is tb retrieved

Returns context of the respective frame of the encapsulating information object

utils.**clean_brian2_quantity**(*x*)
clean quantity of its unit

Parameters **x** (*brian2.units.fundamentalunits.Quantity*) – quantity cleaned of its unit

Returns cleaned quantity with unit scaling, and string representation of the unit

Return type Tuple[numpy.ndarray, str]

utils.**convert_and_clean_brian2_quantity**(*x*)
convert quantity to base unit and clean of its base unit

Parameters **x** (*brian2.units.fundamentalunits.Quantity*) – quantity which is tb converted to base unit and then cleaned of its unit

Returns cleaned quantity with base unit scaling, and string representation of the base unit

Return type Tuple[numpy.ndarray, str]

utils.**unwrap_brian2_variable_view**(*x*)
unwrap instance of brian2.core.variables.VariableView

Parameters **x** (*brian2.core.variables.VariableView*) – instance of brian2.core.variables.VariableView tb unwrapped

Returns variable value

Return type Union[numpy.ndarray, brian2.units.fundamentalunits.Quantity]

utils.**get_brian2_unit**(*x*)
get brian2 unit of quantity

Parameters **x** (*brian2.units.fundamentalunits.Quantity*) – quantity for which unit is tb determined

Returns unit of quantity parameter x

Return type brian2.units.fundamentalunits.Unit

utils.**get_brian2_base_unit**(*x*)
get brian2 base unit of quantity - basic unscaled unit

Parameters **x** (*brian2.units.fundamentalunits.Quantity*) – quantity for which base unit is tb determined

Returns base unit of quantity parameter x

Return type brian2.units.fundamentalunits.Unit

exception utils.**Brian2UnitError**
Bases: Exception

when instance of Brian2.Unit does not match the expected unit

utils.**split_into_temporal_components**(*t*, *full=False*)
split into significant temporal components (significant up until the largest non-zero component) :param full: if set returns all temporal components :return: values and labels of temporal components [ns, mu_s, ms, s, m, h, d, y]

Parameters **t** (*int*) –

utils.**format_duration_ns**(*t*, *drop_zeros=True*)
create string representation of time duration in y, d, h, m, s, ms, mu_s, ns (y:years ~ 365 days)

Parameters **t** (*int*) – time elapsed in nanoseconds (eg. as a difference of time points or since epoch see time.time_ns())

Returns string representation of time with format: y, d, h, m, s, ms, mu_s, ns - where only duration components whose first increment is reached are used

utils.**unique_idx**(*x*)
compute unique values and all indices for each unique value (efficient)

Parameters **x** (*numpy.ndarray*) – 1D array-like object that holds all values

Returns unique values and indices for each unique value, where the ith set of indices contains all indices of the ith value

Return type Tuple[numpy.ndarray, List[numpy.ndarray]]

utils.**values_in_interval**(*t0*, *t1*, *dt*)
compute the number of values in the interval [t0,t1)

Parameters

- **t0** (*float*) – start of interval (incl.)
- **t1** (*float*) – end of interval (excl.)
- **dt** (*float*) – step size of a step

Returns number of values (= number of steps + 1) in interval from t1 to t0 given step size dt

utils.**next_power_of_two**(*x*)

next power of two implemented using bit length of integer (equivalent to ceil(log2(x)), ie smallest sp such that x <= 2 ** sp)

Parameters **x** (*int*) – value for which the next largest power of 2 is tb determined

Returns smallest power of two greater equal to x (smallest sp such that x <= 2 ** sp)

utils.**round_to_res**(*x*, *res*)

round to a resolution of the most significant digit of parameter res (rounded to the number of decimals at which res has the first nz value, eg. 0.001 -> 3 decimals)

Parameters

- **x** (*float*) – number tb rounded
- **res** (*float*) – most significant bit of this resolution specifies the resolution of rounding

Returns number rounded to a resolution of the most significant digit of param res

utils.**compute_time_interval**(*t*, *dt*, *t_start=None*, *t_end=None*)

compute a time interval [t_start, t_end] (closed bounds) given the optional specifications (t_start, t_end) and verify its validity

Parameters

- **t** (*float*) – simulation time [ms]
- **dt** (*float*) – simulation time step [ms]
- **t_start** (*float*) – time lower bound
- **t_end** (*float*) – time upper bound

Returns bounds of the interval, last time point

utils.**restrict_to_interval**(*x*, *dt*, *t_start=None*, *t_end=None*)

restrict data to interval [t_start, t_end] given data is sampled at equidistant intervals of dt

Parameters

- **x** (*numpy.ndarray*) – data tb restricted to interval
- **dt** (*float*) – time step at which data is sampled
- **t_start** (*float*) – incl. lower bound (time [ms]) for the restriction interval
- **t_end** (*float*) – incl. upper bound (time [ms]) for the restriction interval - if t_end > simulation time ~ t_end=None

Returns data in interval [t_start, t_end], t_start, t_end

Return type Tuple[numpy.ndarray, float, float]

utils.**logical_and**(**args*)

1.10 connectivity

1.10.1 Module Contents

Functions

all2all(source, dest)	
bernoulli(source, dest, p)	

connectivity.**all2all**(*source*, *dest*)

Parameters

- **source** (*Iterable[int]*) –
- **dest** (*Iterable[int]*) –

Return type Tuple[numpy.ndarray, numpy.ndarray]

connectivity.**bernoulli**(*source*, *dest*, *p*)

Parameters

- **source** (*Iterable[int]*) –
- **dest** (*Iterable[int]*) –
- **p** (*float*) –

Return type Tuple[numpy.ndarray, numpy.ndarray]

1.11 network

1.11.1 Module Contents

Classes

SpikeDeviceGroup	Defines Interface for spiking devices and interfaces with brian2.SpikeMonitor and brian2.PopulationRateMonitor
PoissonDeviceGroup	Convenience class for interfacing with the brian2.PoissonGroup of the poisson devices in the population
PoissonBlockedStimulus	Convenience class for interfacing with the brian2.PoissonGroup of the poisson devices in the population
NeuronPopulation	Convenience class for interfacing with the brian2.NeuronGroup and the respective brian2.StatusMonitor of the neurons in the population

continues on next page

Table 17 – continued from previous page

Synapse	Convenience class for interfacing with the created instance of `brian2.synapses.synapses.Synapses` and the respective `brian2.StatusMonitor` of the Synapse
Connector	Convenience class for creating synaptic connections - wrapping the instantation and initialization of instances of `brian2.synapses.synapses.Synapses`

class network.**SpikeDeviceGroup**

Defines Interface for spiking devices and interfaces with `brian2.SpikeMonitor` and `brian2.PopulationRateMonitor` to provide monitoring of spike trains and population rates.

property _pop(*self*)

property ids(*self*)

property monitored(*self*)

Returns dictionary of recorded variables by `brian2.SpikeMonitor` and `brian2.PopulationRateMonitor`

monitor_spike(*self*, *ids*, *variables=[]*)

Register neuron ids for monitoring of spikes and related variables of neurons

Parameters

- **ids** (*List[int]*) – list of neuron ids that are to be monitored on spike for each neuron
- **variables** (*List[str]*) – list of neuron variables that are to be monitored additionally to the neuron id for each spike

monitor_rate(*self*, ***kwargs*)

Register neuron population for rate monitoring

class network.**PoissonDeviceGroup**(*size*, *rate*)

Bases: *SpikeDeviceGroup*

Convenience class for interfacing with the `brian2.PoissonGroup` of the poisson devices in the population

Parameters

- **size** (*int*) –
- **rate** (*Union[brian2.units.fundamentalunits.Quantity, Callable, str]*) –

property _pop(*self*)

property ids(*self*)

ids are unique to a device group and chosen tb equal to the index of a device within the device group - therefore ids start at 0 and are contiguous (an index is valid for a given group if index in [0, group.size - 1]) :return: poisson device ids of the instance of *PoissonDeviceGroup* unique to the instance only! (same as `brian2.PoissonGroup`)

property monitored(*self*)

Returns dictionary of recorded variables by `brian2.SpikeMonitor` and `brian2.PopulationRateMonitor`

static create_time_variant_rate(*offset=1.0*, *amplitude=1.0*, *angularfrequency=2 * np.pi*, *timeshift=0.0*)

Create a time variant rate to pass to PoissonDeviceGroup.__init__() to create inhomogeneous poisson processes rate: [ms]->[kHz]: t -> (offset + cos((t - timeshift[ms]) * angularfrequency[Hz]) * amplitude) * kHz

Parameters

- **offset** (*float*) – offset of the rate function
- **amplitude** (*float*) – scaling factor for amplitude of the rate function
- **angularfrequency** (*float*) – angular frequency of the rate function [Hz]
- **timeshift** (*float*) – time shift of the rate function [ms]

Returns expression representing the time variant rate function, which specifies the rate in kHz per definition

class network.**PoissonBlockedStimulus**(*size*, *pattern*, *block_interval*, *one_rate*, *zero_rate*, *t*, *stimulus_dt*)

Bases: *PoissonDeviceGroup*

Convenience class for interfacing with the brian2.PoissonGroup of the poisson devices in the population

Parameters

- **size** (*int*) –
- **pattern** (*numpy.ndarray*) –
- **block_interval** (*List[Tuple[int, int]]*) –
- **one_rate** (*Union[brian2.units.fundamentalunits.Quantity, str]*) –
- **zero_rate** (*Union[brian2.units.fundamentalunits.Quantity, str]*) –
- **t** (*brian2.units.fundamentalunits.Quantity*) –
- **stimulus_dt** (*brian2.units.fundamentalunits.Quantity*) –

static create_blocked_rate(*size*, *pattern*, *block_interval*, *one_rate*, *zero_rate*, *t*, *stimulus_dt*)

generate an array of rates across devices numbering 'size' and time blocks of length stimulus_dt - rates of individual devices are set according to pattern (mask of one devices) and one_rate (rate of one devices) and zero_rate (rate of zero devices) across time blocks in the interval block_interval ([start,end)); all rates in time blocks not in the interval block_interval are 0.0

Parameters

- **size** (*int*) – size of the group ~ number of spike devices
- **pattern** (*numpy.ndarray*) – pattern (mask) across all spike devices in the group (shape: (size,)) - used for setting rate for all indices in block_interval
- **block_interval** (*List[Tuple[int, int]]*) – set of half-open time intervals ([start,end)) of (time) indices of blocked array tb set to rate
- **t** (*brian2.units.fundamentalunits.Quantity*) – simulation time [ms]
- **stimulus_dt** (*brian2.units.fundamentalunits.Quantity*) – time step of the stimulus ~ size of one block for which rate is held constant in interval t e [t' * stimulus_dt, (t'+1) * stimulus_dt]: stimulus_block[t]
- **one_rate** (*Union[brian2.units.fundamentalunits.Quantity, str]*) –
- **zero_rate** (*Union[brian2.units.fundamentalunits.Quantity, str]*) –

Returns rates for individual devices across time blocks of size stimulus_dt

property monitored(*self*)

Returns dictionary of recorded variables by `brian2.SpikeMonitor` and `brian2.PopulationRateMonitor`

static create_blocked_interval(*offset*, *stim_dur*, *stim_relax*, *sim_t*)

Parameters

- **offset** (*brian2.units.fundamentalunits.Quantity*) – time offset [ms] - offset where no stimulus is presented
- **stim_dur** (*brian2.units.fundamentalunits.Quantity*) – stimulus duration [ms] - duration of an instance of stimulus presentation
- **stim_relax** (*brian2.units.fundamentalunits.Quantity*) – stimulus relaxation [ms] - relaxation period between stimulus presentations
- **sim_t** (*brian2.units.fundamentalunits.Quantity*) – duration of the simulation [ms]

Returns set of block intervals and stimulation time step

Return type Tuple[List[Tuple[int, int]], brian2.units.fundamentalunits.Quantity]

class network.**NeuronPopulation**(*size*, *eqs*, **args*, ***kwargs*)

Bases: *SpikeDeviceGroup*

Convenience class for interfacing with the `brian2.NeuronGroup` and the respective `brian2.StatusMonitor` of the neurons in the population

Example Instantiation of Neuron Population and Initialization of Membrane Potential

(note if variable is of same dimension as the neuron population use **:prop:`NeuronPopulation.size`** instead of `NeuronPopulation.get_var_size()`)

```
with BrianExperiment():
    N = NeuronPopulation(1000,'dv/dt = (1-v)/tau : volt')
    mu = 0.
    sigma = 1.
    N.set_pop_var("v", draw_normal(mu=mu, sigma=sigma, size=N.get_pop_var_size("v
↪")) * mV)
    # N.set_pop_var("v", draw_normal(mu=mu, sigma=sigma, size=N.size) * mV)
    vals = N.get_pop_var("v") / mV
    mean = np.mean(vals)
    std = np.std(vals)
    print(f"mu:    is w/in 0.1 tolerance ({abs(mean - mu) / sigma < 0.1})")
    print(f"sigma: is w/in 0.1 tolerance ({abs(std - sigma) / sigma < 0.1})")
```

```
mu:    is w/in 0.1 tolerance (True)
sigma: is w/in 0.1 tolerance (True)
```

Parameters

- **size** (*int*) –
- **eqs** (*str*) –

set_pop_var(*self*, *variable*, *value*)
set population variable - variables defined in eqs param of `NeuronPopulation.__init__()`

Parameters

- **variable** (*str*) – name of variable used in eqs
- **value** (*Union[brian2.units.fundamentalunits.Quantity, numpy.ndarray]*) – value tb assigned to param variable

get_pop_var(*self*, *variable*)
get population variable - variables defined in eqs param of `NeuronPopulation.__init__()`

Parameters **variable** (*str*) – name of variable used in eqs

Returns value bound to param variable

Return type brian2.units.fundamentalunits.Quantity

get_pop_var_size(*self*, *variable*)
get size of a population variable - variables defined in eqs param of `NeuronPopulation.__init__()`

Parameters **variable** (*str*) – name of variable used in eqs

Returns size of value bound to param variable

Return type int

property _pop(*self*)

property size(*self*)

Returns size of the instance of *NeuronPopulation*

property ids(*self*)
ids are unique to a neuron population and chosen tb equal to the index of a neuron within the population - therefore ids start at 0 and are contiguous (an index is valid for a given population if index in [0, pop.size - 1]) :return: neuron ids of the instance of *NeuronPopulation* unique to the instance only! (same as `brian2.NeuronGroup`)

property monitored(*self*)

Returns dictionary of recorded variables and their recorded values

__len__(*self*)

monitor(*self*, *ids*, *variables=[]*, *dt=None*)
Register neuron ids for monitoring of states neuron variables

Parameters

- **ids** (*List[int]*) – list of neuron ids whose states are to be monitored for each neuron
- **variables** (*List[str]*) – list of variables that are to be monitored for each of the neurons
- **dt** (*float*) – time step to be used for monitoring - df: time step specified in `BrianExperiment.__init__()` of enclosing instance of *BrianExperiment* used

class network.**Synapse**(*synapse_object*, *synapse_params={}*, *on_pre=None*)
Convenience class for interfacing with the created instance of `brian2.synapses.synapses.Synapses` and the respective `brian2.StatusMonitor` of the Synapse An instance of this class *Synapse* is returned by *Connector.__call__()*

multi-synapses (>1 synpase btw same source and dest) not supported (see member multisynaptic_index of brian2.synapses.synapses.Synapses)

Parameters

- **synapse_object** (*brian2.synapses.synapses.Synapses*) –
- **synapse_params** (*Dict[str, numpy.ndarray]*) –

property synapse_params(*self*)

Returns synapse parameters set on the underlying synapse object

property synapses(*self*)

Returns synapses in terms of tuple of pre- and postsynaptic neuron id (internally resolved to synapse id in brian2.synapses.synapses.Synapse only unique to the synapse instance)

property monitored(*self*)

Returns dictionary of recorded variables and their recorded values

__len__(*self*)

monitor(*self*, *synapses*, *variables*, *dt=None*)
Register synapses for monitoring

Parameters

- **synapses** (*Union[numpy.ndarray, List[Tuple[int, int]]]*) – group of synapses defined as a tuple of the pre- and postsynaptic neuron ids that are to be monitored - note that if this instance manages a large number of synapses and the number of elements provided is of the same order mapping to brian2 indices will be prohibitively expensive unless param synapses == Synapse.synapses (property)
- **variables** (*List[str]*) – list of variables that are to be monitored for each of the synapses
- **dt** (*brian2.units.fundamentalunits.Quantity*) – time step to be used for monitoring - df: time step specified in BrianExperiment.__init__() of enclosing instance of *BrianExperiment* used

class network.**Connector**(*synapse_type='static'*)
Convenience class for creating synaptic connections - wrapping the instantation and initialization of instances of brian2.synapses.synapses.Synapses

Parameters **synapse_type** (*str*) –

__call__(*self*, *sourcePop*, *destPop*, *sourceIds*, *destIds*, *connect*, *syn_params={}*, *model=''*, *on_pre=None*, ***kwargs*)
Creates synaptic connections between two instances of *NeuronPopulation* of synapse type specified in Synapse.__init__()

Parameters

- **sourcePop** (SpikeDeviceGroup) – instance subclassed from *SpikeDeviceGroup* that contains the subset of presynaptic neurons referenced by ids in parameter sourceIds
- **destPop** (SpikeDeviceGroup) – instance subclassed from *SpikeDeviceGroup* that contains the subset of postsynaptic neurons referenced by ids in parameter destIds

- **sourceIds** (*List[int]*) – subset of neuron ids for presynaptic neurons for which synapses are tb created
- **destIds** (*List[int]*) – subset of neuron ids for postsynaptic neurons for which synapses are tb created
- **connect** (*Union[Callable[[List[int], List[int]], List[Tuple[int, int]]], Tuple[str, Dict[str, Union[int, float]]]]*) – Callable or tuple of specifier ct and params that specify the topology between the two instances of *NeuronPopulation* options for topologies in ct: 'all2all' | 'one2one' | 'bernoulli', note that bernoulli requires param 'p'
- **syn_params** (*Dict[str, brian2.units.fundamentalunits.Quantity]*) – parameters tb set for the synaptic model on a per synapse basis
- **model** (*str*) –
- **on_pre** (*str*) –

Returns instance of *Synapse* which allows interacting with the synapses created

Return type *Synapse*

1.12 ExperimentAnalysis

1.12.1 Module Contents

Classes

ExperimentAnalysis	Analyse data from *BrianExperiment.BrianExperiment*

class ExperimentAnalysis.**ExperimentAnalysis**(*experiment_data*, *t_start=10.0*, *t_end=None*)

Analyse data from *BrianExperiment.BrianExperiment*

Example for analyzing data by all analysis methods

```
with TestEnv():
    for run in range(2):
        with BrianExperiment(persist=True, path="file.h5", object_path=f"/run_{run}/
↪data") as exp:
            exp.persist_data["mon"] = np.arange(10)
    with FileMap("file_analysis.h5") as af:
        with FileMap("file.h5") as f:
            for run in f.keys():
                exp_analysis = ExperimentAnalysis(experiment_data=f[run]["data"])
                exp_analysis.analyze_all()
                af[run] = exp_analysis.report()

        print({run:list(af[run].keys()) for run in af.keys()})
```

```
{ "run_1" : ["x", "y", "z"], "run_2" : ["x", "y", "z"] }
```

Example for analyzing data by specific analysis methods

```
with TestEnv():
    for run in range(2):
        with BrianExperiment(persist=True, path="file.h5", object_path=f"/run_{run}/
↪data") as exp:
            exp.persist_data["mon"] = np.arange(10)
    with FileMap("file_analysis.h5") as af:
        with FileMap("file.h5") as f:
            for run in f.keys():
                exp_analysis = ExperimentAnalysis(experiment_data=f[run]["data"])
                exp_analysis.analyze_instantaneous_rate()
                af[run] = exp_analysis.report()

        print({run:list(af[run].keys()) for run in af.keys()})
```

```
{ "run_1" : ["instantaneous_rate"], "run_2" : ["instantaneous_rate"] }
```

Parameters

- **experiment_data** (*Union[Dict, persistence.Reader]*) –
- **t_start** (*float*) –
- **t_end** (*float*) –

property report(*self*)

analyze_all(*self*)

analyze_instantaneous_rate(*self*, *pop_name=None*)
compute the instantaneous rate for populations

Parameters **pop_name** (*List[str]*) – populations for which the instantaneous rate is computed
- df: None ~ all populations

analyze_smoothed_rate(*self*, *mode='gaussian'*, *window_size=1.0*)
compute the smoothed rate for populations analyzed in *ExperimentAnalysis.analyze_instantaneous_rate()*

Parameters

- **mode** (*str*) – mode used for smoothing the instantaneous rate
- **window_size** (*float*) – window size tb used for smoothing in [ms]

analyze_power_spectral_density(*self*, *pop_name=None*, *separate_intervals=False*, *f_lower_bound=50.0*, *f_upper_bound=300.0*)
compute power spectral density - computes psd over the entire signal if separate_intervals set also computes psd in separate time intervals

Parameters

- **pop_name** (*List[str]*) – list of populations to compute psd for - defaults to all NeuronPopulations
- **separate_intervals** (*bool*) – also compute psd for separate time intervals (using sliding window)
- **f_lower_bound** (*float*) – does not consider frequencies (and resp. psd) below lower bound

- **f_upper_bound** (*float*) – does not consider frequencies (and resp. psd) above upper bound

analyze_peaks(*self*, *pop_name=None*, *smoothed=True*)
analyze the peaks (& troughs) of the population rate of neuron populations with a minimum distance between peaks of half the wavelength of the fundamental frequency of the population rate

Parameters

- **pop_name** (*List[str]*) – population names for which peaks are tb detected
- **smoothed** (*bool*) – whether to analyze the smoothed or instantaneous population rate

analyze_cell_rate(*self*, *pop_name=None*)
cell rate per cell, the population average and the population maximum for populations

Parameters **pop_name** (*List[str]*) – populations for which the instantaneous rate is computed - df: None ~ all populations

analyze_snr(*self*, *bin_size=10.0*)
signal-to-noise ratio for populations analyzed in *ExperimentAnalysis.analyze_power_spectral_density()*

analyze_synchronization_frequency(*self*)
synchronization_frequency for populations analyzed in *ExperimentAnalysis.analyze_power_spectral_density()*

analyze_avg_power(*self*)
average power for populations analyzed in *ExperimentAnalysis.analyze_power_spectral_density()*

analyze_total_synaptic_conductance(*self*, *pop_e*, *pop_i*, *synaptic_input_e_e_name='x_AMPA'*, *synaptic_input_i_e_name='x_GABA'*, *conductance_e_e_name='gsynE_E'*, *conductance_i_e_name='gsynI_E'*)
total synaptic conductance for populations pop_e assumes synaptic connectivity to pop_e is limited to e-e: pop_e -> pop_e and i-e: pop_i -> pop_e

Parameters

- **pop_e** (*str*) – excitatory population for which total synaptic conductance is computed
- **pop_i** (*str*) – inhibitory population which connects to pop_e
- **synaptic_input_e_e_name** (*str*) – name of synaptic input variable that is modified on presynaptic spike for e-e synapses
- **synaptic_input_i_e_name** (*str*) – name of synaptic input variable that is modified on presynaptic spike for i-e synapses
- **conductance_e_e_name** (*str*) – name of conductance for synaptic inputs of e-e synapses
- **conductance_i_e_name** (*str*) – name of conductance for synaptic inputs of i-e synapses

Returns total conductance and respective ids of pop_e

analyze_snapshots(*self*, *pop_name*)
extract snapshots from population rate and spike trains

Parameters **pop_name** (*str*) – population name for which to extract snapshots

analyze_similarity_distribution(*self*, *pop_name*)
compute similarity distribution across snapshots per pattern (distribution: (N x C), (i,j) ~ similarity of snapshot from cycle j with pattern i)

assumption: patterns used for pop_name are assigned to the BrianExperiment.persist_data

with BrianExperiment(...) as exp: exp.persist_data[pop_name] = {"pattern":np.ndarray, "sparsity":float}

Parameters **pop_name** (*str*) – population name for which to compute the similarity threshold

1.13 plot

1.13.1 Module Contents

Classes

ExperimentPlotter	Plot data from *BrianExperiment.BrianExperiment* and analysis data from *ExperimentAnalysis.ExperimentAnalysis*

Functions

discrete_palette(n, color='husl')	discrete cirular color palette
sequential_palette(base_color, n)	create a generator over sequential colors of a primary hue, base_color
color_palette(cmap, n)	generator over the color map of length n
subdivide_subplot(fig, ax, rows, cols)	subdivide axes / subplot into a grid of axes
psth_snap(fig, ax, snapshots, snap_times, stimulus_pattern, num_presentations, inter_presentation_interval, stimulus_length, num_bins = 10)	Peristimulus Time Histogram for snapshot activity over stimulus presentation cycles
psth(fig, ax, spike_train_times, spike_train_ids, stimulus_pattern, num_presentations, inter_presentation_interval, stimulus_length, num_bins = 10)	Peristimulus Time Histogram for snapshot activity over stimulus presentation cycles
plot_spike_train_interval(fig, ax, spike_train_ids, spike_train_times, lbound, ubound, pop_rate, t_start, t_end, dt)	plot spikes of a specific time interval [lbound, ubound]
plot_spike_train_presentation_cycle(fig, ax, spike_train_ids, spike_train_times, stimulus_onset, inter_presentation_interval, stimulus_length, pop_rate, t_start, t_end, dt, stimulus_pattern = None)	plot spikes of a specific presentation cycle [stimulus_onset - inter_presentation_interval, stimulus_onset + inter_presentation_interval]
plot_similarity_top_snaps(fig, ax, pop_name, title, rows, cols, similarity_threshold, similarity_distribution)	plot top snapshots for which the threshold is exceeded for each pattern
plot_similarity_distributions(fig, ax, pop_name, title, rows, cols, similarity_threshold, similarity_distribution)	plot similarity distributions with a single (same) threshold
plot_similarity_distributions_individual(fig, ax, pop_name, title, rows, cols, similarity_threshold, similarity_distribution)	plot similarity distributions with individual thresholds

continues on next page

Table 20 – continued from previous page

plot_similarity_per_snapshot_over_time(fig, ax, pop_name, title, pvalues, troughs, stimulus_block_interval, t_start, t_end, dt, significance = 0.05)	plot similarity for the respective reference pattern over time
plot_similarity_blocked_stimulus(fig, ax, pop_name, title, pvalues, troughs, stimulus_onset, stimulus_length, sign_snaps_pre, sign_snaps_stim, sign_snaps_post, sign_snaps_pre_far = None, sign_snaps_pre_close = None, sign_snaps_post_close = None, sign_snaps_post_far = None, significance = 0.05)	plot similarity for the respective reference pattern over time
plot_similarity_blocked_stimulus_sliding_window(fig, ax, pop_name, title, pvalues, troughs, stimulus_onset, inter_onset_interval, stimulus_length, window_length, window_step, sign_snaps, mean_pvals, significance = 0.05)	plot similarity for the respective reference pattern over time
plot_similarity_blocked_stimulus_sliding_window_single(fig, ax, stimulus_onset, inter_onset_interval, stimulus_length, window_step, sign_snaps = None, mean_pvals = None, mean_similarity = None, sign_snaps_unweighted = None, mean_pvals_unweighted = None, mean_similarity_unweighted = None, zoomed = False, excerpt_size = 100.0)	plot significant any combination of snapshots, mean pvalues and mean similarity for the respective reference pattern using a sliding window
plot_similarity_per_pattern(fig, ax, pop_name, title, rows, cols, pvalues, significance = 0.05)	plot similarity per pattern
plot_pairwise_similarity(fig, ax, pop_name, title, rows, cols, pvalues, significance = 0.05)	plot pairwise similarity
plot_total_synaptic_conductance(fig, ax, pop_name, title, total_synaptic_conductance)	plot total synaptic conductance
plot_spike_train(fig, ax, pop_name, train, size, color = {}, title = '')	id to spike train plot for neuron populations
plot_variable(fig, ax, times, pop_name, ids, variable, color, xlabel, ylabel, title = '')	plot of voltages of excitatory and inhibitory population
plot_instantaneous_rate(fig, ax, times, instantaneous_rate, color, title = '')	plot instantaneous rate of populations
plot_smoothed_rate(fig, ax, times, smoothed_rate, color, title = '', sync_freq = {}, snr = {})	plot smooth rate of populations
plot_cell_rate(fig, ax, pop_name, cell_rate, unit, ids, color, title = '')	plot smooth rate of populations
plot_multitaper_spectrum(fig, ax, frequency, psd, color = None, pop_name = None, title = '')	Plot multitaper power spectral density of a population rate for an entire time series
plot_multitaper_spectogram(fig, ax, frequency, psd, t_start, t_end, pop_name = None, title = '')	Plot multitaper spectogram of a population rate developing over time (separate psds computed with a sliding window)
plot_synchronization_regimes()	Synchronization features across the regimes Ing, Bifurcation, Ping

Attributes

color_its

colors

plot.**discrete_palette**(*n*, *color='husl'*)
discrete cirular color palette

Parameters

- **n** – number of colors tb generated
- **color** – color tb used for :func:'sns.color_palette'

plot.**sequential_palette**(*base_color*, *n*)
create a generator over sequential colors of a primary hue, base_color

Parameters

- **base_color** (*str*) – base color for colormap
- **n** (*int*) – length of the iterator

plot.**color_palette**(*cmap*, *n*)
generator over the color map of length n

Parameters

- **cmap** (*matplotlib.colors.LinearSegmentedColormap*) – linear segmented colormap
- **n** (*int*) – length of the iterator

plot.**color_its**

plot.**colors**

plot.**subdivide_subplot**(*fig*, *ax*, *rows*, *cols*)
subdivide axes / subplot into a grid of axes

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure to which parameter ax belongs
- **ax** (*matplotlib.pyplot.Axes.axes*) – axes / subplot which is tb subdivided
- **rows** (*int*) – number of rows of the grid into which the parameter ax is subdivided
- **cols** (*int*) – number of cols of the grid into which the parameter ax is subdivided

plot.**psth_snap**(*fig*, *ax*, *snapshots*, *snap_times*, *stimulus_pattern*, *num_presentations*, *inter_presentation_interval*, *stimulus_length*, *num_bins=10*)
Peristimulus Time Histogram for snapshot activity over stimulus presentation cycles

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **snapshots** (*numpy.ndarray*) – snapshots (C, pattern_length), where C is number snapshots
- **snap_times** (*numpy.ndarray*) – snapshot firing times resolved relative to stimulus [ms]

- **num_presentations** (*int*) – number of stimulus presentations
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets or ceasures equivalently [ms]
- **stimulus_length** (*float*) – length of stimulus presentation [ms]
- **num_bins** (*int*) – number of bins
- **stimulus_pattern** (*numpy.ndarray*) –

plot.**psth**(*fig*, *ax*, *spike_train_times*, *spike_train_ids*, *stimulus_pattern*, *num_presentations*, *inter_presentation_interval*, *stimulus_length*, *num_bins=10*)

Peristimulus Time Histogram for snapshot activity over stimulus presentation cycles [stimulus_onset - inter_presentation_interval + stimulus_length, stimulus_onset + inter_presentation_interval]

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **spike_times** – timings of spikes [ms] (S,), where S is number spike events
- **spike_ids** – ids of neuron for which a spike event is evoked (S,), where S is number spike events
- **num_presentations** (*int*) – number of stimulus presentations
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets or ceasures equivalently [ms]
- **stimulus_length** (*float*) – length of stimulus presentation [ms]
- **num_bins** (*int*) – number of bins
- **spike_train_times** (*numpy.ndarray*) –
- **spike_train_ids** (*numpy.ndarray*) –
- **stimulus_pattern** (*numpy.ndarray*) –

plot.**plot_spike_train_interval**(*fig*, *ax*, *spike_train_ids*, *spike_train_times*, *lbound*, *ubound*, *pop_rate*, *t_start*, *t_end*, *dt*)

plot spikes of a specific time interval [lbound, ubound]

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **spike_train_ids** (*numpy.ndarray*) – neuron ids of spikes
- **spike_train_times** (*numpy.ndarray*) – spike times of spikes [ms]
- **lbound** (*float*) – lower bound of time interval [ms]
- **ubound** (*float*) – upper bound of time interval [ms]
- **pop_rate** (*numpy.ndarray*) – population rate
- **t_start** (*float*) – start of simulation/analysis
- **t_end** (*float*) – end of simulation/analysis
- **dt** (*float*) – time step of simulation

plot.**plot_spike_train_presentation_cycle**(*fig*, *ax*, *spike_train_ids*, *spike_train_times*, *stimulus_onset*, *inter_presentation_interval*, *stimulus_length*, *pop_rate*, *t_start*, *t_end*, *dt*, *stimulus_pattern=None*)

plot spikes of a specific presentation cycle [stimulus_onset - inter_presentation_interval, stimulus_onset + inter_presentation_interval]

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **spike_train_ids** (*numpy.ndarray*) – neuron ids of spikes
- **spike_train_times** (*numpy.ndarray*) – spike times of spikes [ms]
- **stimulus_pattern** (*numpy.ndarray*) – one of the patterns embedded in the weight matrix and whose perturbation is used as a stimulus (pop_length,)
- **stimulus_onset** (*float*) – onset of stimulus presentation [ms]
- **inter_presentation_interval** (*float*) – interval between two stimulus presentation onsets [ms]
- **stimulus_length** (*float*) – length of stimulus [ms]
- **pop_rate** (*numpy.ndarray*) – population rate
- **t_start** (*float*) – start time of pop_rate
- **t_end** (*float*) – end time of pop_rate simulation/analysis
- **dt** (*float*) – resolution of pop_rate

plot.**plot_similarity_top_snaps**(*fig*, *ax*, *pop_name*, *title*, *rows*, *cols*, *similarity_threshold*, *similarity_distribution*)

plot top snapshots for which the threshold is exceeded for each pattern

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **rows** (*int*) – number of rows into which the axes/subplot is subdivided
- **cols** (*int*) – number of columns into which the axes/subplot is subdivided
- **similarity_threshold** (*int*) – similarity threshold corresponding to the tightest lower bound on the significance level for all patterns - assumes patterns have fixed (and equal) # 1s
- **similarity_distribution** (*numpy.ndarray*) – similarities between patterns and snapshots (N x C) where N is the number of patterns and C the number of cycles
- **pop_name** (*str*) –
- **title** (*str*) –

plot.**plot_similarity_distributions**(*fig*, *ax*, *pop_name*, *title*, *rows*, *cols*, *similarity_threshold*, *similarity_distribution*)

plot similarity distributions with a single (same) threshold

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance

- **rows** (*int*) – number of rows into which the axes/subplot is subdivided
- **cols** (*int*) – number of columns into which the axes/subplot is subdivided
- **similarity_threshold** (*int*) – similarity threshold corresponding to the tightest lower bound on the significance level for all patterns - assumes patterns have fixed (and equal) # 1s
- **similarity_distribution** (*numpy.ndarray*) – similarities between patterns and snapshots (N x C) where N is the number of patterns and C the number of cycles
- **pop_name** (*str*) –
- **title** (*str*) –

plot.**plot_similarity_distributions_individual**(*fig*, *ax*, *pop_name*, *title*, *rows*, *cols*, *similarity_threshold*, *similarity_distribution*)

plot similarity distributions with individual thresholds

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **rows** (*int*) – number of rows into which the axes/subplot is subdivided
- **cols** (*int*) – number of columns into which the axes/subplot is subdivided
- **similarity_threshold** (*numpy.ndarray*) – similarity threshold (N x 1) for each pattern corresponding to the tightest lower bound on the significance level
- **similarity_distribution** (*numpy.ndarray*) – similarities between patterns and snapshots (N x C) where N is the number of patterns and C the number of cycles
- **pop_name** (*str*) –
- **title** (*str*) –

plot.**plot_similarity_per_snapshot_over_time**(*fig*, *ax*, *pop_name*, *title*, *pvalues*, *troughs*, *stimulus_block_interval*, *t_start*, *t_end*, *dt*, *significance=0.05*)

plot similarity for the respective reference pattern over time

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **pvalues** (*numpy.ndarray*) – pvalues of similarity of the respective and snapshot (C,), where C is the number of snapshots
- **troughs** (*numpy.ndarray*) – troughs of the population rate (basis for the calculation of snapshots) indices in ['t_start','t_end'] with step 'dt'
- **stimulus_block_interval** (*Union[numpy.ndarray, Dict]*) – set of intervals during which stimulus is present composed of start and end time in [ms] (B x 2), where B is the number of intervals and stimulus_block_interval[b,0], stimulus_block_interval[b,1] is start/end time of interval b, or set of parameters incl a subset of 'offset', 'amplitude', 'angularfrequency' and 'timeshift' characterizing an inhomogeneous poisson process
- **t_start** (*float*) – start of analysis
- **t_end** (*float*) – end of analysis
- **dt** (*float*) – time step of simulation

- **significance** (*float*) – significance level for plotting pvalues
- **pop_name** (*str*) –
- **title** (*str*) –

plot.**plot_similarity_blocked_stimulus**(*fig*, *ax*, *pop_name*, *title*, *pvalues*, *troughs*, *stimulus_onset*, *stimulus_length*, *sign_snaps_pre*, *sign_snaps_stim*, *sign_snaps_post*, *sign_snaps_pre_far=None*, *sign_snaps_pre_close=None*, *sign_snaps_post_close=None*, *sign_snaps_post_far=None*, *significance=0.05*)

plot similarity for the respective reference pattern over time

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **pvalues** (*numpy.ndarray*) – pvalues of similarity of the respective and snapshot (C,), where C is the number of snapshots
- **troughs** (*numpy.ndarray*) – troughs of the population rate (basis for the calculation of snapshots) [ms]
- **stimulus_onset** (*float*) – point of time of onset of stimulus [ms]
- **stimulus_length** (*float*) – length of stimulus [ms]
- **sign_snaps_pre** (*float*) – fraction of significant snaps in window of size parameter 'stimulus_length' before stimulus presentation
- **sign_snaps_stim** (*float*) – fraction of significant snaps in window of size parameter 'stimulus_length' at stimulus presentation
- **sign_snaps_post** (*float*) – fraction of significant snaps in window of size parameter 'stimulus_length' after stimulus presentation
- **significance** (*float*) – significance level for plotting pvalues
- **pop_name** (*str*) –
- **title** (*str*) –
- **sign_snaps_pre_far** (*float*) –
- **sign_snaps_pre_close** (*float*) –
- **sign_snaps_post_close** (*float*) –
- **sign_snaps_post_far** (*float*) –

plot.**plot_similarity_blocked_stimulus_sliding_window**(*fig*, *ax*, *pop_name*, *title*, *pvalues*, *troughs*, *stimulus_onset*, *inter_onset_interval*, *stimulus_length*, *window_length*, *window_step*, *sign_snaps*, *mean_pvals*, *significance=0.05*)

plot similarity for the respective reference pattern over time

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **pvalues** (*numpy.ndarray*) – pvalues of similarity of the respective and snapshot (C,), where C is the number of snapshots

- **troughs** (*numpy.ndarray*) – troughs of the population rate (basis for the calculation of snapshots) [ms]
- **stimulus_onset** (*float*) – point of time of onset of stimulus [ms]
- **inter_onset_interval** (*float*) – interval time between the onsets of any two subsequent stimulus presentations [ms]
- **stimulus_length** (*float*) – length of stimulus [ms]
- **window_length** (*float*) – length of sliding window [ms]
- **window_step** (*float*) – step size of sliding window [ms]
- **sign_snaps** (*numpy.ndarray*) – fraction of significant snapshots at significance level over windows (window order: [so, so + wl], ..., [so + ioi - wl, so + ioi],

  where so is stimulus_onset, wl is window_length, ioi is inter_onset_interval)
- **mean_pvals** (*numpy.ndarray*) – mean of the pvalues over windows (same order as 'sign_snaps')
- **significance** (*float*) – significance level for plotting pvalues
- **pop_name** (*str*) –
- **title** (*str*) –

plot.**plot_similarity_blocked_stimulus_sliding_window_delta**(*fig*, *ax*, *stimulus_onset*, *inter_onset_interval*, *stimulus_length*, *window_step*, *sign_snaps=None*, *mean_pvals=None*, *mean_similarity=None*, *sign_snaps_unweighted=None*, *mean_pvals_unweighted=None*, *mean_similarity_unweighted=None*, *zoomed=False*, *excerpt_size=100.0*)

plot significant any combination of snapshots, mean pvalues and mean similarity for the respective reference pattern using a sliding window for the same experiment configuration simulated with and without (random connectivity) weights and their delta

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **stimulus_onset** (*float*) – point of time of onset of stimulus [ms]
- **inter_onset_interval** (*float*) – interval time between the onsets of any two subsequent stimulus presentations [ms]
- **stimulus_length** (*float*) – length of stimulus [ms]
- **window_step** (*float*) – step size of sliding window [ms]
- **sign_snaps** (*numpy.ndarray*) – fraction of significant snapshots at significance level over window (window order: [so, so + wl], ..., [so + ioi - wl, so + ioi],

  where so is stimulus_onset, wl is window_length, ioi is inter_onset_interval)
- **mean_pvals** (*numpy.ndarray*) – mean of the pvalues over window (same order as 'sign_snaps')

- **mean_similarity** (*numpy.ndarray*) – mean of the similarity over window (same order as 'sign_snaps')
- **sign_snaps_unweighted** (*numpy.ndarray*) – fraction of significant snapshots at significance level over windows for unweighted simulation (control) (window order: [so, so + wl], . . . , [so + ioi - wl, so + ioi],

  where so is stimulus_onset, wl is window_length, ioi is inter_onset_interval) for unweighted simulation (control)
- **mean_pvals_unweighted** (*numpy.ndarray*) – mean of the pvalues over windows (same order as 'sign_snaps')
- **mean_similarity_unweighted** (*numpy.ndarray*) – mean of the similarity over window (same order as 'sign_snaps')
- **zoomed** (*bool*) – whether or not to restrict plot to excerpts around stimulus onset and end
- **excerpt_size** (*float*) – size of the excerpt [ms] around stimulus onset and stimulus end - considered only if zoomed set to True

plot.**plot_similarity_per_pattern**(*fig*, *ax*, *pop_name*, *title*, *rows*, *cols*, *pvalues*, *significance=0.05*)
plot similarity per pattern

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **rows** (*int*) – number of rows into which the axes/subplot is subdivided
- **cols** (*int*) – number of columns into which the axes/subplot is subdivided
- **pvalues** (*numpy.ndarray*) – pvalues of similarity per pattern and snapshot (N x C), where N is number of patterns and C is snapshots
- **significance** (*float*) – significance level for plotting pvalues
- **pop_name** (*str*) –
- **title** (*str*) –

plot.**plot_pairwise_similarity**(*fig*, *ax*, *pop_name*, *title*, *rows*, *cols*, *pvalues*, *significance=0.05*)
plot pairwise similarity :param fig: figure instance :param ax: axis instance :param rows: number of rows into which the axes/subplot is subdivided :param cols: number of columns into which the axes/subplot is subdivided :param pvalues: pvalues of similarity per pattern and snapshot (N x C), where N is number of patterns and C is snapshots :param significance: significance level for plotting pvalues

Parameters

- **fig** (*matplotlib.pyplot.Figure*) –
- **ax** (*matplotlib.pyplot.Axes.axes*) –
- **pop_name** (*str*) –
- **title** (*str*) –
- **rows** (*int*) –
- **cols** (*int*) –
- **pvalues** (*numpy.ndarray*) –
- **significance** (*float*) –

plot.**plot_total_synaptic_conductance**(*fig*, *ax*, *pop_name*, *title*, *total_synaptic_conductance*)
plot total synaptic conductance

Parameters

- **fig** (*matplotlib.pyplot.Figure*) – figure instance
- **ax** (*matplotlib.pyplot.Axes.axes*) – axis instance
- **pop_name** (*str*) – name of population
- **total_synaptic_conductance** (*numpy.ndarray*) – total synaptic conductance per neuron
- **title** (*str*) –

plot.**plot_spike_train**(*fig*, *ax*, *pop_name*, *train*, *size*, *color={}*, *title=''*)
id to spike train plot for neuron populations

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance
- **ax** (*matplotlib.axes.Axes*) – axis instance
- **pop_name** (*List[str]*) – populations tb plotted in order
- **train** (*Dict[str, Tuple[numpy.ndarray, numpy.ndarray]]*) – spike train by neuron population as a Tuple of ids and spikes (where neuron ids[i] spiked at spike time spikes[i])
- **size** (*Dict[str, int]*) – popullation size by neuron population
- **color** (*Dict[str, str]*) – color by neuron population (opt.)
- **title** (*str*) – title of plot

plot.**plot_variable**(*fig*, *ax*, *times*, *pop_name*, *ids*, *variable*, *color*, *xlabel*, *ylabel*, *title=''*)
plot of voltages of excitatory and inhibitory population

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance
- **ax** (*matplotlib.axes.Axes*) – axis instance
- **times** – time points
- **pop_name** (*List[str]*) – names of populations tb plotted in order
- **ids_e** – ids of neurons by population
- **variable** (*Dict[str, numpy.ndarray]*) – variable by neuron id and population
- **color** (*Dict[str, Iterator]*) – iterator over colors by population
- **xlabel** (*str*) – label for the x axis
- **ylabel** (*str*) – label for the y axis
- **title** (*str*) – title of plot
- **ids** (*Dict[str, List[int]]*) –

plot.**plot_instantaneous_rate**(*fig*, *ax*, *times*, *instantaneous_rate*, *color*, *title=''*)
plot instantaneous rate of populations

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance

- **ax** (*matplotlib.axes.Axes*) – axis instance
- **times** (*numpy.ndarray*) – time points
- **instantaneous_rate** (*Dict[str, Tuple[numpy.ndarray, str]]*) – instantaneous population rate over time by population
- **color** (*str*) – color by population
- **title** (*str*) – title of plot

plot.**plot_smoothed_rate**(*fig*, *ax*, *times*, *smoothed_rate*, *color*, *title=''*, *sync_freq={}*, *snr={}*)
plot smooth rate of populations

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance
- **ax** (*matplotlib.axes.Axes*) – axis instance
- **times** (*numpy.ndarray*) – time points
- **smoothed_rate** (*Dict[str, Tuple[numpy.ndarray, str, str]]*) – smoothed population rate over time by population
- **color** (*Dict[str, str]*) – color by population
- **title** (*str*) – title of plot
- **sync_freq** (*Dict[str, float]*) – synchronization frequency of the population rate by population
- **snr** (*Dict[str, float]*) – snr of the population rate by population

plot.**plot_cell_rate**(*fig*, *ax*, *pop_name*, *cell_rate*, *unit*, *ids*, *color*, *title=''*)
plot smooth rate of populations

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance
- **ax** (*matplotlib.axes.Axes*) – axis instance
- **cell_rate** (*numpy.ndarray*) – cell rate and ids per neuron by population
- **unit** (*str*) – unit of cell rate
- **ids** (*numpy.ndarray*) – neuron ids corresponding to cell rates
- **color** (*Dict[str, str]*) – color by population
- **title** (*str*) – title of plot
- **pop_name** (*str*) –

plot.**plot_multitaper_spectrum**(*fig*, *ax*, *frequency*, *psd*, *color=None*, *pop_name=None*, *title=''*)
Plot multitaper power spectral density of a population rate for an entire time series - log power against frequency

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance
- **ax** (*matplotlib.axes.Axes*) – axis instance
- **frequency** – frequencies corresponding to spectral densities in param psd
- **psd** – power spectral density
- **color** (*str*) – color of psd plot

- **pop_name** (*str*) – used for specifying the title (optional)
- **title** (*str*) – title of plot

plot.**plot_multitaper_spectogram**(*fig*, *ax*, *frequency*, *psd*, *t_start*, *t_end*, *pop_name=None*, *title=''*)
Plot multitaper spectogram of a population rate developing over time (separate psds computed with a sliding window) - power against frequency and time

Parameters

- **fig** (*matplotlib.figure.Figure*) – figure instance
- **ax** (*matplotlib.axes.Axes*) – axis instance
- **frequency** (*numpy.ndarray*) – corresponding frequencies to densities in param psd[:,i] for any specific interval i
- **psd** (*numpy.ndarray*) – power spectral density (shape: (nfft/2, intervals) ~ rows -> psd, col -> time) (see also *analysis.multitaper_power_spectral_density()*)
- **t_start** (*float*) – start time of the time series
- **t_end** (*float*) – end time of the time series
- **pop_name** (*str*) – used for specifying the title (optional)
- **title** (*str*) – title of plot

plot.**plot_synchronization_regimes**()
Synchronization features across the regimes Ing, Bifurcation, Ping

class plot.**ExperimentPlotter**(*pop_name_e*, *pop_name_i*, *data=None*, *analysis=None*, *layout='vertical'*, *t_start=10.0*, *t_end=None*, ***kwargs*)
Plot data from *BrianExperiment.BrianExperiment* and analysis data from *ExperimentAnalysis.ExperimentAnalysis*

Example for plotting

```
from BrianExperiment import BrianExperiment
from analysis import ExperimentAnalysis
from plot import ExperimentPlotter
from persistence import FileMap
from utils import TestEnv

with TestEnv():
    with BrianExperiment(persist=True, path="file.h5") as exp:
        G = NeuronPopulation(4, 'dv/dt=(1-v)/(10*ms):1', threshold='v > 0.1
↪', reset="v=0", method="rk4")
        G.monitor_spike(G.ids)
        connect = Connector(synapse_type="static")
        syn_pp = connect(G, G, G.ids, G.ids, connect=("bernoulli", {"p":0.3}
↪), on_pre='v += 0.1')
        exp.run(5*ms)
    with FileMap("file_analysis.h5") as af:
        with FileMap("file.h5") as f:
            for run in f.keys():
                exp_analysis = ExperimentAnalysis(experiment_data=f[run][
↪"data"])
                exp_analysis.analyze_all()
                af[run] = exp_analysis.report()
```

(continues on next page)

(continued from previous page)

```
    plotter = ExperimentPlotter(data=f, analysis=af)
    # define plots
    plotter.plot_spike_train()
    # draw plots
    plotter.draw()
    # show plots
    plotter.show()
```

Parameters

- **pop_name_e** (*str*) –
- **pop_name_i** (*str*) –
- **data** (*Union[Dict, persistence.Reader]*) –
- **analysis** (*Union[Dict, persistence.Reader]*) –
- **layout** (*str*) –
- **t_start** (*float*) –
- **t_end** (*float*) –

draw(*self*)

save(*self*, *path*)

Parameters **path** (*str*) –

show(*self*)

set_title(*self*, *title*)

set_window_title(*self*, *title*)

plot_spike_train(*self*, *pop_name=None*, *color={}*, *title=''*)
plot spike train of populations

Parameters

- **pop_name** (*List[str]*) – population names of populations tb plotted
- **color** (*Dict[str, str]*) – color by population names
- **title** (*str*) –

plot_voltage(*self*, *pop_name=[]*, *ids={}*, *color={}*)
plot voltages assuming specific neuron model for E and I populations

Parameters

- **pop_name** (*List[str]*) –
- **ids** (*Dict[str, List[int]]*) –
- **color** (*Dict[str, str]*) –

plot_variable(*self*, *variable*, *ylabel*, *scale=1.0*, *pop_name=[]*, *ids={}*, *color={}*, *title=''*)

Parameters

- **variable** (*Union[str, List[str]]*) – variable tb plotted or list of variable_names to be mapped to elements of pop_name
- **ylabel** (*str*) – label of the y axis (vs time on x axis)
- **scale** (*float*) – factor by which the variable values provided in __init__() param data are tb scaled
- **pop_name** (*List[str]*) – names of the populations tb plotted - by default uses __init__() params pop_e and pop_i
- **ids** (*Dict[str, List[int]]*) – mapping of population names to ids for which the variable is tb plotted
- **color** (*Dict[str, str]*) – mapping of population names to colors
- **title** (*str*) –

plot_instantaneous_rate(*self*, *pop_name=[]*, *color={}*, *title=''*)

Parameters

- **pop_name** (*List[str]*) –
- **color** (*Dict[str, str]*) –
- **title** (*str*) –

plot_smoothed_rate(*self*, *pop_name=[]*, *color={}*, *title=''*, *sync_freq={}*, *snr={}*)

Parameters

- **sync_freq** (*Dict[str, float]*) – synchronization frequency of the population rate by population
- **snr** (*Dict[str, float]*) – signal to noise ratio of the population rate by population
- **pop_name** (*List[str]*) –
- **color** (*Dict[str, str]*) –
- **title** (*str*) –

plot_cell_rate(*self*, *pop_name*, *color={}*, *title=''*)

Parameters

- **pop_name** (*str*) –
- **color** (*Dict[str, str]*) –

plot_power_spectrum(*self*, *pop_name*, *title=''*)
Plot log power spectral density against frequencies

Parameters

- **pop_name** (*str*) – name of the neuron population (mutually excl w/ pop_name_i)
- **title** (*str*) –

self.t_start_al and self.t_end_al are ignored here - as data points used in spectral analysis cannot simply be removed if this class defines a narrower bound

plot_power_spectogram_over_time(*self*, *pop_name*, *title=''*)
Plot power spectral density against frequency and time

Parameters

- **pop_name** (*str*) – name of the SpikeDeviceGroup for which the power spectrum is tb plotted
- **title** (*str*) –

self.t_start_al and self.t_end_al are ignored here - as data points used in spectral analysis cannot simply be removed if this class defines a narrower bound

plot_similarity_distribution(*self*, *pop_name*, *similarity_threshold*, *rows*, *cols*, *color={}*, *title=''*)
plot similarity distributions

Parameters

- **pop_name** (*str*) – name of the population
- **similarity_threshold** (*numpy.ndarray*) – similarity threshold for respective patterns in key 'similarity_distribution' in parameter analysis passed in `__init__()` (N x 1) - needs tb computed externally
- **rows** (*int*) – number of rows into which the axes/subplot is subdivided (rows * cols >= #patterns must hold)
- **cols** (*int*) – number of columns into which the axes/subplot is subdivided (rows * cols >= #patterns must hold)
- **color** (*Dict[str, str]*) –

plot_total_synaptic_conductance(*self*, *pop_name*, *color={}*, *title=''*)
plot total synaptic conductance

Parameters

- **pop_name** (*str*) – name of the population
- **color** (*Dict[str, str]*) –

1.14 mp_run

1.14.1 Module Contents

Functions

run_single_proc(f, parameters, kwargs, base_path)	
run_experiments(f, parameters, kwargs, base_path, multi_proc = True, num_procs = None)	
parse_cli_arg_iterable(s)	

Attributes

path
root_checked
path
parser

mp_run.**path**

mp_run.**root_checked = False**

mp_run.**path**

mp_run.**run_single_proc**(*f*, *parameters*, *kwargs*, *base_path*)

Parameters **f** (*Callable*) –

mp_run.**run_experiments**(*f*, *parameters*, *kwargs*, *base_path*, *multi_proc=True*, *num_procs=None*)

Parameters

- **f** (*Callable*) –
- **multi_proc** (*bool*) –
- **num_procs** (*int*) –

mp_run.**parse_cli_arg_iterable**(*s*)

Parameters **s** (*str*) –

mp_run.**parser**

1.15 distribution

1.15.1 Module Contents

Functions

draw_uniform(a = 0.0, b = 1.0, size = 1)	draw random samples from uniform distribution
draw_normal(mu = 0.0, sigma = 1.0, size = 1)	draw random samples from normal distribution
draw_poisson(lmbda = 1.0, size = 1)	draw random samples from poisson distribution
draw_exponential(lmbda = 1.0, size = 1)	draw random samples from exponential distribution
draw_bernoulli(p = 0.5, size = 1)	draw random samples from bernoulli distribution
draw_uniformly_random_from_values(values, size = 1)	draw samples from provided values uniformly at random (with replacement)

distribution.**draw_uniform**(*a=0.0*, *b=1.0*, *size=1*)
draw random samples from uniform distribution

Parameters

- **a** (*float*) – minimum value
- **b** (*float*) – maximum value
- **size** (*int*) – number of random samples tb drawn

Returns sampled values

Return type numpy.ndarray

distribution.**draw_normal**(*mu=0.0*, *sigma=1.0*, *size=1*)
draw random samples from normal distribution

Parameters

- **mu** (*float*) – mean of the distribution
- **sigma** (*float*) – standard deviation of the distribution
- **size** (*int*) – number of random samples tb drawn

Returns sampled values

Return type numpy.ndarray

distribution.**draw_poisson**(*lmbda=1.0*, *size=1*)
draw random samples from poisson distribution

Parameters

- **lmbda** (*float*) – lmbda of the distribution
- **size** (*int*) – number of random samples tb drawn

Returns sampled values

Return type numpy.ndarray

distribution.**draw_exponential**(*lmbda=1.0*, *size=1*)
draw random samples from exponential distribution

Parameters

- **lmbda** (*float*) – lmbda of the distribution
- **size** (*int*) – number of random samples tb drawn

Returns sampled values

Return type numpy.ndarray

distribution.**draw_bernoulli**(*p=0.5*, *size=1*)
draw random samples from bernoulli distribution

Parameters

- **p** (*float*) – success probability
- **size** (*int*) – number of random samples tb drawn

Returns sampled values

Return type numpy.ndarray

distribution.**draw_uniformly_random_from_values**(*values*, *size=1*)
draw samples from provided values uniformly at random (with replacement)

Parameters

- **values** (*numpy.ndarray*) – set of values from which samples are drawn uniformly at random
- **size** (*int*) – number of random samples tb drawn

Returns sampled values

Return type numpy.ndarray

1.16 differential_equations

1.16.1 Submodules

differential_equations.eif_equations

Equations for the exponential integrate-and-fire neuron with AMPA and GABA type synaptic input based on

- Nicolas Fourcaud-Trocmé, David Hansel, Carl Van Vreeswijk, and Nicolas Brunel. How spike generation mechanisms determine the neuronal response to fluctuating inputs. Journal of neuroscience, 23(37):11628–11640, 2003.
- Nicolas Brunel and Xiao-Jing Wang. What determines the frequency of fast network oscillations with irregular neural discharges? i. synaptic dynamics and excitation-inhibition balance. Journal of neurophysiology, 90(1):415–430, 2003.
- AAshqar :ref: https://github.com/AAshqar/GammaCoupling/blob/develop/NeuronsSpecs/NeuronEqs_DFsepI.py

Module Contents

differential_equations.eif_equations.**eq_eif = Multiline-String**

```
    dV/dt = (-gL*(V-eL) + gL*deltaT*exp((V-VT)/deltaT) - IsynE - IsynI -␣
↪IsynE_ext + Iext) / C : volt
    IsynE_ext = gsynE * (V - esynE) * synE_ext : amp
    IsynE = gsynE * (V - esynE) * synE : amp
    IsynI = gsynI * (V - esynI) * synI : amp
    dsynE_ext/dt = alpha * x_AMPA_ext - synE_ext/decay_AMPA : 1
    dsynE/dt = alpha * x_AMPA - synE/decay_AMPA : 1
    dsynI/dt = alpha * x_GABA - synI/decay_GABA : 1
    dx_AMPA_ext/dt = -x_AMPA_ext/rise_AMPA : 1
    dx_GABA/dt = -x_GABA/rise_GABA : 1
    dx_AMPA/dt = -x_AMPA/rise_AMPA : 1
    Iext : amp
```

differential_equations.eif_equations.**eq_eif_E**

differential_equations.eif_equations.**eq_eif_I**

differential_equations.eif_equations.**pre_eif_E = x_AMPA += psx_AMPA**

differential_equations.eif_equations.**pre_eif_I = x_GABA += psx_GABA**

differential_equations.eif_equations.**pre_eif_Pois = x_AMPA_ext += psx_AMPA_ext**

differential_equations.eif_parameters

Parameters for the exponential integrate-and-fire neuron with AMPA and GABA type synaptic input based on

- Nicolas Fourcaud-Trocmé, David Hansel, Carl Van Vreeswijk, and Nicolas Brunel. How spike generation mechanisms determine the neuronal response to fluctuating inputs. Journal of neuroscience, 23(37):11628–11640, 2003.
- Nicolas Brunel and Xiao-Jing Wang. What determines the frequency of fast network oscillations with irregular neural discharges? i. synaptic dynamics and excitation-inhibition balance. Journal of neurophysiology, 90(1):415–430, 2003.
- AAshqar :ref:https://github.com/AAshqar/GammaCoupling/blob/develop/NeuronsSpecs/NeuronParams.py

Module Contents

differential_equations.eif_parameters.**C_E**

differential_equations.eif_parameters.**C_I**

differential_equations.eif_parameters.**gL_E**

differential_equations.eif_parameters.**gL_I**

differential_equations.eif_parameters.**eL_E**

differential_equations.eif_parameters.**eL_I**

differential_equations.eif_parameters.**refractory_E**

differential_equations.eif_parameters.**refractory_I**

differential_equations.eif_parameters.**deltaT**

differential_equations.eif_parameters.**VT**

differential_equations.eif_parameters.**V_thr**

differential_equations.eif_parameters.**V_r**

differential_equations.eif_parameters.**esynE**

differential_equations.eif_parameters.**esynI**

differential_equations.eif_parameters.**rise_AMPA**

differential_equations.eif_parameters.**rise_GABA**

differential_equations.eif_parameters.**decay_AMPA**

differential_equations.eif_parameters.**decay_GABA**

differential_equations.eif_parameters.**gsynE_E**

differential_equations.eif_parameters.**gsynI_E**

differential_equations.eif_parameters.**gsynE_I**

differential_equations.eif_parameters.**gsynI_I**

differential_equations.eif_parameters.**latency_AMPA**

differential_equations.eif_parameters.**latency_GABA**

differential_equations.eif_parameters.**psx_AMPA = 1.0**

differential_equations.eif_parameters.**psx_GABA = 1.0**

differential_equations.eif_parameters.**psx_AMPA_ext = 1.5**

differential_equations.eif_parameters.**alpha**

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z