VolterraBasis Documentation

Hadrien Vroylandt

Jun 26, 2023

DOCUMENTATION

This project compute position-dependent memory kernel for Generalized Langevin Equations. Please refer to Position-dependent memory kernel in generalized Langevin equations: Theory and numerical estimation, J. Chem. Phys. 156, 244105 (2022); https://doi.org/10.1063/5.0094566, also available at https://arxiv.org/abs/2201.02457 for a detailed description of the algorithm.

CHAPTER

ONE

QUICK START

This package aims at computing memory kernel when studying Generalized Langevin Equations (GLE).

1.1 Inversion of Volterra Integral Equations

Several algorithms for the inversion of the Volterra Integral Equations are available. Please refer to P. Linz, "Numerical methods for Volterra integral equations of the first kind", The Computer Journal 12, 393–397 (1969) for mathematical details.

1.1.1 Functionnal basis

The estimation of the memory kernel necessite the choice of a functionnal basis. Functional basis are implemented in `VolterraBasis.basis` that could be imported and initialized as

```
>>> import VolterraBasis.basis as bf
>>> basis=bf.BSplineFeatures(15)
```

Several options are available for the type of basis, please refer to the documentation. Although multidimensionnal trajectories can be analysed, not all functionnal basis are multidimensionnal.

1.1.2 Force and memory estimate

Once the mean force and memory have been computed, the value of the force and memory kernel at given position can be computed trought function *`VolterraBasis.Pos_gle.force_eval()`* and *`VolterraBasis.Pos_gle.kernel_eval()`*

1.1.3 Choice of the form of the GLE

Several options are available to choose the form of the GLE:

- *`VolterraBasis.Pos_gle`* implement the form of the GLE featured in Vroylandt and Monmarché with memory kernel linear in velocity.
- *`VolterraBasis.Pos_gle_with_friction`* is similar to the previous but don't assume that the instantaneous friction is zero.
- *`VolterraBasis.Pos_gle_const_kernel`* is the traditionnal GLE with memory kernel linear in velocity and independant of position.

- *VolterraBasis.Pos_gle_no_vel_basis* implement a GLE where the memory kernel has no dependance in velocity.
- *VolterraBasis.Pos_gle_overdamped* compute the memory kernel for an overdamped dynamics.

CHAPTER

TWO

VOLTERRABASIS API

2.1 Loading trajectories

`xframe`(x, time[, v, a, fix_time, round_time, dt])	Creates a xarray dataset (['t', 'x']) from a trajectory.
`compute_va`(xf[, correct_jumps, jump, ...])	Computes velocity and acceleration from a dataset with ['t', 'x'] as returned by xframe.
`compute_a`(xvf)	Computes the acceleration from a dataset with ['t', 'x', 'v'].
`concat_underdamped`(xva)	Return the DataSet such that x is now (x,v) and v is now (v,a),
`compute_1d_fe`(xva_list[, bins, kT, hist])	Computes the free energy from the trajectoy using a cubic spline interpolation.

2.1.1 `VolterraBasis.xframe`

`VolterraBasis.`**`xframe`**(*x*, *time*, *v=None*, *a=None*, *fix_time=False*, *round_time=0.0001*, *dt=-1*)

Creates a xarray dataset ([‘t’, ‘x’]) from a trajectory.

Parameters

x
[array] The time series. The array can be in any type as long as xarray can handle it. This include numpy array, dask array,…

time
[numpy array] The respective time values.

fix_time
[bool, default=False] Round first timestep to round_time precision and replace times.

round_time
[float, default=1.e-4] When fix_time is set times are rounded to this value.

dt
[float, default=-1] When positive, this value is used for fixing the time instead of the first timestep.

v
[numpy array, default=None] Velocity if computed externally

a
[numpy array, default=None] Acceleration if computed externally

Examples using `VolterraBasis.xframe`

- *Memory Kernel Estimation with the usual GLE*
- *Checking solution of volterra equation*
- *Method comparaison*
- *Prony Series Estimation*
- *Memory Kernel fit*
- *Memory Kernel Estimation*
- *Memory Kernel Estimation with the usual GLE*
- *Kernel Estimation for 2D observable*
- *Generalized Fokker Planck equation*
- *Generalized Fokker Planck equation in underdamped case*
- *GLE Integration*

2.1.2 `VolterraBasis.compute_va`

`VolterraBasis.`**`compute_va`**(*xf*, *correct_jumps=False*, *jump=6.283185307179586*, *jump_thr=5.497787143782138*, *lamb_finite_diff=0.5*)

Computes velocity and acceleration from a dataset with ['t', 'x'] as returned by xframe.

Parameters

xf
[xarray dataframe (['t', 'x'])]

correct_jumps
[bool, default=False] Jumps in the trajectory are removed (relevant for periodic data).

Examples using `VolterraBasis.compute_va`

- *Memory Kernel Estimation with the usual GLE*
- *Checking solution of volterra equation*
- *Method comparaison*
- *Prony Series Estimation*
- *Memory Kernel fit*
- *Memory Kernel Estimation*
- *Memory Kernel Estimation with the usual GLE*
- *Kernel Estimation for 2D observable*
- *Generalized Fokker Planck equation*
- *Generalized Fokker Planck equation in underdamped case*
- *GLE Integration*

2.1.3 `VolterraBasis.compute_a`

`VolterraBasis.`**`compute_a`**(*xvf*)

Computes the acceleration from a dataset with ['t', 'x', 'v'].

Parameters

xvf
[xarray dataset (['x', 'v'])]

Examples using `VolterraBasis.compute_a`

- *GLE Integration*

2.1.4 `VolterraBasis.concat_underdamped`

`VolterraBasis.`**`concat_underdamped`**(*xva*)

Return the DataSet such that x is now (x,v) and v is now (v,a),

Examples using `VolterraBasis.concat_underdamped`

- *Generalized Fokker Planck equation in underdamped case*

2.1.5 `VolterraBasis.compute_1d_fe`

`VolterraBasis.`**`compute_1d_fe`**(*xva_list*, *bins=150*, *kT=2.494*, *hist=False*)

Computes the free energy from the trajectoy using a cubic spline interpolation.

Parameters

bins
[str, or int, default="auto"] The number of bins. It is passed to the numpy.histogram routine, see its documentation for details.

hist: bool, default=False
If False return the free energy else return the histogram

2.2 Memory kernel estimation

`Estimator_gle`(xva_arg, model_class, basis[, ...])	The main class for the position dependent memory extraction holding all data.

2.2.1 `VolterraBasis`.Estimator_gle

class VolterraBasis.**Estimator_gle**(*xva_arg*, *model_class*, *basis*, *trunc=1.0*, *L_obs=None*, *saveall=True*, *prefix=''*, *verbose=True*, *n_jobs=1*, ***kwargs*)

The main class for the position dependent memory extraction holding all data.

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

L_obs: str, default given by the model
Name of the column containing the time derivative of the observable

__init__(*xva_arg*, *model_class*, *basis*, *trunc=1.0*, *L_obs=None*, *saveall=True*, *prefix=''*, *verbose=True*, *n_jobs=1*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

L_obs: str, default given by the model
Name of the column containing the time derivative of the observable

`check_volterra_inversion`(*return_diff=False*)

For checking if the volterra equation is correctly inversed Compute the integral in volterra equation using trapezoidal rule. This only check the volterra of the first kind

Parameters

return_diff
[bool, default = False] Indicate if you want the result of the intégral or the difference between the result and the expected value

`compute_basis_mean`(*basis_type='force'*)

Compute mean value of the basis function

`compute_corrs`(*large=False*, *rank_tol=None*, ***kwargs*)

Compute correlation functions.

Parameters

large
[bool, default=False] When large is true, it use a slower way to compute correlation that is less demanding in memory

rank_tol: float, default=None
Tolerance for rank computation in case of projection onto the range of the basis

second_order_method:bool, default = True
If set to False do less computation but prevent to use second_order method in Volterra

`compute_effective_mass`()

Return average effective mass computed from equipartition with the velocity.

`compute_gram_kernel`()

Return gram matrix of the kernel part of the basis.

`compute_kernel`(*method='rectangular'*, *k0=None*)

Computes the memory kernel.

Parameters

method
[{"rectangular", "midpoint", "midpoint_w_richardson","trapz","second_kind_rect","second_kind_trapz"}, default=rectangular] Choose numerical method of inversion of the volterra equation

k0
[float, default=0.] If you give a nonzero value for k0, this is used at time zero for the trapz and second kind method. If set to None, the F-routine will calculate k0 from the second kind memory equation.

`compute_mean_force`()

Computes the mean force from the trajectories.

`compute_pos_effective_mass`()

Return position-dependent effective inverse mass

`compute_projected_corrs`(*left_op=None*)

Compute correlation between noise and left_op using the projected correlations

describe_data()

Return a description of the data

to_gfpe(*model=None*, *new_obs_name='dE'*)

Update trajectories to compute derivative of the basis function

Examples using `VolterraBasis.Estimator_gle`

- *Memory Kernel Estimation with the usual GLE*
- *Checking solution of volterra equation*
- *Method comparaison*
- *Prony Series Estimation*
- *Memory Kernel fit*
- *Memory Kernel Estimation*
- *Memory Kernel Estimation with the usual GLE*
- *Kernel Estimation for 2D observable*
- *Generalized Fokker Planck equation*
- *Generalized Fokker Planck equation in underdamped case*
- *GLE Integration*

2.3 Available models of GLE

Pos_gle(*args, **kwargs)	The main class for the position dependent memory extraction, holding all data and the extracted memory kernels.
Pos_gle_with_friction(*args, **kwargs)	A derived class in which we don't enforce zero instantaneous friction
Pos_gle_no_vel_basis(*args, **kwargs)	Use basis function dependent of the position only
Pos_gle_const_kernel(*args, **kwargs)	A derived class in which we the kernel is computed independent of the position
Pos_gle_overdamped(*args[, L_obs, ...])	Extraction of position dependent memory kernel for overdamped dynamics.
Pos_gle_hybrid(*args, **kwargs)	Implement the hybrid projector of arXiv:2202.01922

2.3.1 VolterraBasis.Pos_gle

class VolterraBasis.**Pos_gle**(**args*, ***kwargs*)

The main class for the position dependent memory extraction, holding all data and the extracted memory kernels.

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`__init__`(**args*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`basis_vector`(*xva*, *compute_for='corrs'*)

From one trajectory compute the basis element. This is the main method that should be implemented by children class. It take as argument a trajectory and should return the value of the basis function depending of the wanted case. There is three case that should be implemented.

"force": for the evaluation and computation of the mean force.

"pmf": for evaluation of the pmf using integration of the mean force

"kernel": for the evaluation of the kernel.

"corrs": for the computation of the correlation function.

`compute_corrs_w_noise`(*xva*, *left_op=None*)

Compute correlation between noise and left_op

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

`compute_noise`(*xva*, *trunc_kernel=None*, *start_point=0*, *end_point=None*)

From a trajectory get the noise.

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

trunc_kernel
[int] Number of datapoint of the kernel to consider. Can be used to remove unphysical divergence of the kernel or shorten execution time.

`evolve_volterra`(*G0*, *lenTraj*, *method='trapz'*, *trunc_ind=None*)

Evolve in time the integro-differential equation. This assume that the GLE is a linear GLE (i.e. the set of basis function is on the left and right of the equality)

Parameters

G0
[array] Initial value of the correlation

lenTraj
[int] Length of the time evolution

method
[str, default="trapz"] Method that is used to discretize the continuous Volterra equations

trunc_ind: int, default= self.trunc_ind
Truncate the length of the memory to this value

`flux_from_volterra`(*corrs_force*, *corrs_kernel=None*, *force_coeff=None*, *kernel=None*, *method='trapz'*, *trunc_ind=None*)

From a solution of the Volterra equation, compute the flux term. That allow to compute decomposition of the flux

`force_eval`(*x*, *coeffs=None*)

Evaluate the force at given points x. If coeffs is given, use provided coefficients instead of the force

`inv_mass_eval`(*x*, *coeffs=None*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

kernel_eval(*x*, *coeffs_ker=None*)

Evaluate the kernel at given points x. If coeffs_ker is given, use provided coefficients instead of the kernel

laplace_transform_kernel(*s_start=0.0*, *s_end=None*, *n_points=None*)

Compute the Laplace transform of the kernel matrix

classmethod load_model(*basis*, *coeffs*, ***kwargs*)

Create a model from a save

pmf_eval(*x*, *coeffs=None*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

pmf_num_int_eval(*x*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This take into accound the position dependent mass, but the integration is numeric

save_model()

Return DataSet version of the model than can be save to file

Examples using `VolterraBasis.Pos_gle`

- *Checking solution of volterra equation*
- *Method comparaison*
- *Memory Kernel Estimation*
- *Kernel Estimation for 2D observable*

2.3.2 `VolterraBasis`.Pos_gle_with_friction

class VolterraBasis.**Pos_gle_with_friction**(**args*, ***kwargs*)

A derived class in which we don't enforce zero instantaneous friction

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset (['time', 'x', 'v', 'a']) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`__init__`(**args*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`basis_vector`(*xva*, *compute_for='corrs'*)

From one trajectory compute the basis element. This is the main method that should be implemented by children class. It take as argument a trajectory and should return the value of the basis function depending of the wanted case. There is three case that should be implemented.

“force”: for the evaluation and computation of the mean force.

“pmf”: for evaluation of the pmf using integration of the mean force

“kernel”: for the evaluation of the kernel.

“corrs”: for the computation of the correlation function.

`compute_corrs_w_noise`(*xva*, *left_op=None*)

Compute correlation between noise and left_op

Parameters

xva
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

`compute_noise`(*xva*, *trunc_kernel=None*, *start_point=0*, *end_point=None*)

From a trajectory get the noise.

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

trunc_kernel
[int] Number of datapoint of the kernel to consider. Can be used to remove unphysical divergence of the kernel or shortten execution time.

`evolve_volterra`(*G0*, *lenTraj*, *method='trapz'*, *trunc_ind=None*)

Evolve in time the integro-differential equation. This assume that the GLE is a linear GLE (i.e. the set of basis function is on the left and right of the equality)

Parameters

G0
[array] Initial value of the correlation

lenTraj
[int] Length of the time evolution

method
[str, default="trapz"] Method that is used to discretize the continuous Volterra equations

trunc_ind: int, default= self.trunc_ind
Truncate the length of the memory to this value

`flux_from_volterra`(*corrs_force*, *corrs_kernel=None*, *force_coeff=None*, *kernel=None*, *method='trapz'*, *trunc_ind=None*)

From a solution of the Volterra equation, compute the flux term. That allow to compute decomposition of the flux

`force_eval`(*x*)

Evaluate the force for the position dependent part only

`friction_force_eval`(*x*)

Compute the term of friction, that should be zero

`inv_mass_eval`(*x*, *coeffs=None*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`kernel_eval`(*x*, *coeffs_ker=None*)

Evaluate the kernel at given points x. If coeffs_ker is given, use provided coefficients instead of the kernel

`laplace_transform_kernel`(*s_start=0.0*, *s_end=None*, *n_points=None*)

Compute the Laplace transform of the kernel matrix

`classmethod load_model`(*basis*, *coeffs*, ***kwargs*)

Create a model from a save

`pmf_eval`(*x*, *coeffs=None*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`pmf_num_int_eval`(*x*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This take into accound the position dependent mass, but the integration is numeric

save_model()

Return DataSet version of the model than can be save to file

2.3.3 VolterraBasis.Pos_gle_no_vel_basis

class VolterraBasis.**Pos_gle_no_vel_basis**(**args*, ***kwargs*)

Use basis function dependent of the position only

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

__init__(**args*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`basis_vector`(*xva*, *compute_for='corrs'*)

From one trajectory compute the basis element. This is the main method that should be implemented by children class. It take as argument a trajectory and should return the value of the basis function depending of the wanted case. There is three case that should be implemented.

"force": for the evaluation and computation of the mean force.

"pmf": for evaluation of the pmf using integration of the mean force

"kernel": for the evaluation of the kernel.

"corrs": for the computation of the correlation function.

`compute_corrs_w_noise`(*xva*, *left_op=None*)

Compute correlation between noise and left_op

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

`compute_noise`(*xva*, *trunc_kernel=None*, *start_point=0*, *end_point=None*)

From a trajectory get the noise.

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

trunc_kernel
[int] Number of datapoint of the kernel to consider. Can be used to remove unphysical divergence of the kernel or shortten execution time.

`evolve_volterra`(*G0*, *lenTraj*, *method='trapz'*, *trunc_ind=None*)

Evolve in time the integro-differential equation. This assume that the GLE is a linear GLE (i.e. the set of basis function is on the left and right of the equality)

Parameters

G0
[array] Initial value of the correlation

lenTraj
[int] Length of the time evolution

method
[str, default="trapz"] Method that is used to discretize the continuous Volterra equations

trunc_ind: int, default= self.trunc_ind
Truncate the length of the memory to this value

flux_from_volterra(*corrs_force*, *corrs_kernel=None*, *force_coeff=None*, *kernel=None*, *method='trapz'*, *trunc_ind=None*)

From a solution of the Volterra equation, compute the flux term. That allow to compute decomposition of the flux

force_eval(*x*, *coeffs=None*)

Evaluate the force at given points x. If coeffs is given, use provided coefficients instead of the force

inv_mass_eval(*x*, *coeffs=None*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

kernel_eval(*x*, *coeffs_ker=None*)

Evaluate the kernel at given points x. If coeffs_ker is given, use provided coefficients instead of the kernel

laplace_transform_kernel(*s_start=0.0*, *s_end=None*, *n_points=None*)

Compute the Laplace transform of the kernel matrix

classmethod **load_model**(*basis*, *coeffs*, ***kwargs*)

Create a model from a save

pmf_eval(*x*, *coeffs=None*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

pmf_num_int_eval(*x*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This take into accound the position dependent mass, but the integration is numeric

save_model()

Return DataSet version of the model than can be save to file

2.3.4 VolterraBasis.Pos_gle_const_kernel

class VolterraBasis.**Pos_gle_const_kernel**(**args*, ***kwargs*)

A derived class in which we the kernel is computed independent of the position

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`__init__`(**args*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`basis_vector`(*xva*, *compute_for='corrs'*)

From one trajectory compute the basis element. This is the main method that should be implemented by children class. It take as argument a trajectory and should return the value of the basis function depending of the wanted case. There is three case that should be implemented.

“force”: for the evaluation and computation of the mean force.

“pmf”: for evaluation of the pmf using integration of the mean force

“kernel”: for the evaluation of the kernel.

“corrs”: for the computation of the correlation function.

`compute_corrs_w_noise`(*xva*, *left_op=None*)

Compute correlation between noise and left_op

Parameters

xva
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

`compute_noise`(*xva*, *trunc_kernel=None*, *start_point=0*, *end_point=None*)

From a trajectory get the noise.

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

trunc_kernel
[int] Number of datapoint of the kernel to consider. Can be used to remove unphysical divergence of the kernel or shortten execution time.

`evolve_volterra`(*G0*, *lenTraj*, *method='trapz'*, *trunc_ind=None*)

Evolve in time the integro-differential equation. This assume that the GLE is a linear GLE (i.e. the set of basis function is on the left and right of the equality)

Parameters

G0
[array] Initial value of the correlation

lenTraj
[int] Length of the time evolution

method
[str, default="trapz"] Method that is used to discretize the continuous Volterra equations

trunc_ind: int, default= self.trunc_ind
Truncate the length of the memory to this value

`flux_from_volterra`(*corrs_force*, *corrs_kernel=None*, *force_coeff=None*, *kernel=None*, *method='trapz'*, *trunc_ind=None*)

From a solution of the Volterra equation, compute the flux term. That allow to compute decomposition of the flux

`force_eval`(*x*, *coeffs=None*)

Evaluate the force at given points x. If coeffs is given, use provided coefficients instead of the force

`inv_mass_eval`(*x*, *coeffs=None*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`kernel_eval`(*x*, *coeffs_ker=None*)

Evaluate the kernel at given points x. If coeffs_ker is given, use provided coefficients instead of the kernel

`laplace_transform_kernel`(*s_start=0.0*, *s_end=None*, *n_points=None*)

Compute the Laplace transform of the kernel matrix

`classmethod load_model`(*basis*, *coeffs*, ***kwargs*)

Create a model from a save

`pmf_eval`(*x*, *coeffs=None*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`pmf_num_int_eval`(*x*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This take into accound the position dependent mass, but the integration is numeric

save_model()

Return DataSet version of the model than can be save to file

Examples using VolterraBasis.Pos_gle_const_kernel

- *Prony Series Estimation*
- *Memory Kernel fit*
- *Memory Kernel Estimation with the usual GLE*
- *GLE Integration*

2.3.5 VolterraBasis.Pos_gle_overdamped

class VolterraBasis.**Pos_gle_overdamped**(**args*, *L_obs='v'*, *rank_projection=False*, ***kwargs*)

Extraction of position dependent memory kernel for overdamped dynamics.

Create an instance of the Pos_gle class.

Parameters

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

L_obs: str, default= "a"
Name of the column containing the time derivative of the observable

__init__(**args*, *L_obs='v'*, *rank_projection=False*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

L_obs: str, default= "a"
Name of the column containing the time derivative of the observable

basis_vector(*xva*, *compute_for='corrs'*)

From one trajectory compute the basis element. This is the main method that should be implemented by children class. It take as argument a trajectory and should return the value of the basis function depending of the wanted case. There is three case that should be implemented.

"force": for the evaluation and computation of the mean force.

"pmf": for evaluation of the pmf using integration of the mean force "kernel": for the evaluation of the kernel.

"corrs": for the computation of the correlation function.

`compute_corrs_w_noise`(*xva*, *left_op=None*)

Compute correlation between noise and left_op

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

`compute_noise`(*xva*, *trunc_kernel=None*, *start_point=0*, *end_point=None*)

From a trajectory get the noise.

Parameters

xva
[xarray dataset (['time', 'x', 'v', 'a']) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

trunc_kernel
[int] Number of datapoint of the kernel to consider. Can be used to remove unphysical divergence of the kernel or shortten execution time.

`evolve_volterra`(*G0*, *lenTraj*, *method='trapz'*, *trunc_ind=None*)

Evolve in time the integro-differential equation. This assume that the GLE is a linear GLE (i.e. the set of basis function is on the left and right of the equality)

Parameters

G0
[array] Initial value of the correlation

lenTraj
[int] Length of the time evolution

method
[str, default="trapz"] Method that is used to discretize the continuous Volterra equations

trunc_ind: int, default= self.trunc_ind
Truncate the length of the memory to this value

`flux_from_volterra`(*corrs_force*, *corrs_kernel=None*, *force_coeff=None*, *kernel=None*, *method='trapz'*, *trunc_ind=None*)

From a solution of the Volterra equation, compute the flux term. That allow to compute decomposition of the flux

`force_eval`(*x*, *coeffs=None*)

Evaluate the force at given points x. If coeffs is given, use provided coefficients instead of the force

`inv_mass_eval`(*x*, *coeffs=None*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`kernel_eval`(*x*, *coeffs_ker=None*)

Evaluate the kernel at given points x. If coeffs_ker is given, use provided coefficients instead of the kernel

laplace_transform_kernel(*s_start=0.0*, *s_end=None*, *n_points=None*)

Compute the Laplace transform of the kernel matrix

classmethod load_model(*basis*, *coeffs*, ***kwargs*)

Create a model from a save

pmf_eval(*x*, *coeffs=None*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

pmf_num_int_eval(*x*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This take into accound the position dependent mass, but the integration is numeric

save_model()

Return DataSet version of the model than can be save to file

Examples using `VolterraBasis.Pos_gle_overdamped`

- *Memory Kernel Estimation with the usual GLE*
- *Generalized Fokker Planck equation*
- *Generalized Fokker Planck equation in underdamped case*

2.3.6 `VolterraBasis`.Pos_gle_hybrid

class VolterraBasis.**Pos_gle_hybrid**(**args*, ***kwargs*)

Implement the hybrid projector of arXiv:2202.01922

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`__init__`(**args*, ***kwargs*)

Create an instance of the Pos_gle class.

Parameters

xva_arg
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) or list of datasets.] Use compute_va() or see its output for format details. The timeseries to analyze. It should be either a xarray timeseries or a listlike collection of them.

basis
[scikit-learn transformer to get the element of the basis] This class should implement, basis() and deriv() function and deal with periodicity of the data. If a fit() method is defined, it will be called at initialization

saveall
[bool, default=True] Whether to save all output functions.

prefix
[str] Prefix for the saved output functions.

verbose
[bool, default=True] Set verbosity.

kT
[float, default=2.494] Numerical value for kT.

trunc
[float, default=1.0] Truncate all correlation functions and the memory kernel after this time value.

`basis_vector`(*xva*, *compute_for='corrs'*)

From one trajectory compute the basis element. This is the main method that should be implemented by children class. It take as argument a trajectory and should return the value of the basis function depending of the wanted case. There is three case that should be implemented.

“force”: for the evaluation and computation of the mean force.

“pmf”: for evaluation of the pmf using integration of the mean force

“kernel”: for the evaluation of the kernel.

“corrs”: for the computation of the correlation function.

`compute_corrs_w_noise`(*xva*, *left_op=None*)

Compute correlation between noise and left_op

Parameters

xva
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

`compute_noise`(*xva*, *trunc_kernel=None*, *start_point=0*, *end_point=None*)

From a trajectory get the noise.

Parameters

xva
[xarray dataset ([‘time’, ‘x’, ‘v’, ‘a’]) .] Use compute_va() or see its output for format details. Input trajectory to compute noise.

trunc_kernel
[int] Number of datapoint of the kernel to consider. Can be used to remove unphysical divergence of the kernel or shortten execution time.

`evolve_volterra`(*G0*, *lenTraj*, *method='trapz'*, *trunc_ind=None*)

Evolve in time the integro-differential equation. This assume that the GLE is a linear GLE (i.e. the set of basis function is on the left and right of the equality)

Parameters

G0
[array] Initial value of the correlation

lenTraj
[int] Length of the time evolution

method
[str, default="trapz"] Method that is used to discretize the continuous Volterra equations

trunc_ind: int, default= self.trunc_ind
Truncate the length of the memory to this value

`flux_from_volterra`(*corrs_force*, *corrs_kernel=None*, *force_coeff=None*, *kernel=None*, *method='trapz'*, *trunc_ind=None*)

From a solution of the Volterra equation, compute the flux term. That allow to compute decomposition of the flux

`force_eval`(*x*, *coeffs=None*)

Evaluate the force at given points x. If coeffs is given, use provided coefficients instead of the force

`get_const_kernel_part`()

Return the position independent part of the kernel

`inv_mass_eval`(*x*, *coeffs=None*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`kernel_eval`(*x*)

Evaluate the position dependant part of the kernel at given points x

`laplace_transform_kernel`(*s_start=0.0*, *s_end=None*, *n_points=None*)

Compute the Laplace transform of the kernel matrix

`classmethod load_model`(*basis*, *coeffs*, ***kwargs*)

Create a model from a save

`pmf_eval`(*x*, *coeffs=None*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This assume that the effective mass is independent of the position. If coeffs is given, use provided coefficients instead of the force coefficients.

`pmf_num_int_eval`(*x*, *kT=1.0*, *set_zero=True*)

Compute free energy via integration of the mean force at points x. This take into accound the position dependent mass, but the integration is numeric

`save_model`()

Return DataSet version of the model than can be save to file

2.4 Basis Features

VolterraBasis.basis.LinearFeatures([to_center])	Linear function
VolterraBasis.basis.PolynomialFeatures([...])	Wrapper for numpy polynomial series.
VolterraBasis.basis.FourierFeatures([order, ...])	Fourier series.
VolterraBasis.basis.BSplineFeatures([...])	Bsplines features class
VolterraBasis.basis.FEMScalarFeatures(basis)	Finite elements features for scalar basis
VolterraBasis.basis.SmoothIndicatorFeatures(...)	Indicator function with smooth boundary
VolterraBasis.basis.SplineFctFeatures(knots, ...)	A single basis function that is given from splines fit of data
VolterraBasis.basis.FeaturesCombiner(*basis)	Allow to combine features to build composite basis
VolterraBasis.basis.TensorialBasis2D(b1[, b2])	Combine two 1D basis to get a 2D basis

2.4.1 VolterraBasis.basis.LinearFeatures

class VolterraBasis.basis.**LinearFeatures**(*to_center=False*)

Linear function

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

set_output(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer

- *"pandas"*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.LinearFeatures`

- *Functional basis set*

2.4.2 `VolterraBasis.basis.`PolynomialFeatures

class `VolterraBasis.basis.`**PolynomialFeatures**(*deg=1*, *polynom=<class 'numpy.polynomial.polynomial.Polynomial'>*, *remove_const=True*)

Wrapper for numpy polynomial series.

Providing a numpy polynomial class via polynom keyword allow to change polynomial type.

`__init__`(*deg=1*, *polynom=<class 'numpy.polynomial.polynomial.Polynomial'>*, *remove_const=True*)

Providing a numpy polynomial class via polynom keyword allow to change polynomial type.

`fit_transform`(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`set_output`(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output

- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.PolynomialFeatures`

- *Functional basis set*
- *Kernel Estimation for 2D observable*

2.4.3 `VolterraBasis.basis.FourierFeatures`

`class VolterraBasis.basis.FourierFeatures`(*order=1*, *freq=1.0*, *remove_const=True*)
Fourier series.

Parameters

order
[int] Order of the Fourier series

freq: float
Base frequency

`__init__`(*order=1*, *freq=1.0*, *remove_const=True*)

Parameters

order
[int] Order of the Fourier series

freq: float
Base frequency

`fit_transform`(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

set_output(**, transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.FourierFeatures`

- *Functional basis set*

2.4.4 `VolterraBasis.basis.BSplineFeatures`

class VolterraBasis.basis.**BSplineFeatures**(*n_knots=5, k=3, periodic=False, remove_const=True*)

Bsplines features class

Parameters

n_knots
[int] Number of knots to use

k
[int] Degree of the splines

periodic: bool
Whatever to use periodic splines or not

__init__(*n_knots=5, k=3, periodic=False, remove_const=True*)

Parameters

n_knots
[int] Number of knots to use

k
[int] Degree of the splines

periodic: bool
Whatever to use periodic splines or not

fit_transform(*X, y=None, **fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`set_output`(*, *transform=None*)
Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.BSplineFeatures`

- *Functional basis set*
- *Checking solution of volterra equation*
- *Method comparaison*
- *Prony Series Estimation*
- *Memory Kernel fit*
- *Memory Kernel Estimation*
- *Memory Kernel Estimation with the usual GLE*
- *Generalized Fokker Planck equation*
- *GLE Integration*

2.4.5 VolterraBasis.basis.FEMScalarFeatures

class VolterraBasis.basis.**FEMScalarFeatures**(*basis*)

Finite elements features for scalar basis

Wrapper to finite element basis from scikit-fem Parameters ———- basis: skfem basis

A finite element basis. Should be a scalar basis (H1 or global element)

__init__(*basis*)

Wrapper to finite element basis from scikit-fem Parameters ———- basis: skfem basis

A finite element basis. Should be a scalar basis (H1 or global element)

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

set_output(***, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.FEMScalarFeatures`

- *Memory Kernel Estimation with the usual GLE*

2.4.6 `VolterraBasis.basis`.SmoothIndicatorFeatures

class VolterraBasis.basis.**SmoothIndicatorFeatures**(*states_boundary*, *boundary_type='tricube'*, *periodic=False*)

Indicator function with smooth boundary

Parameters

states_boundary
[list] Number of knots to use

boundary_type
[str or callable] Function to use for the interpolation between zeros and one value If this is a callabe function, first argument is between 0-> 1 and 1 -> 0 and second one is the order of the derivative

periodic: bool
Whatever to use periodic indicator function. If yes, the last indicator will sbe the same function than the first one

__init__(*states_boundary*, *boundary_type='tricube'*, *periodic=False*)

Parameters

states_boundary
[list] Number of knots to use

boundary_type
[str or callable] Function to use for the interpolation between zeros and one value If this is a callabe function, first argument is between 0-> 1 and 1 -> 0 and second one is the order of the derivative

periodic: bool
Whatever to use periodic indicator function. If yes, the last indicator will sbe the same function than the first one

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

set_output(**, transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{"default", "pandas"}, default=None] Configure output of *transform* and *fit_transform*.

- *"default"*: Default output format of a transformer
- *"pandas"*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.SmoothIndicatorFeatures`

- *Generalized Fokker Planck equation*
- *Generalized Fokker Planck equation in underdamped case*

2.4.7 `VolterraBasis.basis.SplineFctFeatures`

class VolterraBasis.basis.**SplineFctFeatures**(*knots, coeffs, k=3, periodic=False*)

A single basis function that is given from splines fit of data

__init__(*knots, coeffs, k=3, periodic=False*)

fit_transform(*X, y=None, **fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

set_output(**, transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{“default”, “pandas”}, default=None] Configure output of *transform* and *fit_transform*.

- *“default”*: Default output format of a transformer
- *“pandas”*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

Examples using `VolterraBasis.basis.SplineFctFeatures`

- *Functional basis set*

2.4.8 `VolterraBasis.basis.FeaturesCombiner`

`class VolterraBasis.basis.`**`FeaturesCombiner`**`(`**basis*`)`

Allow to combine features to build composite basis

`__init__``(`**basis*`)`

`fit_transform``(`*X*, *y=None*, ***fit_params*`)`

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

`set_output``(`***, *transform=None*`)`

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{“default”, “pandas”}, default=None] Configure output of *transform* and *fit_transform*.

- *“default”*: Default output format of a transformer
- *“pandas”*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

2.4.9 VolterraBasis.basis.TensorialBasis2D

class VolterraBasis.basis.**TensorialBasis2D**(*b1*, *b2=None*)

Combine two 1D basis to get a 2D basis

Take two of basis

__init__(*b1*, *b2=None*)

Take two of basis

comb_indices(*i*, *j*)

Get index k of the (i,j) element of the basis

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

X
[array-like of shape (n_samples, n_features)] Input samples.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params**
[dict] Additional fit parameters.

Returns

X_new
[ndarray array of shape (n_samples, n_features_new)] Transformed array.

set_output(*, *transform=None*)

Set output container.

See Introducing the set_output API for an example on how to use the API.

Parameters

transform
[{“default”, “pandas”}, default=None] Configure output of *transform* and *fit_transform*.

- *“default”*: Default output format of a transformer
- *“pandas”*: DataFrame output
- *None*: Transform configuration is unchanged

Returns

self
[estimator instance] Estimator instance.

`split_index(k)`

Get (i,j) decomposition of the keme element of the basis

Examples using `VolterraBasis.basis.TensorialBasis2D`

- *Kernel Estimation for 2D observable*
- *Generalized Fokker Planck equation in underdamped case*

CHAPTER

THREE

GENERAL EXAMPLES

Introductory examples.

3.1 Functional basis set

In this example, we present a subset of implemented functional basis set.

```python
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis.basis as bf

from scipy.interpolate import splrep

x_range = np.linspace(-2, 2, 30).reshape(-1, 1)

t, c, k = splrep(x_range, x_range ** 4 - 2 * x_range ** 2 + 0.5 * x_range)

basis_set = {"Linear": bf.LinearFeatures(), "Polynom": bf.PolynomialFeatures(3), "Hermite Polynom": bf.PolynomialFeatures(3, np.polynomial.Hermite), "Fourier": bf.FourierFeatures(order=2, freq=1.0), "B Splines": bf.BSplineFeatures(6, k=3), "Splines Fct": bf.SplineFctFeatures(t, c, k)}
for key, basis in basis_set.items():
    basis.fit(x_range)

fig_kernel, axs = plt.subplots(2, 3)
m = 0
for key, basis in basis_set.items():
    axs[m // 3][m % 3].set_title(key)
    axs[m // 3][m % 3].set_xlabel("$x$")
    axs[m // 3][m % 3].set_ylabel("$h_k(x)$")
    axs[m // 3][m % 3].grid()
    y = basis.basis(x_range)

    for n in range(y.shape[1]):
        axs[m // 3][m % 3].plot(x_range[:, 0], y[:, n])
    m += 1
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.2 Memory Kernel Estimation with the usual GLE

How to run memory kernel estimation

```
import numpy as np
import dask.array as da

#
import VolterraBasis as vb
import VolterraBasis.basis as bf

import skfem

trj = np.loadtxt("example_lj.trj")
vertices, tri = bf.centroid_driven_mesh(trj[:, 1:3], bins=25)

m = skfem.MeshTri(vertices.T, tri.T)
e = skfem.ElementTriP1()  # skfem.ElementTriRT0()  #
basis_fem = skfem.CellBasis(m, e)

xva_list = []
# trj = da.from_array(trj, chunks=(100, 2))
xf = vb.xframe(trj[:, 1:3], trj[:, 0] - trj[0, 0])
xvaf = vb.compute_va(xf)
xva_list.append(xvaf)

print("Set up traj")

estimator = vb.Estimator_gle(xva_list, vb.Pos_gle_overdamped, bf.FEMScalarFeatures(basis_
↪fem), trunc=1, saveall=False, verbose=False)
model = estimator.compute_mean_force()

xfa = trj[:10, 1:3]

force = model.force_eval(xfa)

#
# model.inv_mass_eval(xfa)
#
estimator.compute_corrs(second_order_method=False)
model = estimator.compute_kernel(method="rect")

time, noise, a, force, mem = model.compute_noise(xvaf)


kernel = model.kernel_eval(xfa)

coeffs = model.save_model()
print(coeffs)
new_model = model.load_model(model.basis, coeffs)

new_kernel = new_model.kernel_eval(xfa)
```

Total running time of the script: (0 minutes 0.000 seconds)

3.3 Checking solution of volterra equation

How to run memory kernel estimation

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)

Nsplines = 10
estimator = vb.Estimator_gle(xva_list, vb.Pos_gle, bf.BSplineFeatures(Nsplines),␣
↪trunc=10, saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.PolynomialFeatures(deg=1), trunc=10, kT=1.0,␣
↪saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.LinearFeatures(), trunc=10, kT=1.0, saveall=False)
print("Dimension of observable", estimator.model.dim_x, estimator.model.rank_projection)
estimator.compute_mean_force()
estimator.compute_corrs()
model = estimator.compute_kernel(method="trapz")

res_diff = estimator.check_volterra_inversion(return_diff=False)

fig_kernel, axs = plt.subplots(1, 1)
# Kernel plot
axs.set_title("Correlation diff")
axs.set_xscale("log")
axs.grid()
estimator.bkdxcorrw.sel(dim_basis=0).squeeze().plot.line("-", x="time_trunc", ax=axs)
axs.plot(np.arange(res_diff.shape[-1]), res_diff[0, 0, :].T, "x")
axs.set_xlabel("$t$")

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.4 Method comparaison

Comparaison of the various algorithm for inversion of the Volterra Integral equation

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)


Nsplines = 10
estimator = vb.Estimator_gle(xva_list, vb.Pos_gle, bf.BSplineFeatures(Nsplines),␣
↪trunc=10, saveall=False)
# mymem = vb.Pos_gle_overdamped(xva_list, bf.BSplineFeatures(Nsplines, remove_
↪const=False), trunc=10, kT=1.0, saveall=False)
estimator.compute_mean_force()
# print(mymem.force_coeff)
estimator.compute_corrs()

fig_kernel, axs = plt.subplots(1, 1)
# # # Kernel plot
axs.set_title("Memory kernel")
axs.set_xscale("log")
axs.set_xlabel("$t$")
axs.set_ylabel("$K(x=2.0,t)$")
axs.set_ylim([-500, 2000])
axs.grid()
# Iterate over method for comparaison
for method in ["rectangular", "midpoint", "midpoint_w_richardson", "trapz", "second_kind_
↪rect", "second_kind_trapz"]:
    model = estimator.compute_kernel(method=method)
    kernel_vb = model.kernel_eval([2.0])
    axs.plot(kernel_vb["time_kernel"], kernel_vb[:, :, 0, 0], "-o", label=method)
axs.legend(loc="best")

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.5 Prony Series Estimation

Memory kernel fitted by a prony series

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)


Nsplines = 10

estimator = vb.Estimator_gle(xva_list, vb.Pos_gle_const_kernel, bf.
↪BSplineFeatures(Nsplines), trunc=10, saveall=False)
# mymem = vb.Pos_gle_overdamped(xva_list, bf.BSplineFeatures(Nsplines, remove_
↪const=False), trunc=10, kT=1.0, saveall=False)
estimator.compute_mean_force()
estimator.compute_corrs()
model = estimator.compute_kernel(method="trapz")
time_ker, kernel = model.kernel["time_kernel"], model.kernel
print("Prony")
A_prony = vb.prony_fit_kernel(time_ker, kernel, thres=None, N_keep=150)
kernel_filtered = vb.prony_inspect_data(kernel[:, 0, 0], thres=None, N_keep=150)
print("Actual number of terms in the series: ", A_prony[0][0][1].shape[0])
fig_kernel, axs = plt.subplots(1, 1)
# # # Kernel plot
axs.set_title("Memory kernel")
axs.set_xscale("log")
axs.set_xlabel("$t$")
axs.set_ylabel("$K(x=2.0,t)$")
axs.grid()


axs.plot(time_ker, kernel[:, 0, 0], "-", label="Memory Kernel")
axs.plot(time_ker, kernel_filtered, "-o", label="Data Filtered")
axs.plot(time_ker, vb.prony_series_kernel_eval(time_ker, A_prony)[:, 0, 0], "-x", label=
↪"Prony fit")
axs.legend(loc="best")

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.6 Memory Kernel fit

Memory kernel fitted by various functionnal forms

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)


Nsplines = 10
# mymem = vb.Pos_gle_const_kernel(xva_list, bf.BSplineFeatures(Nsplines), trunc=10, kT=1.
↪0, saveall=False)
estimator = vb.Estimator_gle(xva_list, vb.Pos_gle_const_kernel, bf.
↪BSplineFeatures(Nsplines), trunc=10, saveall=False)

# mymem = vb.Pos_gle_overdamped(xva_list, bf.BSplineFeatures(Nsplines, remove_
↪const=False), trunc=10, kT=1.0, saveall=False)
estimator.compute_mean_force()
harmonic_coeffs = -1 * estimator.model.force_coeff[0]
# print(mymem.force_coeff)
estimator.compute_corrs()
model = estimator.compute_kernel(method="trapz")
kernel = model.kernel

fig_kernel, axs = plt.subplots(1, 1)
# Kernel plot
axs.set_title("Memory kernel")
# axs.set_xscale("log")
axs.set_xlabel("$t$")
axs.set_ylabel("$K(x=2.0,t)$")
axs.grid()
axs.plot(model.kernel["time_kernel"], kernel[:, 0, 0], "-", label="Memory Kernel")
for type in ["exp", "sech", "gaussian"]:
    print("Fit: " + str(type))
    params = vb.memory_fit(model.kernel["time_kernel"], kernel[:, 0, 0], type=type)
    print(params)
    axs.plot(model.kernel["time_kernel"], vb.memory_fit_eval(model.kernel["time_kernel"],
↪ params), "-x", label="Fit " + str(type))
type = "prony"
print("Fit: " + str(type))
params = vb.memory_fit(model.kernel["time_kernel"], kernel[:, 0, 0], type=type, N_
↪keep=100)
print(params)
```

(continues on next page)

(continued from previous page)

```
axs.plot(model.kernel["time_kernel"], vb.memory_fit_eval(model.kernel["time_kernel"],␣
↪params), "-x", label="Fit " + str(type))
axs.legend(loc="best")

plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.7 Memory Kernel Estimation

How to run memory kernel estimation

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)

Nsplines = 10
estimator = vb.Estimator_gle(xva_list, vb.Pos_gle, bf.BSplineFeatures(Nsplines),␣
↪trunc=10, saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.PolynomialFeatures(deg=1), trunc=10, kT=1.0,␣
↪saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.LinearFeatures(), trunc=10, kT=1.0, saveall=False)
print("Dimension of observable", estimator.model.dim_x)
estimator.compute_mean_force()
estimator.compute_corrs()
model = estimator.compute_kernel(method="trapz")
print(model.force_coeff, model.force_coeff.to_numpy().shape)
kernel = model.kernel_eval([1.5, 2.0, 2.5])
print(kernel)
# To find a correct parametrization of the space
bins = np.histogram_bin_edges(xvaf["x"], bins=15)
xfa = (bins[1:] + bins[:-1]) / 2.0
force = model.force_eval(xfa)


# Compute noise
time_noise, noise_reconstructed, _, _, _ = model.compute_noise(xva_list[0], trunc_
↪kernel=200)


fig_kernel, axs = plt.subplots(1, 3)
```

(continues on next page)

(continued from previous page)

```
# Force plot
axs[0].set_title("Force")
axs[0].set_xlabel("$x$")
axs[0].set_ylabel("$-dU(x)/dx$")
axs[0].grid()
axs[0].plot(xfa, force)
# Kernel plot
axs[1].set_title("Memory kernel")
axs[1].set_xscale("log")
axs[1].grid()
kernel.squeeze().plot.line("-x", x="time_kernel", ax=axs[1])
# axs[1].plot(time, kernel[:, :, 0, 0], "-x")
axs[1].set_xlabel("$t$")
axs[1].set_ylabel("$\\Gamma$")

# Noise plot
axs[2].set_title("Noise")
axs[2].set_xlabel("$t$")
axs[2].set_ylabel("$\\xi_t$")
axs[2].grid()
axs[2].plot(time_noise, noise_reconstructed, "-")
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.8 Memory Kernel Estimation with the usual GLE

How to run memory kernel estimation

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)

Nsplines = 10
estimator = vb.Estimator_gle(xva_list, vb.Pos_gle_const_kernel, bf.
↪BSplineFeatures(Nsplines), trunc=10, saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.PolynomialFeatures(deg=1), trunc=10, kT=1.0,␣
↪saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.LinearFeatures(), trunc=10, kT=1.0, saveall=False)
print("Dimension of observable", estimator.model.dim_x)
estimator.compute_mean_force()
```

(continues on next page)

(continued from previous page)

```
estimator.compute_corrs()
model = estimator.compute_kernel(method="trapz")
time, kernel = model.kernel["time_kernel"], model.kernel[:, 0, 0]
print(time.shape, kernel.shape)
# To find a correct parametrization of the space
bins = np.histogram_bin_edges(xvaf["x"], bins=15)
xfa = (bins[1:] + bins[:-1]) / 2.0
force = model.force_eval(xfa)


time_corrs, noise_corr = estimator.compute_projected_corrs()

vel_var = estimator.compute_effective_mass().eff_mass.values[0, 0]
# Compute noise
time_noise, noise_reconstructed, _, _, _ = model.compute_noise(xva_list[0], trunc_
↪kernel=200)


fig_kernel, axs = plt.subplots(1, 3)
# Force plot
axs[0].set_title("Force")
axs[0].set_xlabel("$x$")
axs[0].set_ylabel("$-dU(x)/dx$")
axs[0].grid()
axs[0].plot(xfa, force)
# Kernel plot
axs[1].set_title("Memory kernel")
axs[1].set_xscale("log")
axs[1].set_xlabel("$t$")
axs[1].set_ylabel("$\\Gamma$")
axs[1].grid()
axs[1].plot(time, kernel, "-")
axs[1].plot(time_corrs, noise_corr[:, 0, 0] * vel_var, "-x")

# Noise plot
axs[2].set_title("Noise")
axs[2].set_xlabel("$t$")
axs[2].set_ylabel("$\\xi_t$")
axs[2].grid()
axs[2].plot(time_noise, noise_reconstructed, "-")
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.9 Kernel Estimation for 2D observable

How to run kernel estimation

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_2d.trj")
xva_list = []
print(trj.shape)
# for i in range(1, trj.shape[1]):
#     xf = vb.xframe(trj[:, i], trj[:, 0])
#     xvaf = vb.compute_va(xf)
#     xva_list.append(xvaf)

xf = vb.xframe(trj[:, (1, 3)], trj[:, 0] - trj[0, 0])
xvaf = vb.compute_va(xf)
xva_list.append(xvaf)


estimator = vb.Estimator_gle(xva_list, vb.Pos_gle, bf.TensorialBasis2D(bf.
↪PolynomialFeatures(deg=1)), trunc=10, saveall=False)
print("Dimension of observable", estimator.model.dim_x)
model = estimator.compute_mean_force()
# print(mymem.force_coeff)
print(model.N_basis_elt, model.N_basis_elt_force, model.N_basis_elt_kernel)
# print(mymem.basis.b1.n_output_features_, mymem.basis.b2.n_output_features_)
estimator.compute_corrs()
model = estimator.compute_kernel(method="trapz")
kernel = model.kernel_eval([[1.5, 1.0], [2.0, 1.5], [2.5, 1.0]])
time = kernel["time_kernel"]
print(time.shape, kernel.shape)
# To find a correct parametrization of the space
bins = np.histogram_bin_edges(xvaf["x"], bins=15)
xfa = (bins[1:] + bins[:-1]) / 2.0
x, y = np.meshgrid(xfa, xfa)
X = np.vstack((x.flatten(), y.flatten())).T
force = model.force_eval(X)

# Compute noise
time_noise, noise_reconstructed, _, _, _ = model.compute_noise(xva_list[0], trunc_
↪kernel=200)


fig_kernel, axs = plt.subplots(1, 3)
# Force plot
axs[0].set_title("Force")
axs[0].set_xlabel("$x$")
axs[0].set_ylabel("$y$")
# axs[0].grid()
```

(continues on next page)

(continued from previous page)

```
axs[0].quiver(x, y, force[:, 0], force[:, 1])
# Kernel plot
axs[1].set_title("Memory kernel")
axs[1].set_xscale("log")
axs[1].set_xlabel("$t$")
axs[1].set_ylabel("$\\Gamma$")
axs[1].grid()
axs[1].plot(time, kernel[:, :, 0, 0], "-x")
axs[1].plot(time, kernel[:, :, 0, 1], "-x")
axs[1].plot(time, kernel[:, :, 1, 0], "-x")
axs[1].plot(time, kernel[:, :, 1, 1], "-x")

# Noise plot
axs[2].set_title("Noise")
axs[2].set_xlabel("$t$")
axs[2].set_ylabel("$\\xi_t$")
axs[2].grid()
axs[2].plot(time_noise, noise_reconstructed, "-")
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.10 Generalized Fokker Planck equation

How to run GFPE estimation

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)
basis_indicator = bf.SmoothIndicatorFeatures([[1.4, 1.5]], "tricube")
basis_indicator = bf.SmoothIndicatorFeatures([[1.0, 1.1], [1.4, 1.5], [1.6, 1.7], [2.0,␣
↪2.1]], "tricube")
basis_splines = bf.BSplineFeatures(10, remove_const=False)

estimator = vb.Estimator_gle(xva_list, vb.Pos_gle_overdamped, basis_indicator, trunc=10,␣
↪saveall=False)
estimator.to_gfpe()

estimator.compute_mean_force()
estimator.compute_corrs()
```

(continues on next page)

(continued from previous page)

```
model = estimator.compute_kernel(method="trapz")


fig_kernel, axs = plt.subplots(1, 1)
# Kernel plot
axs.set_title("Memory kernel")
axs.set_xscale("log")
axs.set_xlabel("$t$")
axs.set_ylabel("$\\Gamma$")
axs.grid()
# axs.plot(model.time, model.kernel[:, 0, 0], "-x")
# axs.plot(model.time, model.kernel[:, 0, 1], "-x")
print(model.kernel.shape, model.kernel.dims)
axs.plot(model.kernel["time_kernel"], model.kernel[:, 2, :], "-x")
axs.plot(model.kernel["time_kernel"], model.kernel[:, :, 2], "-x")


occ = estimator.compute_basis_mean()
print(occ)

time, bkbk = model.evolve_volterra(estimator.bkbkcorrw.isel(time_trunc=0), 500, method=
↪"rect")
print(bkbk.shape)
time, flux = model.flux_from_volterra(bkbk)
#
# fig_pt = plt.figure("Probability of time")
# plt.grid()
# plt.plot(t_new, p_t[:, :], "-x")
# plt.scatter(t_new[-1] * np.ones(model.dim_obs), occ)  # Plot occupations that should␣
↪be long time limit
#
# t_num = np.arange(model.trunc_ind) * (t_new[1] - t_new[0])
# p_t_num = np.einsum("ikj, kl, l->ij", estimator.bkbkcorrw, np.diag(1.0 / occ), p0)
#
# plt.plot(t_num, p_t_num, "--")
#
#
# plt.plot(t_new, np.sum(p_t, axis=1), "-o")
# plt.plot(t_num, np.sum(p_t_num, axis=1), "--o")
#
# fig, ax_anim = plt.subplots()
# ax_anim.grid()
# time_text = ax_anim.text(0.85, 0.95, "0.0", horizontalalignment="left",␣
↪verticalalignment="top", transform=ax_anim.transAxes)
#
# xrange = np.linspace(0.8, 3.0, 150)
# E_eval_unnorm = model.basis_vector(vb.models._convert_input_array_for_
↪evaluation(xrange, 1), compute_for="force")
# norm_E = np.trapz(E_eval_unnorm, x=xrange, axis=0)
# E_eval = E_eval_unnorm @ np.diag(norm_E)
#
#
```

(continues on next page)

(continued from previous page)

```
# proba_val = E_eval @ np.max(p_t[:, :], axis=0)
# dt = t_new[1] - t_new[0]
# print(E_eval.shape, norm_E)
# (ln,) = ax_anim.plot(xrange, proba_val, "-")
#
#
# def update(frame):
#     proba_val = E_eval @ p_t[frame, :]
#     ln.set_data(xrange, proba_val)
#     time_text.set_text("%.3f" % (frame * dt))
#     return (ln, time_text)
#
#
# ani = animation.FuncAnimation(fig, update, frames=np.arange(p_t.shape[0]), blit=True,␣
↪interval=10)
#
#
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.11 Generalized Fokker Planck equation in underdamped case

How to run kernel estimation

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

import VolterraBasis as vb
import VolterraBasis.basis as bf

trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_under = vb.concat_underdamped(xvaf)
    xva_list.append(xva_under)
# print(xva_under.head())
basis_x = bf.SmoothIndicatorFeatures([[1.4, 1.5]], "quartic")
basis_v = bf.SmoothIndicatorFeatures([[-1.1, -1.0], [1.0, 1.1]], "tricube",␣
↪periodic=False)
basis_comb = bf.TensorialBasis2D(basis_x, basis_v)
mymem = vb.Estimator_gle(xva_list, vb.Pos_gle_overdamped, basis_comb, trunc=10,␣
↪saveall=False)
print("Dimension of observable", mymem.model.dim_obs)
mymem.compute_mean_force()

mymem.compute_corrs()
```

(continues on next page)

(continued from previous page)

```
mymem.compute_kernel(method="trapz")
#
#
# fig_kernel, axs = plt.subplots(1, 1)
# # Kernel plot
# axs.set_title("Memory kernel")
# # axs.set_xscale("log")
# axs.set_xlabel("$t$")
# axs.set_ylabel("$\\Gamma$")
# axs.grid()
# axs.plot(mymem.time, np.sum(mymem.kernel, axis=2), "-x")
# axs.plot(mymem.time, mymem.kernel[:, basis_comb.comb_indices(0, 0), :], "-x")
# # axs.plot(mymem.time, mymem.kernel[:, basis_comb.comb_indices(0, 0), basis_comb.comb_
↪indices(0, 0)], "-x")
# # axs.plot(mymem.time, mymem.kernel[:, basis_comb.comb_indices(1, 0), basis_comb.comb_
↪indices(0, 0)], "-x")
# # axs.plot(mymem.time, mymem.kernel[:, basis_comb.comb_indices(0, 0), basis_comb.comb_
↪indices(1, 0)], "-x")
# # axs.plot(mymem.time, mymem.kernel[:, basis_comb.comb_indices(1, 0), basis_comb.comb_
↪indices(1, 0)], "-x")
#
#
# # Survival problem
# # sink_index = basis_comb.comb_indices(1, 1)
# p0 = np.zeros(mymem.dim_obs)
# p0[basis_comb.comb_indices(0, 1)] = 1.0
# t_new, p_t = mymem.solve_gfpe(5000, method="trapz", p0=p0)
# fig_pt = plt.figure("Probability of time")
# plt.grid()
#
# occ = mymem.occupations()
# t_num = np.arange(mymem.trunc_ind) * (t_new[1] - t_new[0])
# p_t_num = np.einsum("ikj, kl, l->ij", mymem.bkdxcorrw, np.diag(1.0 / occ), p0)
# plt.plot(t_num, p_t_num, "--")
#
# plt.plot(t_new, p_t, "-")
#
# plt.plot(t_new, np.sum(p_t, axis=1), "-o")
#
# fig, ax_anim = plt.subplots()
# ax_anim.grid()
# dt = t_new[1] - t_new[0]
# time_text = ax_anim.text(0.05, 1.05, "0.0", horizontalalignment="left",␣
↪verticalalignment="top", transform=ax_anim.transAxes)
#
#
# xrange = np.linspace(0.8, 3.0, 50)
# yrange = np.linspace(-2.0, 2.0, 50)
# # Do mesh
# xx, yy = np.meshgrid(xrange, yrange)
# E_eval = basis_comb.basis(np.column_stack((xx.flatten(), yy.flatten())))
```

(continues on next page)

(continued from previous page)

```
# proba_val = E_eval @ p_t[0, :]
# print(E_eval.shape, proba_val.shape, xx.shape, yy.shape, np.column_stack((xx.flatten(),
↪ yy.flatten())).shape)
# quad = ax_anim.pcolormesh(xx, yy, proba_val.reshape(50, 50), shading="gouraud", cmap=
↪"viridis")
# fig.colorbar(quad)
#
#
# def update(frame):
#     proba_val = E_eval @ p_t[frame, :]
#     quad.set_array(proba_val.reshape(50, 50))
#     time_text.set_text("%.3f" % (frame * dt))
#     return (quad, time_text)
#
#
# ani = animation.FuncAnimation(fig, update, frames=np.arange(p_t.shape[0]), blit=True,␣
↪interval=10)
#
#
# fig_basis, axis_basis = plt.subplots(2, 1)
#
# Ex_basis = basis_x.basis(xrange.reshape(-1, 1))
# print(Ex_basis.shape)
#
# axis_basis[0].plot(xrange, Ex_basis)
#
# Ev_basis = basis_v.basis(yrange.reshape(-1, 1))
# print(Ev_basis.shape)
#
# axis_basis[1].plot(yrange, Ev_basis)
#
# plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

3.12 GLE Integration

How to run integration of the GLE once estimated. Note: Due to time limit on readthedocs, the trajectories here are too short for convergence and figure are quite noisy.

```
import numpy as np
import matplotlib.pyplot as plt

import VolterraBasis as vb
import VolterraBasis.basis as bf


def compute_1d_fe(xva_list):
    """
    Computes the free energy from the trajectoy using a cubic spline
    interpolation.
```

(continues on next page)

(continued from previous page)

```
    Parameters
    ----------

    bins : str, or int, default="auto"
        The number of bins. It is passed to the numpy.histogram routine,
        see its documentation for details.
    hist: bool, default=False
        If False return the free energy else return the histogram
    """

    # # D'abord on obtient les bins
    min_x = np.min([xva["x"].min("time") for xva in xva_list])
    max_x = np.max([xva["x"].max("time") for xva in xva_list])

    n_bins = 50
    x_bins = np.linspace(min_x, max_x, n_bins)
    mean_val = 0
    count_bins = 0
    for xva in xva_list:
        # add v^2 to the list
        ds_groups = xva.assign({"v2": xva["v"] * xva["v"]}).groupby_bins("x", x_bins)
        # print(ds_groups)
        mean_val += ds_groups.sum().fillna(0)
        count_bins += ds_groups.count().fillna(0)
    fehist = (count_bins / count_bins.sum())["x"]
    mean_val = mean_val / count_bins
    pf = fehist.to_numpy()

    xfa = (x_bins[1:] + x_bins[:-1]) / 2.0

    xf = xfa[np.nonzero(pf)]
    fe = -np.log(pf[np.nonzero(pf)])
    fe -= np.min(fe)
    mean_a = mean_val["a"].to_numpy()[np.nonzero(pf)]

    return xf, fe, mean_a


trj = np.loadtxt("example_lj.trj")
xva_list = []
print(trj.shape)
corrs_vv_md = 0.0
for i in range(1, trj.shape[1]):
    xf = vb.xframe(trj[:, i], trj[:, 0] - trj[0, 0])
    xvaf = vb.compute_va(xf)
    xva_list.append(xvaf)
    corrs_vv_md += vb.correlation_fft(xvaf["v"].T) / (trj.shape[1] - 1)
Nsplines = 10

ntrajs = trj.shape[1] - 1
```

(continues on next page)

(continued from previous page)

```
xf_md, fe_md, mean_a_md = compute_1d_fe(xva_list)

estimator = vb.Estimator_gle(xva_list, vb.Pos_gle_const_kernel, bf.
↪BSplineFeatures(Nsplines), trunc=10, saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.PolynomialFeatures(deg=1), trunc=10, kT=1.0,␣
↪saveall=False)
# mymem = vb.Pos_gle(xva_list, bf.LinearFeatures(), trunc=10, kT=1.0, saveall=False)
print("Dimension of observable", estimator.model.dim_x)
estimator.compute_mean_force()
estimator.compute_corrs()
estimator.compute_pos_effective_mass()
model = estimator.compute_kernel(method="trapz")
time, kernel = model.kernel["time_kernel"], model.kernel[:, 0, 0]
force_md = model.force_eval(xf_md)


integrator = vb.Integrator_gle_const_kernel(model)  # np.ones(Nsplines - 1)

xva_new = []
corrs_vv_cg = 0.0
for n in range(ntrajs):
    start = integrator.initial_conditions(xva_list[n])
    xva = integrator.run(40000, start)
    xva = vb.compute_a(xva)
    xva_new.append(xva)
    print(xva)
    corrs_vv_cg += vb.correlation_fft(xva["v"].T) / ntrajs

xf_cg, fe_cg, mean_a_cg = compute_1d_fe(xva_new)


fig_integration, axs = plt.subplots(2, 2)
# New traj plot
axs[0, 0].set_title("Traj")
axs[0, 0].set_xlabel("$t$")
axs[0, 0].set_ylabel("$r(t)$")
axs[0, 0].grid()
for n in range(ntrajs):
    axs[0, 0].plot(xva_new[n]["time"], xva_new[n]["x"], "-")

# Density Plot
axs[0, 1].set_title("Density")
axs[0, 1].set_xlabel("$r$")
axs[0, 1].set_ylabel("PMF")
axs[0, 1].grid()
axs[0, 1].plot(xf_md, fe_md, "-", label="MD")
axs[0, 1].plot(xf_cg, fe_cg, "-", label="CG")

# Force Plot
axs[1, 1].set_title("Force")
axs[1, 1].set_xlabel("$r$")
axs[1, 1].set_ylabel("f(r)")
```

(continues on next page)

(continued from previous page)

```
axs[1, 1].grid()
axs[1, 1].plot(xf_md, force_md, "-", label="MD mean force")
axs[1, 1].plot(xf_cg, mean_a_cg, "-", label="CG")

# Correlation plot
axs[1, 0].set_title("Corrs")
axs[1, 0].set_xscale("log")
axs[1, 0].set_xlabel("$t$")
axs[1, 0].set_ylabel("$\\langle v,v \\rangle$")
axs[1, 0].grid()
axs[1, 0].plot(corrs_vv_md[0, 0, :1000], "-", label="MD")
axs[1, 0].plot(corrs_vv_cg[0, 0, :1000], "-", label="CG")


plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

CHAPTER FOUR

INSTALLATION

Run

```
>>> pip install git+https://github.com/HadrienNU/VolterraBasis.git
```

to install.

CHAPTER

FIVE

GETTING STARTED

To run the code, you should first instanciate the Estimator_gle class

```
>>> mymem =  Estimator_gle(traj_list, vb.Pos_gle, bf.BSplineFeatures(Nsplines))
```

The mandatory arguments are a list of trajectories, the choice of a model and a funcionnal basis.

The list of trajectories should be created through the `VolterraBasis.xframe()` method such as

```
>>> trj = np.loadtxt("example_lj.trj")
>>> xf = vb.xframe(trj[:, 1], trj[:, 0]) # First argument is trajectory, second is time
>>> xvaf = vb.compute_va(xf) #  Compute velocity and acceleration
>>> trajs_list=[xvaf]
```

You should then compute mean force and correlation using

```
>>> mymem.compute_mean_force()
>>> mymem.compute_corrs()
```

CHAPTER

SIX

INVERSION OF VOLTERRA INTEGRAL EQUATIONS

Computation of the memory kernel is obtained using

```
>>> mymem.compute_kernel()
```

Several algorithms for the inversion of the Volterra Integral Equations are available. Please refer to P. Linz, "Numerical methods for Volterra integral equations of the first kind", The Computer Journal 12, 393–397 (1969) for mathematical details.

CHAPTER

SEVEN

FUNCTIONNAL BASIS

The estimation of the memory kernel necessite the choice of a functionnal basis. Functional basis are implemented in `VolterraBasis.basis` that could be imported and initialized as

```
>>> import VolterraBasis.basis as bf
>>> basis=bf.BSplineFeatures(15)
```

Several options are available for the type of basis, please refer to the documentation. Although multidimensionnal trajectories can be analysed, not all functionnal basis are multidimensionnal.

CHAPTER

EIGHT

FORCE AND MEMORY ESTIMATE

Once the mean force and memory have been computed, the value of the force and memory kernel at given position can be computed trought function `VolterraBasis.Pos_gle.force_eval()` and `VolterraBasis.Pos_gle.kernel_eval()`

CHAPTER

NINE

CHOICE OF THE FORM OF THE GLE

Several options are available to choose the form of the GLE:

- *VolterraBasis.Pos_gle* implement the form of the GLE featured in Vroylandt and Monmarché with memory kernel linear in velocity.
- *VolterraBasis.Pos_gle_with_friction* is similar to the previous but don't assume that the instantaneous friction is zero.
- *VolterraBasis.Pos_gle_const_kernel* is the traditionnal GLE with memory kernel linear in velocity and independant of position.
- *VolterraBasis.Pos_gle_no_vel_basis* implement a GLE where the memory kernel has no dependance in velocity.
- *VolterraBasis.Pos_gle_overdamped* compute the memory kernel for an overdamped dynamics.

INDEX

Symbols

B

C

L

P

S

T

X