Karenina

Simulation and modeling tools for studying Anna Karenina effects in animal microbiomes This package aims to develop tools for modeling microbiome variability in disease. Initial versions focus on simulating microbiome change over time using simple Ornstein-Uhlenbeck (OU) models.

Usage & Installation can be found at:

- https://www.github.com/zaneveld/karenina
- https://www.github.com/zaneveld/q2-karenina

Indices and tables

- Index
- Module Index
- Search Page

Spatial Ornstein Uhlenbeck

karenina.spatial_ornstein_uhlenbeck.**check_perturbation_timepoint**(*perturbation_timepoint*, *n_timepoints*)

Raise ValueError if perturbation_timepoint is < 0 or >n_timepoints

Parameters:
- **perturbation_timepoint** – defined timepoint for perturbation application
- **n_timepoints** – number of timepoints

karenina.spatial_ornstein_uhlenbeck.**ensure_exists**(*output_dir*)

Ensure that output_dir exists

Parameters: **output_dir** – path to output directory

karenina.spatial_ornstein_uhlenbeck.**parse_perturbation_file**(*pert_file_path*, *perturbation_timepoint*, *perturbation_duration*)

Return a list of perturbations

infile – a .tsv file describing one perturbation per line assume input file is correctly formatted (no warnings if not)

NOTE: each pertubation should be in the format:

set_xyz_lambda_low = {“start”:opts.perturbation_timepoint, “end”:opts.perturbation_timepoint + opts.perturbation_duration, “params”:{“lambda”:0.005}, “update_mode”:”replace”, “axes”: [“x”,”y”,”z”]}

Parameters:
- **pert_file_path** – perturbation file path
- **perturbation_timepoint** – timepoint to apply perturbation
- **perturbation_duration** – duration of perturbation

Returns: perturbation list parsed from pert file contents

karenina.spatial_ornstein_uhlenbeck.**write_options_to_log**(*log*, *opts*)

Writes user's input options to log file

Parameters:
- **log** – log filename
- **opts** – options

Fit Timeseries

karenina.fit_timeseries.**aic**(*n_param*, *nLogLik*)

calculates AIC with 2 * n_parameters - 2 * LN(-1 * nLogLik)

Parameters:
- **n_param** – number of parameters
- **nLogLik** – negative log likelihood

Returns: aic score

karenina.fit_timeseries.**fit_cohorts**(*input*, *ind*, *tp*, *tx*, *method*, *verbose=False*)

Completes the same operation as fit_input, for cohorts. Fit and minimize the timeseries for each subject and cohort defined.

Parameters:
- **input** – dataframe: [#SampleID,individual,timepoint,treatment,pc1,pc2,pc3]
- **ind** – subject column identifier
- **tp** – timepoint column identifier
- **tx** – treatment column identifier
- **method** – "basinhopping" if not defined in opts

Returns: dataframe:
[ind,"pc","sigma","lambda","theta","optimizer","nLogLik","n_parameters","aic",tp,tx,"x"]

karenina.fit_timeseries.**fit_input**(*input*, *ind*, *tp*, *tx*, *method*)

Fit and minimize the timeseries for each subject defined

Parameters:
- **input** – dataframe: [#SampleID,individual,timepoint,treatment,pc1,pc2,pc3]
- **ind** – subject column identifier
- **tp** – timepoint column identifier
- **tx** – treatment column identifier
- **method** – "basinhopping" if not defined in opts

Returns: dataframe:
[ind,"pc","sigma","lambda","theta","optimizer","nLogLik","n_parameters","aic",tp,tx,"x"]

karenina.fit_timeseries.**fit_normal**(*data*)

Return the mean and standard deviation of normal data

Parameters: **data** – fit data to normal distribution

Returns: mu, std, nLogLik

karenina.fit_timeseries.**fit_timeseries**(*fn_to_optimize*, *x0*, *xmin=array([-inf*, *-inf*, *-inf])*, *xmax=array([inf*, *inf*, *inf])*, *global_optimizer='basinhopping'*, *local_optimizer='Nelder-Mead'*, *stepsize=0.01*, *niter=200*, *verbose=False*)

Minimize a function returning input & result fn_to_optimize – the function to minimize. Must return a single value or array x.

x0 – initial parameter value or array of parameter values xmax – max parameter values (use inf for infinite) xmin – min parameter values (use -inf for infinite) global_optimizer – the global optimization method (see scipy.optimize) local_optimizer – the local optimizer (must be supported by global method)

Parameters:
- **fn_to_optimize** – function that is being optimized, generated from fn_to_optimize
- **x0** – initial parameter value or array of parameter values
- **xmin** – min parameter values (-inf for infinite)
- **xmax** – max parameter values (inf for infinite)
- **global_optimizer** – global optimization method
- **local_optimizer** – local optimization method (must be supported by global)
- **stepsize** – size for each step (.01)
- **niter** – number of iterations (200)
- **verbose** – verbose output, default = False

Returns: global_min, f_at_global_min

karenina.fit_timeseries.**gen_output**(*fit_ts*, *ind*, *tp*, *tx*, *method*)

Generate output dataframe for either cohort, or non-cohort data

Parameters:
- **fit_ts** – dataframe [0: [[Sigma, Lambda, Theta], nLogLik], 1: Individuals, 2: Times 3: Treatments, 4: Values, 5: PC Axis
- **ind** – individual identifier
- **tp** – timepoint identifier
- **tx** – treatment identifier
- **method** – optimization method

Returns: Formatted dataframe for csv output

karenina.fit_timeseries.**get_OU_nLogLik**(*x*, *times*, *Sigma*, *Lambda*, *Theta*)

Return the negative log likelihood for an OU model given data x – an array of x values, which MUST be ordered by time times – the time values at which x was taken. MUST match x in order.

OU model parameters

1. Sigma – estimated Sigma for OU model (extent of change over time)
2. Lambda – estimated Lambda for OU model (tendency to return to average position)
3. Theta – estimated Theta for OU model (average or 'home' position)

Parameters:
- **x** – array of values ordered by time
- **times** – times associated with x values
- **Sigma** – initial sigma value
- **Lambda** – initial lambda value
- **Theta** – initial theta value

Returns: nLogLik value

karenina.fit_timeseries.**make_OU_objective_fn**(*x*, *times*, *verbose=False*)

Make an objective function for use with basinhopping with data embedded

scipy.optimize.basinhopping needs a single array p, a function, and will minimize f(p). So we want to embded or dx data and time data *in* the function, and use the values of p to represent parameter values that could produce the data.

Parameters:
- **x –** individual x values for objective function
- **times –** timepoints associated with passed-in x values
- **verbose –** verbose output, default = False

Returns: fn_to_optimize

karenina.fit_timeseries.**make_OU_objective_fn_cohort**(*x*, *times*, *verbose=False*)

Sums nLogLik and build objective function based on cohorts. Operates in the same manner as make_OU_objective_fn, except that it considers a treatment cohort, not just the individuals.

Make an objective function for use with basinhopping with data embedded

scipy.optimize.basinhopping needs a single array p, a function, and will minimize f(p). So we want to embded or dx data and time data *in* the function, and use the values of p to represent parameter values that could produce the data.

Overall strategy: Treat this exactly like the per-individual fitting, BUT within the objective function loop over all individuals in a given treatment (not just one) to get nLogLiklihoods. The nLogLiklihood for the individuals in the treatment is then just the sum of the individual nLogLikelihoods for each individuals timeseries.

data – a dict of {“individual1”: (fixed_x,fixed_times)}

Parameters:
- **x –** x values for cohort objective function
- **times –** times associated with x values from cohort
- **verbose –** verbose output, default = False

Returns: fn_to_optimize

karenina.fit_timeseries.**parse_metadata**(*metadata*, *individual*, *timepoint*, *treatment*, *site*)

Parse relevant contents from metadata file to complete input dataframe

Parameters:
- **metadata –** tsv file location
- **individual** – subject column identifier
- **timepoint –** timepoint column identifier
- **treatment –** treatment column identifier
- **site –** [subjectID, x1, x2, x3]

Returns: input dataframe: [#SampleID,individual,timepoint,treatment,pc1,pc2,pc3]

karenina.fit_timeseries.**parse_pcoa**(*pcoa_qza*, *individual*, *timepoint*, *treatment*, *metadata*)

Load data from PCoA output in Qiime2 Format

Parameters:
- **pcoa_qza –** Location of PCoA file
- **individual** – Subject column identifier(s) [ex: Subject; Subject,BodySite]
- **timepoint –** Timepoint column identifier
- **treatment –** Treatment column identifier
- **metadata –** optionally defined metadata file location, if not defined, will use metadata from PCoA.qza

Returns: input dataframe, tsv filepath location

Fit Timeseries Benchmark

karenina.fit_timeseries_benchmark.**benchmark**(*max_tp=300*, *output=None*, *verbose=False*)

Verifies that fit_timeseries recovers OU model params

Parameters:
- **output** – location for output log
- **max_tp** – maximum timepoints to test
- **verbose** – verbosity

Returns: output dataframe of benchmarked data

karenina.fit_timeseries_benchmark.**vis**(*df*, *output*)

Visualizes benchmarking output error for various tested timepoints

Parameters:
- **df** – Dataframe containing model, timepoints, and list of errors
- **output** – output directory

Visualization

karenina.visualization.**get_timeseries_data**(*individuals, axes=['x', 'y', 'z']*)

karenina.visualization.**save_simulation_data**(*data*, *ids*, *output*)

Saves simulation output data in PCoA format

Parameters:
- **data** – data to save
- **ids** – Sample_IDs
- **output** – output filepath

karenina.visualization.**save_simulation_figure**(*individuals*, *output_folder*, *n_individuals*, *n_timepoints*, *perturbation_timepoint*)

Save a .pdf image of the simulated PCoA plot

Parameters:
- **individuals** – array of individuals
- **output_folder** – output filepath
- **n_individuals** – number of individuals
- **n_timepoints** – number of timepoints
- **perturbation_timepoint** – timepoint of perturbation application

karenina.visualization.**save_simulation_movie**(*individuals*, *output_folder*, *n_individuals*, *n_timepoints*, *black_background=True*, *verbose=False*)

Save an .ffmpg move of the simulated community change

Parameters:
- **individuals** – array of individuals to visualize
- **output_folder** – output directory filepath
- **n_individuals** – number of individuals
- **n_timepoints** – number of timepoints
- **black_background** – T/F, default = True
- **verbose** – verbose output, default = False

karenina.visualization.**update_3d_plot**(*end_t*, *timeseries_data*, *ax*, *lines*, *points=None*, *start_t=0*)

Updates visualization 3d plot

Parameters:
- **end_t –** end timepoint
- **timeseries_data –** data from timeseries
- **ax –** visualization ax
- **lines** – lines of data
- **points –** values to update
- **start_t –** start timepoint (0)

Experiment

class karenina.experiment. **Experiment**(*treatment_names*, *n_individuals*, *n_timepoints*, *individual_base_params*, *treatment_params*, *interindividual_variation*, *verbose*)

Bases: **object**

This class has responsibility for simulating an experimental design for simulation.

A fixed number of individuals are simulated across experimental conditions called 'treatments' Each treatment can have different numbers of individuals. All treatment must have the same number of timepoints.

Each treatment can be associated with the imposition of one or more Perturbations. Each perturbation is inserted or removed from all individuals at a fixed time-point.

check_n_timepoints_is_int(*n_timepoints*)

Raise a ValueError if n_timepoints can't be cast as an int

Parameters: **n_timepoints –** number of timepoints

check_variable_specified_per_treatment(*v*, *verbose=False*)

Raise a ValueError if v is not the same length as the number of treatments

Parameters:
- **v –** variable
- **verbose –** verbose output, default = False

q2_data()

generate output data from object's self for Qiime2

Returns: data, ids

run()

Run the experiment, simulating timesteps; saves the simulation figure and movie

simulate_timestep(*t*)

Simulate timestep t of the experiemnt

Approach:

1. First apply any perturbations that should be active but aren't yet
2. Second, simulate the timestep
3. Third, remove any perturbations that should be off

NOTE: this means that a perturbation that starts and ends at t=1 will be active at t=1 That is, the start and end times are inclusive.

Parameters: **t** – timestep

`simulate_timesteps`(*t_start*, *t_end*, *verbose=False*)

Simulate multiple timesteps

Parameters:
- **t_start** – start timepoint
- **t_end** – end timepoint
- **verbose** – verbose output, default = False

`write_to_movie_file`(*output_folder*, *verbose=False*)

Write an MPG movie to output folder

Parameters:
- **output_folder** – output directory
- **verbose** – verbose output, default = False

Individual

class `karenina.individual.`**`Individual`**(*subject_id, coords=['x', 'y', 'z'], metadata={}, params={}, interindividual_variation=0.01, verbose=False*)

Bases: `object`

Generates an individual for OU simulation

`apply_perturbation`(*perturbation*)

Apply a perturbation to the appropriate axes

Parameters: **perturbation** – perturbation to apply

`apply_perturbation_to_axis`(*axis*, *perturbation*)

Apply a perturbation to a Processes objects

Parameters:
- **axis** – Axis to apply perturbation to
- **perturbation** – perturbation to apply

`check_identity`(*verbose=False*)

Check identity of movement process :param verbose: verbose output, default = False :return: True if processes are equivalent, False if not

`get_data`(*n_timepoints*)

get data from movement processes

Parameters: **n_timepoints** – number of timepoints to gather data for
Returns: data for timepoints

`remove_perturbation`(*perturbation*)

Remove a perturbation from the appropriate axes

Parameters: **perturbation** – perturbation to remove

remove_perturbation_from_axis(*axis*, *perturbation*)

Remove a perturbation from one or more Process objects

Parameters:
- **axis** – axis to remove perturbation from
- **perturbation** – perturbation to remove from axis

simulate_movement(*n_timepoints*, *params=None*)

Simulate movement over timepoints

Parameters:
- **n_timepoints** – number of timepoints to simulate
- **params** – parameters to change from baseparams

karenina.individual.**random**() → x in the interval [0, 1).

Perturbation

class karenina.perturbation.**Perturbation**(*start, end, params, update_mode='replace', axes=['x', 'y', 'z']*)

Bases: **object**

Alter a simulation to impose a press disturbance, shifting the microbiome torwards a new configuration

start –inclusive timepoint to start perturbation. Note that this is read at the Experiment level, not by underlying Process objects.

end – inclusive timepoint to end perturbation. Note that this is read at the Experiment level, not by underlying Process objects.

params – dict of parameter values altered by the disturbance

- 'mode' – how the perturbation updates parameter values.
- 'replace' – replace old value with new one
- 'add' – add new value to old one
- 'multiply' – multiply the two values

axes – axes to which the perturbation applies. Like Start and End this is a 'dumb' value, read externally by the Experiment object

is_active(*t*)

determines if timepoint is active

Parameters: **t** – timepoint
Returns: True if timepoint is active, False if not

update_by_addition(*curr_param*, *perturbation_param*)

Update parameters by addition

Parameters: **curr_param** – current parameter

- **perturbation_param** – new parameter

Returns: curr_param + perturbation_param

update_by_multiplication(*curr_param*, *perturbation_param*)

Update parameters by multiplication

Parameters:
- **curr_param** – current parameter
- **perturbation_param** – new parameter

Returns: curr_param * perturbation_param

update_by_replacement(*curr_param*, *perturbation_param*)

update parameters by replacement :param curr_param: current parameter :param perturbation_param: new parameter :return: perturbation_param

update_params(*params*)

Update baseparams

Parameters: **params** – params to update

Returns: new_params

Process

class karenina.process.**Process**(*start_coord*, *motion='Ornstein-Uhlenbeck'*, *history=None*, *params={'L': 0.2*, *'delta': 0.25}*)

Bases: **object**

Represents a 1d process in a Euclidean space

bm_change(*dt*, *delta*)

Change the Brownian motion process

Parameters:
- **dt** – time elapsed since last update
- **delta** – delta value to adjust

Returns: change

bm_update(*dt*, *delta*)

Update the Brownian motion process

Parameters:
- **dt** – time elapsed since last update
- **delta** – delta value to adjust

Returns: change

ou_change(*dt*, *mu*, *L*, *delta*)

Change the Ornstein Uhlenbeck motion process

The Ornstein Uhlenbeck process is modelled as:

ds = lambda * (mu - s) * dt + dW

ds – change in our process from the last timepoint L – lambda, the speed of reversion to mean (NOTE: lambda is a reserved keyword in Python so I use L) mu – mean position s – current position dt – how much time has elapsed since last update dW – the Weiner Process (basic Brownian motion)

This says we update as usual for Brownian motion, but add in a term that reverts us to some mean position (mu) over time (dt) at some speed (lambda)

Parameters:
- **dt** – time elapsed since last update
- **delta** – delta value to adjust

Returns: change in process since last timepoint

`ou_update`(*dt*, *mu*, *L*, *delta*, *min_bound=-1.0*, *max_bound=1.0*)

Update the Brownian motion process

Parameters:
- **dt** – time elapsed since last update
- **mu** – mean position
- **L** – speed of reversion to the mean
- **delta** – variance from the mean
- **min_bound** – minimum bound
- **max_bound** – maximum bound

`update`(*dt*)

Update the process

Parameters: **dt** – time elapsed since last update