GLE$_{a}nalysisEMDocumentation$

Hadrien Vroylandt

Feb 21, 2023

DOCUMENTATION

This project is implementation of Expectation-Maximization algorithm for the inference of Generalized Langevin Equations. Please refer to https://arxiv.org/abs/2110.04246 for a detailed description of the algorithm.

CHAPTER

ONE

USER GUIDE: EM ESTIMATOR FOR GENERALIZED LANGEVIN EQUATIONS

1.1 EM Estimator

The central piece of the package is the *GLE_analysisEM.GLE_Estimator*. All estimators in scikit-learn are derived from this class. In more details, this base class enables to set and get parameters of the estimator. It can be imported as:

```
>>> from GLE_analysisEM import GLE_Estimator
```

Once imported, lauching the estimation is as simple as

```
>>> estimator = GLE_Estimator(dim_x=dim_x, dim_h=dim_h, basis=basis)
>>> estimator.fit(X, idx_trajs=idx)
```

Several parameters are available. At least `dim_x` that give the dimension of the system under study and `dim_h` that give the number of hidden dimension to fit should be provided. A functional basis for fitting of the mean force is also required and is explained below.

The trajectories data are provided to the `GLE_analysisEM.fit` function as `X` and `idx` under a format that is explained below.

Once fitted (that can be quite long), the estimated parameters can be obtained as a dictionary

```
>>> estimator.get_coefficients()
```

1.2 Functional basis

In GLE_analysisEM, the mean force term is fitted on a functional basis that should be provided to *GLE_analysisEM.GLE_Estimator*. Functional basis are implemented in *GLE_analysisEM.GLE_BasisTransform* that could be imported and initialized as

```
>>> from GLE_analysisEM import GLE_BasisTransform
>>> basis = GLE_BasisTransform(basis_type="linear")
```

Several options are available for the type of basis, please refer to the documentation of *GLE_analysisEM.GLE_BasisTransform*. Some type required the basis to be fitted from the data a priori using

```
>>> basis = GLE_BasisTransform(basis_type="free_energy").fit(X)
```

1.3 Trajectory format

The trajectories should be pass as a single array of the shape (Ndatas x dim). If multiple trajectories shoud be provided, all trajectories should be stacked together and another array that contain the indices to split the array should be provided (passed to numpy.split). Helpers function that load trajectories from files (one file per trajectories) are provided in `GLE_analysisEM.data_loaders`.

1.4 Generation of new trajectories

Once the estimator is converged, it can be used to generate new trajectories with `GLE_analysisEM.sample`.

1.5 Predict value of the hidden variables

An estimation of the value of the hidden variable can be obtained with `GLE_analysisEM.predict`

CHAPTER

TWO

GLE_ANALYSISEM API

This section contains API documentation for the most commonly used interfaces of the library.

2.1 Estimator

GLE_Estimator([dim_x, dim_h, tol, max_iter, ...])	A GLE estimator based on Expectation-Maximation algorithm.

2.1.1 GLE_analysisEM.GLE_Estimator

class GLE_analysisEM.**GLE_Estimator**(*dim_x=1*, *dim_h=1*, *tol=1e-05*, *max_iter=100*, *OptimizeForce=True*, *OptimizeDiffusion=True*, *init_params='random'*, *model='euler'*, *basis=GLE_BasisTransform()*, *A_init=None*, *C_init=None*, *force_init=None*, *mu_init=None*, *sig_init=None*, *n_init=1*, *random_state=None*, *warm_start=False*, *no_stop=False*, *verbose=0*, *verbose_interval=10*, *multiprocessing=1*)

A GLE estimator based on Expectation-Maximation algorithm.

Parameters

dim_x
[int, default=1] The number of visible dimensions

dim_h
[int, default=1] The number of hidden dimensions

tol
[float, defaults to 1e-5.] The convergence threshold. EM iterations will stop when the lower bound average gain is below this threshold.

max_iter: int, default=100
The maximum number of EM iterations

OptimizeForce: bool, default=True
Optimize or not the force coefficients, to be set to False if the force or the potential have been externally determined

OptimizeDiffusion: bool, default=True
Optimize or not the diffusion coefficients

init_params
[{'user','random','markov'}, defaults to 'random'.] The method used to initialize the fitting coefficients. Must be one of:

```
'user' : coefficients are initialized at values provided by the user
'random' : coefficients are initialized randomly.
'markov' : coefficients are initialized with Markovian estimation of␣
↪the visible part
```

model
[{}, default to 'euler'.] Choice of time discretized model to be fitted. For now only euler model is implemented

basis: a scikit-learn Transformer class, default to linear basis.
Transformer to get value of the basis function

A_init, C_init, force_init, mu_init, sig_init: array, optional
The user-provided initial coefficients, defaults to None. If it None, coefficients are initialized using the *init_params* method.

n_init
[int, defaults to 1.] The number of initializations to perform. The best results are kept.

no_stop: bool, default to False
Does not stop the iterations if the algorithm have converged

warm_start
[bool, default to False.] If 'warm_start' is True, the solution of the last fitting is used as initialization for the next call of fit(). This can speed up convergence when fit is called several times on similar problems.

random_state
[int, RandomState instance or None, optional (default=None)] Controls the random seed given to the method chosen to initialize the parameters (see *init_params*). In addition, it controls the generation of random samples from the fitted distribution (see the method *sample*). Pass an int for reproducible output across multiple function calls. See Glossary.

verbose
[int, default to 0.] Enable verbose output. If 1 then it prints the current initialization and each iteration step. If greater than 1 then it prints also the log probability and the time needed for each step.

verbose_interval
[int, default to 10.] Number of iteration done before the next print.

multiprocessing: int, default to 1
Number of process to use for E step

__init__(*dim_x=1*, *dim_h=1*, *tol=1e-05*, *max_iter=100*, *OptimizeForce=True*, *OptimizeDiffusion=True*, *init_params='random'*, *model='euler'*, *basis=GLE_BasisTransform()*, *A_init=None*, *C_init=None*, *force_init=None*, *mu_init=None*, *sig_init=None*, *n_init=1*, *random_state=None*, *warm_start=False*, *no_stop=False*, *verbose=0*, *verbose_interval=10*, *multiprocessing=1*)

Examples using GLE_analysisEM.GLE_Estimator

- *Running GLE Estimator*
- *Likelihood Evolution*
- *M step*
- *Parameters estimation from Markovian*
- *Parameters estimation*

2.2 Basis Features

GLE_BasisTransform([basis_type, transformer])	A transformer that give values of the basis along the trajectories.

2.2.1 GLE_analysisEM.GLE_BasisTransform

class GLE_analysisEM.**GLE_BasisTransform**(*basis_type='linear'*, *transformer=None*, ***kwargs*)

A transformer that give values of the basis along the trajectories.

Parameters

dim_x
[int, default=1] The number of visible dimensions.

basis_type
[str, default= "linear"] Give the type of basis projection Must be one of:

```
"linear" : Linear basis.
"polynomial" : Polynomial basis.
"bins" : Bins basis.
"bsplines" : BSplines basis.
"free_energy" : Use histogram estimation of the free energy as␣
↪unique basis function.
"free_energy_kde" : Use Kernel Density estimation of the free energy␣
↪as unique basis function.
"custom": custom basis, you should pass a Transformer class
```

__init__(*basis_type='linear'*, *transformer=None*, ***kwargs*)

Examples using GLE_analysisEM.GLE_BasisTransform

- *Running GLE Estimator*
- *Likelihood Evolution*
- *M step*
- *Parameters estimation from Markovian*
- *Parameters estimation*

2.3 Helper functions

GLE_analysisEM.datas_loaders.loadData(paths, ...)	Loads trajectories from a list of file
GLE_analysisEM.post_processing.memory_kernel(...)	Return the value of the estimated memory kernel

2.3.1 GLE_analysisEM.datas_loaders.loadData

GLE_analysisEM.datas_loaders.**loadData**(*paths*, *dim_x*, *maxlenght=None*)

Loads trajectories from a list of file

Parameters

paths
[list of str] List of paths to trajectory files, one trajectory per file The file are loaded with `numpy.loadtxt` and should have one column by dimension and one data point per line.

dim_x
[int] Number of column to take from each file

2.3.2 GLE_analysisEM.post_processing.memory_kernel

GLE_analysisEM.post_processing.**memory_kernel**(*ntimes*, *dt*, *coeffs*, *dim_x*, *noDirac=False*)

Return the value of the estimated memory kernel

Parameters

ntimes,dt: Number of timestep and timestep
coeffs
[Coefficients for diffusion and friction]

dim_x: Dimension of visible variables
noDirac: Remove the dirac at time zero

Returns

timespan
[array-like, shape (n_samples,)] Array of time to evaluate memory kernel

kernel_evaluated
[array-like, shape (n_samples, dim_x,dim_x)] Array of values of the kernel at time provided

Examples using GLE_analysisEM.post_processing.memory_kernel

- *Parameters estimation*

CHAPTER

THREE

GENERAL EXAMPLES

Introductory examples.

3.1 Running GLE Estimator

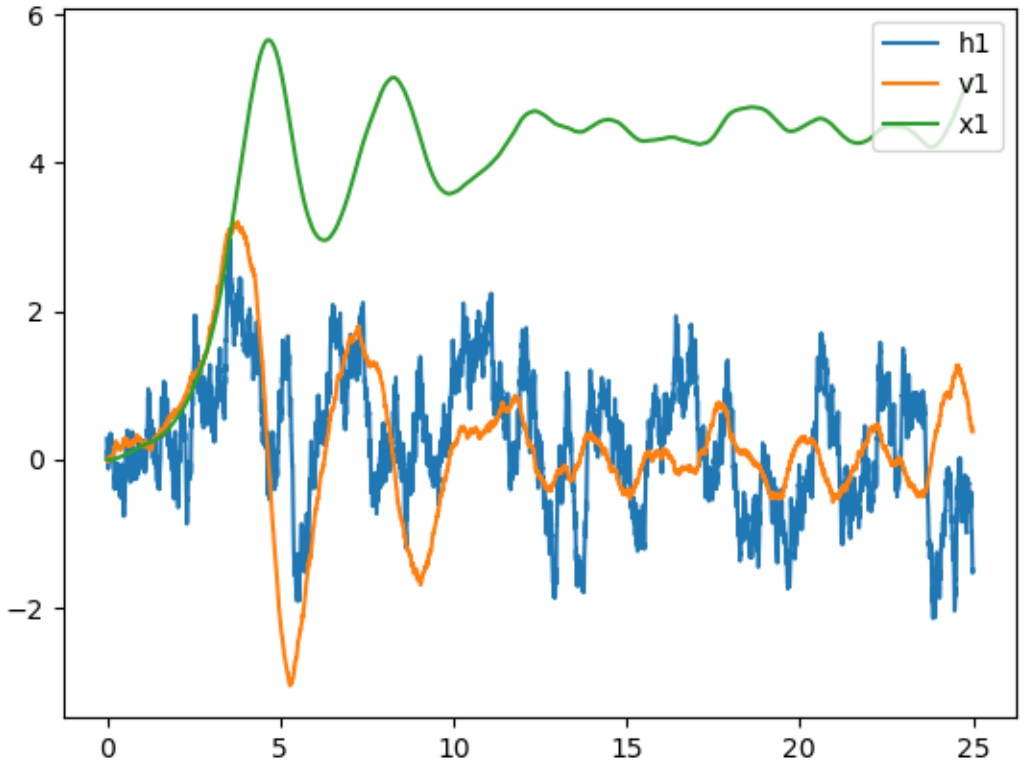
An example of how to run estimation *GLE_analysisEM.GLE_Estimator*

```
import numpy as np
from GLE_analysisEM import GLE_Estimator, GLE_BasisTransform

dim_x = 1
dim_h = 1
random_state = 42
force = -np.identity(dim_x)

ntrajs = 25

pot_gen = GLE_BasisTransform(basis_type="linear")

# Trajectory generation
generator = GLE_Estimator(verbose=2, dim_x=dim_x, dim_h=dim_h, force_init=force, init_
↪params="random", basis=pot_gen, random_state=random_state)
X, idx, Xh = generator.sample(n_samples=5000, n_trajs=ntrajs, x0=0.0, v0=0.0)
print("Real parameters", generator.get_coefficients())


# Estimation of parameters
basis = GLE_BasisTransform(basis_type="free_energy")  # Choice of basis for mean force
estimator = GLE_Estimator(dim_x=dim_x, dim_h=dim_h, basis=basis, max_iter=100, n_init=1,␣
↪random_state=random_state + 1, verbose=1, verbose_interval=50)
estimator.fit(X, idx_trajs=idx)
print("Estimated parameters", estimator.get_coefficients())
```

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

3.2 Generating GLE Samples

Generation of sample trajectory via `GLE_analysisEM.GLE_Estimator.sample`




```
{'A': array([[ 0.02002006,  1.        ],
       [-1.        ,  1.50130236]]), 'C': array([[1.00000000e+00, 1.07699575e-16],
       [1.21168839e-16, 1.00000000e+00]]), 'force': array([[1.]]), 'μ_0': array([0.]), '_
↪0': array([[1.]]), 'SST': array([[0.0002002 , 0.        ],
       [0.        , 0.01501302]]), 'dt': 0.005, 'basis': {'basis_type': 'linear',
↪'transformer__accept_sparse': False, 'transformer__check_inverse': True, 'transformer__
↪feature_names_out': None, 'transformer__func': <function dV at 0x7ff7d6de5670>,
↪'transformer__inv_kw_args': None, 'transformer__inverse_func': None, 'transformer__kw_
↪args': None, 'transformer__validate': False, 'transformer': FunctionTransformer(func=
↪<function dV at 0x7ff7d6de5670>), 'dim_x': 1, 'nb_basis_elt': 1}}
```

```
import numpy as np
from matplotlib import pyplot as plt
from GLE_analysisEM import GLE_Estimator, GLE_BasisTransform
```

(continues on next page)

(continued from previous page)

```
from sklearn.preprocessing import FunctionTransformer


a = 0.025
b = 1.0


def dV(X):
    """
    Compute the force field
    """
    return -4 * a * np.power(X, 3) + 2 * b * X


dim_x = 1
dim_h = 1
model = "euler"
force = np.identity(dim_x)

basis = GLE_BasisTransform(transformer=FunctionTransformer(dV))
generator = GLE_Estimator(verbose=1, dim_x=dim_x, dim_h=dim_h, model=model, basis=basis,␣
↪force_init=force, init_params="random", multiprocessing=4)
X, idx, h = generator.sample(n_samples=5000, x0=0.0, v0=0.0)
print(generator.get_coefficients())
for n in range(dim_h):
    plt.plot(X[:, 0], h[:, n], label="h{}".format(n + 1))

for n in range(dim_x):
    plt.plot(X[:, 0], X[:, n * 2 + 2], label="v{}".format(n + 1))
    plt.plot(X[:, 0], X[:, n * 2 + 1], label="x{}".format(n + 1))


plt.legend(loc="upper right")
plt.show()
```

Total running time of the script: (0 minutes 0.496 seconds)

3.3 Likelihood Evolution

Evolution of the log_likelihood along fit for train and test trajectories

```
import numpy as np
import pandas as pd
from GLE_analysisEM import GLE_Estimator, GLE_BasisTransform

from matplotlib import pyplot as plt

# Printing options
pd.set_option("display.max_rows", None)
pd.set_option("display.max_columns", None)
```

(continues on next page)

(continued from previous page)

```
pd.set_option("display.width", None)
pd.set_option("display.max_colwidth", None)

dim_x = 1
dim_h = 1
random_state = 42
force = -np.identity(dim_x)
max_iter = 10
N_big_steps = 150
ntrajs = 25

pot_gen = GLE_BasisTransform(basis_type="linear")

# Trajectory generation
generator = GLE_Estimator(verbose=2, dim_x=dim_x, dim_h=dim_h, force_init=force, init_
↪params="random", basis=pot_gen, random_state=random_state)
X, idx, Xh = generator.sample(n_samples=5000, n_trajs=ntrajs, x0=0.0, v0=0.0)
X_val, idx_val, Xh_val = generator.sample(n_samples=5000, n_trajs=10, x0=0.0, v0=0.0)
print("Real parameters", generator.get_coefficients())

initial_ll = generator.score(X, idx_trajs=idx)
initial_ll_val = generator.score(X_val, idx_trajs=idx_val)
print("Initial ll", initial_ll, initial_ll_val)

basis = GLE_BasisTransform(basis_type="linear")
# Trajectory estimation
estimator = GLE_Estimator(init_params="random", dim_x=dim_x, dim_h=dim_h, basis=basis,␣
↪OptimizeDiffusion=True, no_stop=True, max_iter=max_iter, n_init=1, random_state=None,␣
↪verbose=0, multiprocessing=8)
# We set some initial conditions, check for stability
# estimator.set_init_coeffs(generator.get_coefficients())

logL_train = np.empty((N_big_steps * max_iter,))
logL_val = np.empty((N_big_steps * max_iter,))
for i in range(N_big_steps):
    print("Step {}".format(i))
    estimator.set_params(warm_start=True)
    estimator.fit(X, idx_trajs=idx)
    estimator.get_coefficients()
    logL_train[i * max_iter : (i + 1) * max_iter] = estimator.logL[0, :]
    logL_val[i * max_iter : (i + 1) * max_iter] = estimator.score(X_val, idx_trajs=idx_
↪val)

print(estimator.get_coefficients())


plt.plot(logL_train[1:], label="Log L train")
plt.plot(logL_val[1:], label="Log L validation")
plt.plot([initial_ll] * N_big_steps * max_iter, label="Initial ll train")
plt.plot([initial_ll_val] * N_big_steps * max_iter, label="Initial ll validation")
plt.legend(loc="upper right")
plt.show()
```

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

3.4 M step

Inner working of the M step, maximum likelihood estimation of the coefficients *GLE_analysisEM.GLE_Estimator*

```
import numpy as np
import pandas as pd

# from matplotlib import pyplot as plt
from GLE_analysisEM import GLE_Estimator, GLE_BasisTransform, sufficient_stats,␣
↪sufficient_stats_hidden
from sklearn.preprocessing import FunctionTransformer

# from GLE_analysisEM.utils import loadTestDatas_est


# Printing options
pd.set_option("display.max_rows", None)
pd.set_option("display.max_columns", None)
pd.set_option("display.width", None)
pd.set_option("display.max_colwidth", None)

# dim_x = 1
# dim_h = 1
# model = "aboba"
# force = -np.identity(dim_x)
#
a = 0.025
b = 1.0


def dV(X):
    """
    Compute the force field
    """
    return -4 * a * np.power(X, 3) + 2 * b * X


# generator = GLE_Estimator(verbose=1, dim_x=dim_x, dim_h=dim_h, EnforceFDT=True, force_
↪init=force, init_params="random", model=model, random_state=42)
# X, idx, Xh = generator.sample(n_samples=1000, n_trajs=500, x0=0.0, v0=0.0, basis=basis)

dim_x = 1
dim_h = 1
random_state = 42
force = -np.identity(dim_x)

A = np.array([[5e-5, 1.0], [-1.0, 0.5]])
C = np.identity(dim_x + dim_h)
# ------ Generation ------#
pot_gen = GLE_BasisTransform(basis_type="linear")
```

(continues on next page)

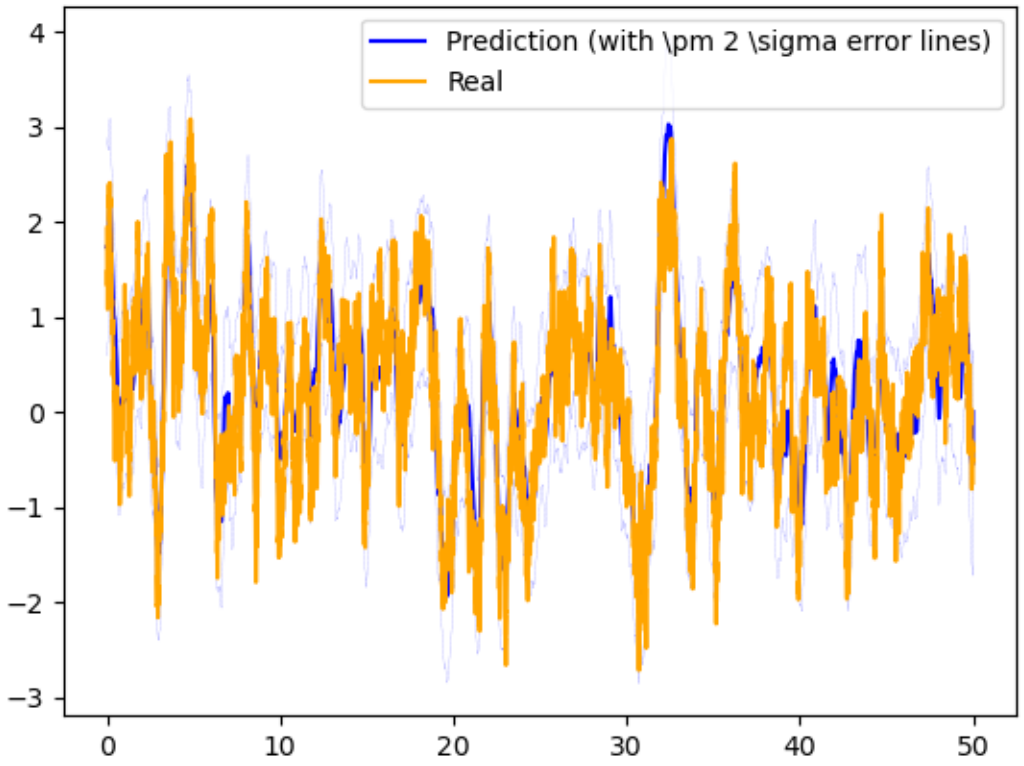
(continued from previous page)

```
# pot_gen_polynom = GLE_BasisTransform(basis_type="polynomial", degree=3)
# pot_gen = GLE_BasisTransform(transformer=FunctionTransformer(dV))
generator = GLE_Estimator(verbose=2, dim_x=dim_x, dim_h=dim_h, basis=pot_gen, force_
↪init=force, init_params="random", C_init=C, random_state=random_state)
X, idx, Xh = generator.sample(n_samples=10000, n_trajs=25, x0=0.0, v0=0.0)

basis = GLE_BasisTransform(basis_type="linear")
# basis = GLE_BasisTransform(basis_type="polynomial", degree=3)

est = GLE_Estimator(init_params="user", dim_x=dim_x, dim_h=dim_h, basis=basis,␣
↪OptimizeDiffusion=True)
est.set_init_coeffs(generator.get_coefficients())
est.dt = X[1, 0] - X[0, 0]
est._check_initial_parameters()


traj_list_h = np.split(Xh, idx)
# for n, traj in enumerate(traj_list_h):
#     traj_list_h[n] = traj_list_h[n][:-1, :]  # For euler

Xproc, idx = est.model_class.preprocessingTraj(basis, X, idx_trajs=idx)
traj_list = np.split(Xproc, idx)

est._initialize_parameters(random_state=42)
datas = 0.0
for n, traj in enumerate(traj_list):
    datas_visible = sufficient_stats(traj, est.dim_x)
    zero_sig = np.zeros((len(traj), 2 * est.dim_h, 2 * est.dim_h))
    # muh = np.hstack((np.roll(traj_list_h[n], -1, axis=0), traj_list_h[n]))
    muh, Sigh = est._e_step(traj)  # Compute hidden variable distribution
    datas += sufficient_stats_hidden(muh, Sigh, traj, datas_visible, est.dim_x, est.dim_
↪h, est.dim_coeffs_force) / len(traj_list)
    # print(datas)

print(generator.get_coefficients())
logL1 = est.loglikelihood(datas)
print(logL1)

est._m_step(datas)
logL3 = est.loglikelihood(datas)
print("Analytic", est.get_coefficients())
print("Diff")
anal_coef = est.get_coefficients()
```

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

3.5 E step, Kalman filter

Inner working of the E step *GLE_analysisEM.GLE_Estimator*




```
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/
↪examples/plot_e_step.py:17: FutureWarning: Passing a negative integer is deprecated in␣
↪version 1.0 and will not be supported in future version. Instead, use None to not␣
↪limit the column width.
  pd.set_option("display.max_colwidth", -1)
{'A': array([[ 0.02645208,  1.        ],
       [-1.        ,  1.6849652 ]]), 'C': array([[ 1.00000000e+00, -1.67919914e-17],
       [-3.58404070e-17,  1.00000000e+00]]), 'force': array([[-1.]]), 'μ_0': array([0.]),
↪ '_0': array([[1.]]), 'SST': array([[0.00026452, 0.        ],
       [0.        , 0.01684965]]), 'dt': 0.005, 'basis': {'basis_type': 'linear',
↪'transformer': None, 'dim_x': 1, 'nb_basis_elt': 1, 'mean': array([0.]), 'n_output_
↪features': 1}}
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
```

(continues on next page)

(continued from previous page)

```
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
```

(continues on next page)

(continued from previous page)

```
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:103: RuntimeWarning: Mean of empty slice.
  hSdouble = 0.5 * np.log(detd[detd > 0.0]).mean()
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/conda/latest/lib/python3.
↪9/site-packages/numpy/core/_methods.py:188: RuntimeWarning: invalid value encountered␣
↪in double_scalars
  ret = ret.dtype.type(ret / rcount)
/home/docs/checkouts/readthedocs.org/user_builds/gle-analysisem/checkouts/latest/GLE_
↪analysisEM/_gle_estimator.py:104: RuntimeWarning: Mean of empty slice.
  hSsimple = 0.5 * np.log(dets[dets > 0.0]).mean()
{'A': array([[ 0.02645208,  1.        ],
       [-1.        ,  1.6849652 ]]), 'C': array([[ 1.00000000e+00, -2.65775124e-16],
       [-4.93096709e-17,  1.00000000e+00]]), 'force': array([[-1.]]), 'µ_0': array([0.]),
↪ '_0': array([[1.]]), 'SST': array([[ 2.64520814e-04, -1.73472348e-18],
       [-1.73472348e-18,  1.68496520e-02]]), 'dt': 0.005, 'basis': {'basis_type': 'linear
↪', 'transformer': None, 'dim_x': 1, 'nb_basis_elt': 1, 'mean': array([0.]), 'n_output_
↪features': 1}}
Real datas
dxdx    [[0.0003105327689501738, 3.421232830534903e-05], [3.421232830534903e-05, 0.
↪016813548574053382]]
xdx     [[-0.00011558038291017877, 0.004690654455607655], [-0.0047642304638938135, -0.
```

(continues on next page)

(continued from previous page)

```
↪0084199405683121 67]]
xx      [[0.9011063718703859, -0.025579194692135276], [-0.025579194692135276, 0.
↪9535644807589472]]
bkx     [[0.013926371314893363, 0.004575700354467006]]
bkdx    [[-0.00453191153991419, 0.00011368340416723901]]
bkbk    [[0.8948908969005819]]
μ_0     [-0.20328789892473598]
_0     [[0.0]]
hS      NaN
dtype: object
Estimated datas
dxdx    [[0.0003105327689501738, 4.139405031848609e-05], [4.139405031848609e-05, 0.
↪016939558734364833]]
xdx     [[-0.00011558038291017877, 0.0047719838764412525], [-0.004819320300032244, -0.
↪00847955320308491]]
xx      [[0.9011063718703859, -0.016194579896374562], [-0.016194579896374562, 0.
↪9589237375785946]]
bkx     [[0.013926371314893363, 0.010360234927923786]]
bkdx    [[-0.00453191153991419, 4.9370791776712645e-05]]
bkbk    [[0.8948908969005819]]
μ_0     [0.2904700508604896]
_0     [[0.3378154504392095]]
hS     -0.638995
dtype: object
Diff
dxdx    [[0.0, 0.20991620181588788], [0.20991620181588788, 0.007494560696479 59]]
xdx     [[0.0, 0.01733860841878234], [-0.01156321814318897, -0.007079935337915734]]
xx      [[0.0, 0.3668846853355458], [0.3668846853355458, 0.005620235367179342]]
bkx     [[0.0, 1.2641856164837486]]
bkdx    [[0.0, -0.5657168067901666]]
bkbk    [[0.0]]
μ_0     [2.428860509636294]
_0     [[inf]]
hS      NaN
dtype: object
[[ 0.00014039  0.00497411]
 [-0.00513833  0.008756  ]]
[[2.63865309e-04 1.32586614e-07]
 [1.32586614e-07 1.68407917e-02]] [[0.00026452 0.        ]
 [0.         0.01684965]]
```

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from GLE_analysisEM import GLE_Estimator, GLE_BasisTransform, sufficient_stats,␣
↪sufficient_stats_hidden
```

(continues on next page)

(continued from previous page)

```
# Printing options
pd.set_option("display.max_rows", None)
pd.set_option("display.max_columns", None)
pd.set_option("display.width", None)
pd.set_option("display.max_colwidth", -1)

dim_x = 1
dim_h = 1
random_state = None
force = -np.identity(dim_x)
A = [[5, 1.0], [-1.0, 2.07]]
C = np.identity(dim_x + dim_h)  #
basis = GLE_BasisTransform(basis_type="linear")
generator = GLE_Estimator(verbose=1, dim_x=dim_x, dim_h=dim_h, basis=basis, C_init=C,␣
↪force_init=force, init_params="random", random_state=random_state)
X, idx, Xh = generator.sample(n_samples=10000, n_trajs=10, x0=0.0, v0=0.0)
traj_list_h = np.split(Xh, idx)
time = np.split(X, idx)[0][:, 0]
for n, traj in enumerate(traj_list_h):
    traj_list_h[n] = traj_list_h[n][:-1, :]

print(generator.get_coefficients())

est = GLE_Estimator(init_params="user", dim_x=dim_x, dim_h=dim_h, basis=basis)
est.set_init_coeffs(generator.get_coefficients())
est.dt = time[1] - time[0]
est._check_initial_parameters()

Xproc, idx = est.model_class.preprocessingTraj(est.basis, X, idx_trajs=idx)
traj_list = np.split(Xproc, idx)
est.dim_coeffs_force = est.basis.nb_basis_elt_

datas = 0.0
for n, traj in enumerate(traj_list):
    datas_visible = sufficient_stats(traj, est.dim_x)
    zero_sig = np.zeros((len(traj), 2 * est.dim_h, 2 * est.dim_h))
    muh = np.hstack((np.roll(traj_list_h[n], -1, axis=0), traj_list_h[n]))
    datas += sufficient_stats_hidden(muh, zero_sig, traj, datas_visible, est.dim_x, est.
↪dim_h, est.dim_coeffs_force) / len(traj_list)

est._initialize_parameters(None)
print(est.get_coefficients())
print("Real datas")
print(datas)
new_stat = 0.0
noise_corr = 0.0
for n, traj in enumerate(traj_list):
    datas_visible = sufficient_stats(traj, est.dim_x)
    muh, Sigh = est._e_step(traj)  # Compute hidden variable distribution
    new_stat += sufficient_stats_hidden(muh, Sigh, traj, datas_visible, est.dim_x, est.
↪dim_h, est.dim_coeffs_force) / len(traj_list)
print("Estimated datas")
```

(continues on next page)

(continued from previous page)

```
print(new_stat)
print("Diff")
print((new_stat - datas) / np.abs(datas))


Pf = np.zeros((dim_x + dim_h, dim_x))
Pf[:dim_x, :dim_x] = 5e-3 * np.identity(dim_x)
YX = new_stat["xdx"].T - np.matmul(Pf, np.matmul(force, new_stat["bkx"]))
XX = new_stat["xx"]
A = -np.matmul(YX, np.linalg.inv(XX))


Pf = np.zeros((dim_x + dim_h, dim_x))
Pf[:dim_x, :dim_x] = 5e-3 * np.identity(dim_x)

# A = generator.friction_coeffs
print(A)
bkbk = np.matmul(Pf, np.matmul(np.matmul(force, np.matmul(new_stat["bkbk"], force.T)),␣
↪Pf.T))
bkdx = np.matmul(Pf, np.matmul(force, new_stat["bkdx"]))
bkx = np.matmul(Pf, np.matmul(force, new_stat["bkx"]))

residuals = new_stat["dxdx"] + np.matmul(A, new_stat["xdx"]) + np.matmul(A, new_stat["xdx
↪"]).T - bkdx.T - bkdx
residuals += np.matmul(A, np.matmul(new_stat["xx"], A.T)) - np.matmul(A, bkx.T) - np.
↪matmul(A, bkx.T).T + bkbk
print(residuals, generator.diffusion_coeffs)
# SST = 0.5 * (residuals + residuals.T)

fig, axs = plt.subplots(1, dim_h)
# plt.show()
for k in range(dim_h):
    axs.plot(time[:-1], muh[:, k], label="Prediction (with \\pm 2 \\sigma error lines)",␣
↪color="blue")
    axs.plot(time[:-1], muh[:, k] + 2 * np.sqrt(Sigh[:, k, k]), "--", color="blue",␣
↪linewidth=0.1)
    axs.plot(time[:-1], muh[:, k] - 2 * np.sqrt(Sigh[:, k, k]), "--", color="blue",␣
↪linewidth=0.1)
    axs.plot(time[:-1], traj_list_h[n][:, k], label="Real", color="orange")
    axs.legend(loc="upper right")
plt.show()
```

Total running time of the script: (0 minutes 18.367 seconds)

3.6 Parameters estimation from Markovian

Plot obtained values of the parameters versus the actual ones when the parameters are estimated from a Markov model

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

from GLE_analysisEM import GLE_BasisTransform, GLE_Estimator, GLE_PotentialTransform

from GLE_analysisEM import Markov_Estimator
from GLE_analysisEM.post_processing import forcefield, forcefield_plot2D, correlation

from sklearn.preprocessing import FunctionTransformer

a = 0.025
b = 1.0


def dV(X):
    """
    Compute the force field
    """
    return 4 * a * np.power(X, 3) - 2 * b * X


dim_x = 1
dim_h = 1
random_state = None
force = -np.identity(dim_x)
# force = [[-0.25, -1], [1, -0.25]]
A = np.array([[5e-2, -1.0], [1.0, 0.1]])

# ------ Generation ------#
# pot_gen = GLE_BasisTransform(basis_type="linear")
pot_gen = GLE_BasisTransform(transformer=FunctionTransformer(dV))
# pot_gen_polynom = GLE_BasisTransform(basis_type="polynomial", degree=3)
generator = GLE_Estimator(verbose=2, dim_x=dim_x, dim_h=2, basis=pot_gen, init_params=
↪"random", force_init=force, random_state=random_state)
X, idx, Xh = generator.sample(n_samples=20000, n_trajs=25, x0=0.0, v0=0.0)
print("---- Real ones ----")
print(generator.get_coefficients())

# for n in range(dim_x):
#     plt.plot(X[:, 0], X[:, n * 2 + 2], label="v{}".format(n + 1))
#     plt.plot(X[:, 0], X[:, n * 2 + 1], label="x{}".format(n + 1))
#
# plt.show()
# ------ Estimation ------#
# basis = GLE_BasisTransform(basis_type="linear")
basis = GLE_BasisTransform(basis_type="polynomial", degree=3).fit(X[1 : 1 + dim_x])
estimator = Markov_Estimator(init_params="random", verbose=2, verbose_interval=1, dim_
↪x=dim_x, basis=basis, n_init=1, OptimizeForce=True, random_state=7)
estimator.fit(X, idx_trajs=idx)
```

(continues on next page)

(continued from previous page)

```
# print(estimator.get_coefficients())

# Free energy

potential = GLE_PotentialTransform(estimator="histogram", dim_x=dim_x)
potential.fit(X)

# ------ Plotting ------#
fig, axs = plt.subplots(2)


# ------ Force field ------#
axs[0].set_title("Force field")
force_true = generator.get_coefficients()["force"]
force_fitted = estimator.get_coefficients()["force"]
if dim_x == 1:
    x_lims = [[-10, 10, 25]]
    xfx_true = forcefield(x_lims, pot_gen, force_true)
    xfx = forcefield(x_lims, basis, force_fitted)
    axs[0].plot(xfx_true[:, 0], xfx_true[:, 1], label="True force field")
    axs[0].plot(xfx[:, 0], xfx[:, 1], label="Fitted force field")

    x_lims = [[-8, 8, 150]]
    x_val = np.linspace(x_lims[0][0], x_lims[0][1], x_lims[0][2]).reshape(-1, 1)
    pot_val = potential.predict(x_val)
    axs[0].plot(x_val[:, 0], pot_val[:, 0], label="Fitted potential")

if dim_x == 2:
    x_lims = [[-2, 2, 10], [-2, 2, 10]]
    x_true, y_true, fx_true, fy_true = forcefield_plot2D(x_lims, basis, force_true)
    x, y, fx, fy = forcefield_plot2D(x_lims, basis, force_fitted)
    axs[0].quiver(x_true, y_true, fx_true, fy_true, width=0.001, color="green", label=
↪"True force field")
    axs[0].quiver(x, y, fx, fy, width=0.001, color="blue", label="Fitted force field")

axs[0].legend(loc="upper right")


def simulated_vacf(estimator, basis):
    """
    Get vacf via numericall simulation of the model
    """
    Ntrajs = 5
    X, idx, Xh = estimator.sample(n_samples=10000, n_trajs=Ntrajs)
    traj_list = np.split(X, idx)
    vacf = 0.0
    for n, trj in enumerate(traj_list):
        vacf += correlation(trj[:, 1 + estimator.dim_x])
        time = trj[:, 0]
    # vacf /= Ntrajs
    return time, vacf / vacf[0]
```

(continues on next page)

(continued from previous page)

```
# ------ Diffusion ------#
traj_list = np.split(X, idx)
vacf_num = 0.0
for n, trj in enumerate(traj_list):
    vacf_num += correlation(trj[:, 2])
    time = trj[:, 0]
# vacf_num /= len(traj_list)
vacf_num /= vacf_num[0]
time_sim, vacf_sim = simulated_vacf(estimator, basis)
axs[1].plot(time[: len(time_sim) // 2], vacf_sim, label="Fitted VACF")
axs[1].set_title("Velocity autocorrelation function")
axs[1].plot(time[: len(time) // 2], vacf_num, label="Numerical VACF")


axs[1].legend(loc="upper right")

plt.show()
```

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

3.7 Parameters estimation

Plot obtained values of the parameters versus the actual ones

```
import numpy as np
from matplotlib import pyplot as plt
from GLE_analysisEM import GLE_Estimator, GLE_BasisTransform
from GLE_analysisEM.post_processing import memory_kernel, forcefield, forcefield_plot2D,␣
↪correlation, memory_timescales

from sklearn.preprocessing import FunctionTransformer

a = 0.025
b = 1.0


def dV(X):
    """
    Compute the force field
    """
    return 4 * a * np.power(X, 3) - 2 * b * X


dim_x = 1
dim_h = 2
random_state = None
force = -np.identity(dim_x)
# force = [[-0.25, -1], [1, -0.25]]
A = np.array([[5e-2, -1.0], [1.0, 0.1]])
```

(continues on next page)

(continued from previous page)

```
# ------ Generation ------#
# pot_gen = GLE_BasisTransform(basis_type="linear")
pot_gen = GLE_BasisTransform(transformer=FunctionTransformer(dV))
generator = GLE_Estimator(verbose=2, dim_x=dim_x, dim_h=dim_h, basis=pot_gen, force_
↪init=force, init_params="random", random_state=random_state)
X, idx, Xh = generator.sample(n_samples=20000, n_trajs=25, x0=0.0, v0=0.0)

# ------ Estimation ------#
# basis = GLE_BasisTransform(basis_type="linear")
basis = GLE_BasisTransform(basis_type="free_energy_kde")
estimator = GLE_Estimator(verbose=2, verbose_interval=10, dim_x=dim_x, dim_h=dim_h,␣
↪basis=basis, n_init=1, random_state=7, tol=1e-7, no_stop=False, max_iter=100,␣
↪multiprocessing=8)
estimator.fit(X, idx_trajs=idx)

# ------ Plotting ------#
fig, axs = plt.subplots(2, 2)


# ------ Force field ------#
axs[0, 0].set_title("Force field")
force_true = generator.get_coefficients()["force"]
force_fitted = estimator.get_coefficients()["force"]
if dim_x == 1:
    x_lims = [[-10, 10, 25]]
    xfx_true = forcefield(x_lims, pot_gen, force_true)
    xfx = forcefield(x_lims, basis, force_fitted)
    axs[0, 0].plot(xfx_true[:, 0], xfx_true[:, 1], label="True force field")
    axs[0, 0].plot(xfx[:, 0], xfx[:, 1], label="Fitted force field")

if dim_x == 2:
    x_lims = [[-2, 2, 10], [-2, 2, 10]]
    x_true, y_true, fx_true, fy_true = forcefield_plot2D(x_lims, basis, force_true)
    x, y, fx, fy = forcefield_plot2D(x_lims, basis, force_fitted)
    axs[0, 0].quiver(x_true, y_true, fx_true, fy_true, width=0.001, color="green", label=
↪"True force field")
    axs[0, 0].quiver(x, y, fx, fy, width=0.001, color="blue", label="Fitted force field")

axs[0, 0].legend(loc="upper right")

# ------ Memory kernel ------#
axs[0, 1].set_title("Memory kernel")
time, kernel = memory_kernel(1000, estimator.dt, estimator.get_coefficients(), dim_x)
time_true, kernel_true = memory_kernel(1000, generator.dt, generator.get_coefficients(),␣
↪dim_x)


axs[0, 1].plot(time, kernel[:, 0, 0], label="Fitted memory kernel")
axs[0, 1].plot(time_true, kernel_true[:, 0, 0], label="True memory kernel")
axs[0, 1].legend(loc="upper right")
```

(continues on next page)

(continued from previous page)

```
# ------ Memory eigenvalues ------#
axs[1, 1].set_title("Kernel Eigenvalues")
mem_ev = memory_timescales(estimator.get_coefficients(), dim_x=dim_x)
mem_ev_true = memory_timescales(generator.get_coefficients(), dim_x=dim_x)
axs[1, 1].scatter(np.real(mem_ev), np.imag(mem_ev), label="Ev fitted")
axs[1, 1].scatter(np.real(mem_ev_true), np.imag(mem_ev_true), label="Ev true")
# axs[1, 1].set_aspect(1)


def simulated_vacf(estimator):
    """
    Get vacf via numericall simulation of the model
    """
    Ntrajs = 5
    X, idx, Xh = estimator.sample(n_samples=10000, n_trajs=Ntrajs)
    traj_list = np.split(X, idx)
    vacf = 0.0
    for n, trj in enumerate(traj_list):
        vacf += correlation(trj[:, 1 + estimator.dim_x])
        time = trj[:, 0]
    vacf /= vacf[0]
    return time, vacf


# ------ Diffusion ------#
traj_list = np.split(X, idx)
vacf_num = 0.0
for n, trj in enumerate(traj_list):
    vacf_num += correlation(trj[:, 2])
    time = trj[:, 0]
vacf_num /= vacf_num[0]
time_sim, vacf_sim = simulated_vacf(estimator)
axs[1, 0].plot(time[: len(time_sim) // 2], vacf_sim, label="Fitted VACF")
axs[1, 0].set_title("Velocity autocorrelation function")
axs[1, 0].plot(time[: len(time) // 2], vacf_num, label="Numerical VACF")
axs[1, 0].legend(loc="upper right")

plt.show()
```

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

Note: Installing the library is as simple as running

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

CHAPTER

FOUR

USER GUIDE

CHAPTER

FIVE

API DOCUMENTATION

CHAPTER

SIX

EXAMPLES

INDEX

Symbols

G

L

M