PyEDGAR

Release 0.2.0

May 28, 2019

Contents

CHAPTER 1

Overview

docs	
tests	

Package for performing dynamical Galerkin expansion on trajectory data. Currently in pre-alpha.

- Free software: MIT license

1.1 Installation

To install the code, download the directory from github, navigate into the folder, and run

```
pip install -e .
```

We are currently working on getting the package onto pip.

1.2 Documentation

https://PyEDGAR.readthedocs.io/

1.3 Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 2

Installation

At the command line:

```
pip install pyedgar
```

CHAPTER 3

Usage

To use PyEDGAR in a project:

```
import pyedgar
```

CHAPTER 4

Jupyter notebook tutorials

4.1 Committor Estimate on the Muller-Brown Potential

```
import matplotlib.pyplot as plt
import numpy as np
import pyedgar
from pyedgar.data_manipulation import tlist_to_flat, flat_to_tlist

%matplotlib inline
```

4.1.1 Load Data and set Hyperparameters

We first load in the pre-sampled data. The data consists of 1000 short trajectories, each with 5 datapoints. The precise sampling procedure is described in "Galerkin Approximation of Dynamical Quantities using Trajectory Data" by Thiede et al. Note that this is a smaller dataset than in the paper. We use a smallar dataset to ensure the diffusion map basis construction runs in a reasonably short time.

Set Hyperparameters

Here we specify a few hyperparameters. Thes can be varied to study the behavior of the scheme in various limits by the user.

```
ntraj = 1000
trajectory_length = 5
dim = 10
```

Load and format the data

```
trajs = np.load('data/muller_brown_trajs.npy')[:ntraj, :trajectory_length, :dim] #␣
↪Raw trajectory
stateA = np.load('data/muller_brown_stateA.npy')[:ntraj, :trajectory_length] # 1 if␣
↪in state A, 0 otherwise
stateB = np.load('data/muller_brown_stateB.npy')[:ntraj, :trajectory_length] # 1 if␣
↪in state B, 0 otherwise

print("Data shape: ", trajs.shape)

# Convert to list of trajectories format
trajs = [traj_i for traj_i in trajs]
stateA = [A_i for A_i in stateA]
stateB = [B_i for B_i in stateB]
```

```
Data shape:  (1000, 5, 10)
```

We also convert the data into the flattened format. This converts the data into a 2D array, which allows the data to be passed into many ML packages that require a two-dimensional dataset. In particular, this is the format accepted by the Diffusion Atlas object. Trajectory start/stop points are then stored in the traj_edges array.

```
flattened_trajs, traj_edges = tlist_to_flat(trajs)
flattened_stateA = np.hstack(stateA)
flattened_stateB = np.hstack(stateB)
print("Flattened Shapes are: ", flattened_trajs.shape, flattened_stateA.shape,␣
↪flattened_stateB.shape,)
```

```
Flattened Shapes are:  (5000, 10) (5000,) (5000,)
```

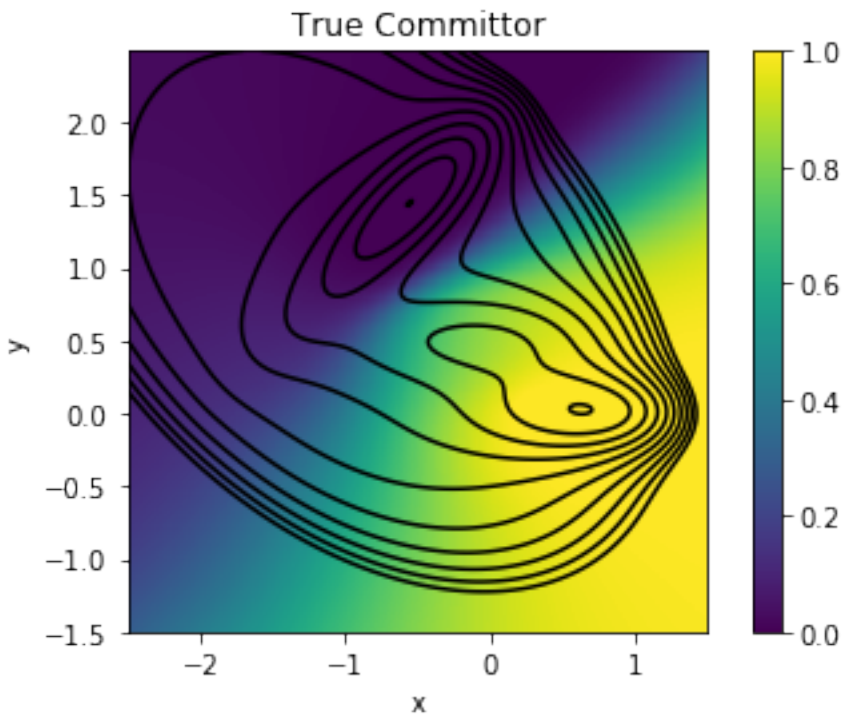
Finally, we load the reference, "true" committor for comparison.

```
ref_comm = np.load('reference/reference_committor.npy')
ref_potential = np.load('reference/potential.npy')
xgrid = np.load('reference/xgrid.npy')
ygrid = np.load('reference/ygrid.npy')
```

```
 # Plot the true committor.
fig, ax = plt.subplots(1)
HM = ax.pcolor(xgrid, ygrid, ref_comm, vmin=0, vmax=1)
ax.contour(xgrid, ygrid, ref_potential, levels=np.linspace(0, 10., 11), colors='k') #␣
↪Contour lines every 1 k_B T
ax.set_aspect('equal')
cbar = plt.colorbar(HM, ax=ax)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('True Committor')
```

```
Text(0.5,1,'True Committor')
```



4.1.2 Construct DGA Committor

We now use PyEDGAR to build an estimate for the forward committor.

Build Basis Set

We first build the basis set required for the DGA Calculation. In this demo, we will use the diffusion map basis.

```
diff_atlas = pyedgar.basis.DiffusionAtlas.from_sklearn(alpha=0, k=500, bandwidth_type=
↪'-1/d', epsilon='bgh_generous')
diff_atlas.fit(flattened_trajs)
```

```
<pyedgar.basis.DiffusionAtlas at 0x7fab1ef29748>
```

Here, we construct the basis and guess functions, and convert them back into lists of trajectories. The domain is the set of all sets out side of $(A \cup B)^c$.

```
flat_basis, evals = diff_atlas.make_dirichlet_basis(300, in_domain=(1. - flattened_
↪stateA - flattened_stateB), return_evals=True)
flat_guess = diff_atlas.make_FK_soln(flattened_stateB, in_domain=(1. - flattened_
↪stateA - flattened_stateB))

basis = flat_to_tlist(flat_basis, traj_edges)
guess = flat_to_tlist(flat_guess, traj_edges)
```

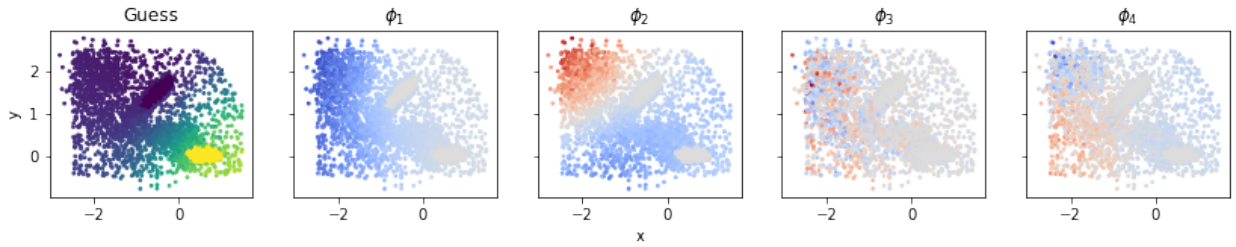
We plot the guess function and the first few basis functions.

```
fig, axes= plt.subplots(1, 5, figsize=(14,4.), sharex=True, sharey=True)
axes[0].scatter(flattened_trajs[:,0], flattened_trajs[:,1],
                c=flat_guess, s=3)
axes[0].set_title('Guess')
axes[0].set_ylabel("y")

for i, ax in enumerate(axes[1:]):
    vm = np.max(np.abs(flat_basis[:, i]))
    ax.scatter(flattened_trajs[:,0], flattened_trajs[:,1],
               c=flat_basis[:, i], s=3, cmap='coolwarm',
               vmin=-1*vm, vmax=vm)
    ax.set_title(r"$\phi_%d$" % (i+1))

for ax in axes:
    ax.set_aspect('equal')
#     ax.
axes[2].set_xlabel("x")
```

```
Text(0.5,0,'x')
```



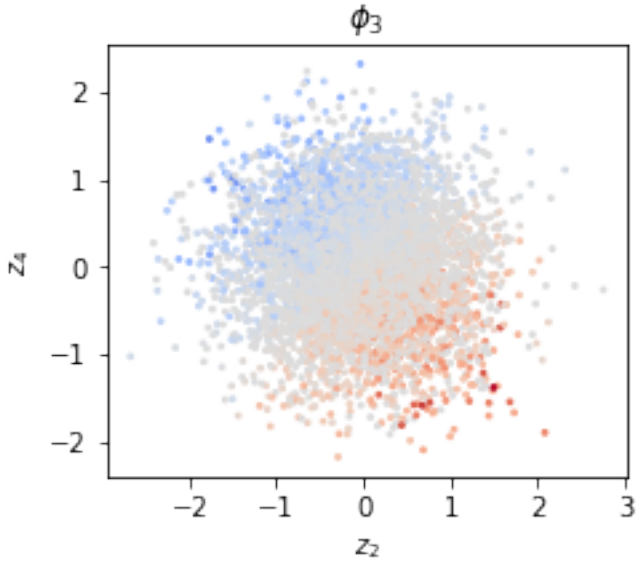
The third basis function looks like noise from the perspective of the x and y coordinates. This is because it correlates most strongly with the harmonic degrees of freedom. Note that due to the boundary conditions, it is not precisely the dominant eigenvector of the harmonic degrees of freedom.

```
fig, (ax1) = plt.subplots(1, figsize=(3.5,3.5))

vm = np.max(np.abs(flat_basis[:,2]))
ax1.scatter(flattened_trajs[:,3], flattened_trajs[:,5],
            c=flat_basis[:, 2], s=3, cmap='coolwarm',
            vmin=-1*vm, vmax=vm)

ax1.set_aspect('equal')
ax1.set_title(r"$\phi_%d$" % 3)
ax1.set_xlabel("$z_2$")
ax1.set_ylabel("$z_4$")
```

```
Text(0,0.5,'$z_4$')
```



Build the committor function

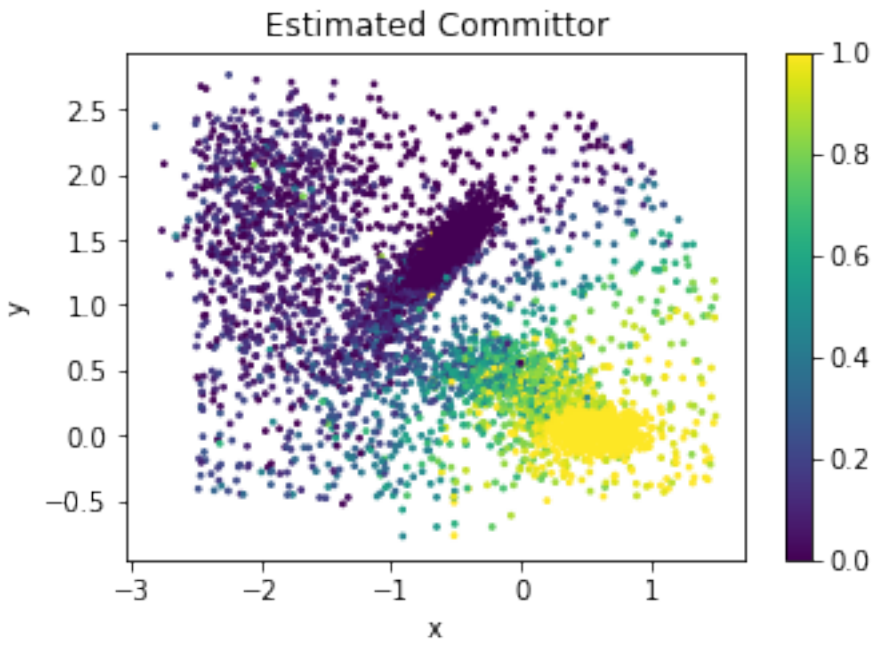
We are ready to compute the committor function using DGA. This can be done by passing the guess function and the basis to the the Galerkin module.

```
g = pyedgar.galerkin.compute_committor(basis, guess, lag=1)
```

```
fig, (ax1) = plt.subplots(1, figsize=(5.5,3.5))

SC = ax1.scatter(flattened_trajs[:,0], flattened_trajs[:,1], c=np.array(g).ravel(),␣
↪vmin=0., vmax=1., s=3)

ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Estimated Committor')
plt.colorbar(SC)
ax1.set_aspect('equal')
```

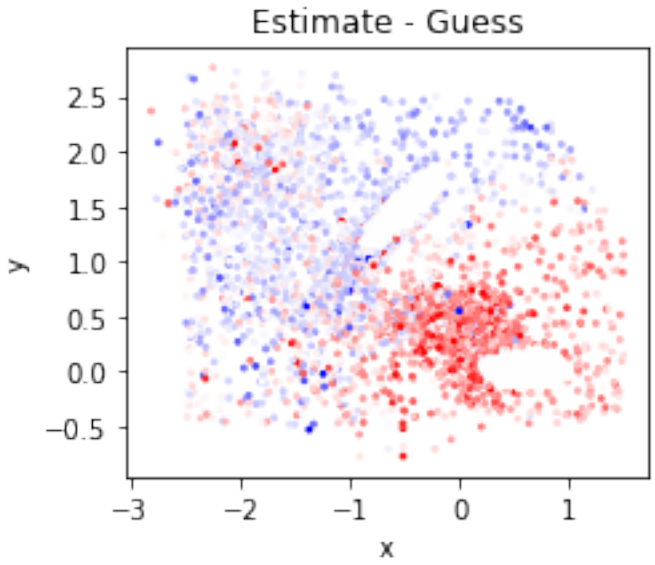


Here, we plot how much the DGA estimate perturbs the Guess function

```
fig, (ax1) = plt.subplots(1, figsize=(3.5,3.5))

ax1.scatter(flattened_trajs[:,0], flattened_trajs[:,1], c=np.array(g).ravel() - flat_
↪guess,
            vmin=-.5, vmax=.5, cmap='bwr', s=3)
ax1.set_aspect('equal')
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('Estimate - Guess')
```

```
Text(0.5,1,'Estimate - Guess')
```



4.1.3 Compare against reference

To compare against the reference values, we will interpolate the reference onto the datapoints usingy scipy's interpolate package.

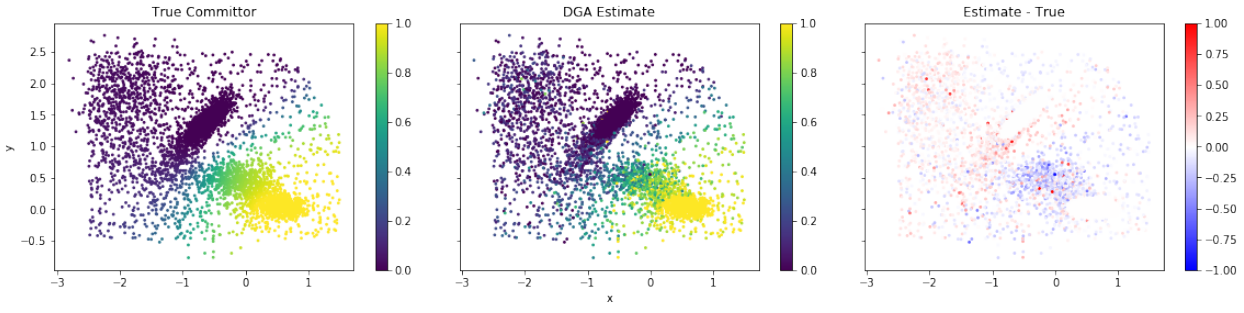
```
import scipy.interpolate as spi

spline = spi.RectBivariateSpline(xgrid, ygrid, ref_comm.T)
ref_comm_on_data = np.array([spline.ev(c[0], c[1]) for c in flattened_trajs[:,:2]])
ref_comm_on_data[ref_comm_on_data < 0.] = 0.
ref_comm_on_data[ref_comm_on_data > 1.] = 1.
```

A comparison of our estimate with the True committor. While the estimate is good, we systematically underestimate the committor near (0, 0.5).

```
fig, axes = plt.subplots(1, 3, figsize=(16,3.5), sharex=True, sharey=True)
(ax1, ax2, ax3) = axes
SC = ax1.scatter(flattened_trajs[:,0], flattened_trajs[:,1], c=ref_comm_on_data,␣
↪vmin=0., vmax=1., s=3)
plt.colorbar(SC, ax=ax1)
SC = ax2.scatter(flattened_trajs[:,0], flattened_trajs[:,1], c=np.array(g).ravel(),␣
↪vmin=0., vmax=1., s=3)
plt.colorbar(SC, ax=ax2)
SC = ax3.scatter(flattened_trajs[:,0], flattened_trajs[:,1], c=np.array(g).ravel() -
↪ref_comm_on_data,
                 vmin=-1, vmax=1, s=3, cmap='bwr')
plt.colorbar(SC, ax=ax3)


# ax1.set_aspect('equal')
ax2.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_title('True Committor')
ax2.set_title('DGA Estimate')
ax3.set_title('Estimate - True')
plt.tight_layout(pad=-1.)
for ax in axes:
    ax.set_aspect('equal')
```



4.2 Delay Embedding and the MFPT

Here, we give an example script, showing the effect of Delay Embedding on a Brownian motion on the Muller-Brown potential, projeted onto its y-axis. This script may take a long time to run, as considerable data is required to accurately reconstruct the hidden degrees of freedom.

```
import matplotlib.pyplot as plt
import numpy as np
import pyedgar
from pyedgar.data_manipulation import tlist_to_flat, flat_to_tlist, delay_embed, lift_
↪function

%matplotlib inline
```

4.2.1 Load Data and set Hyperparameters

We first load in the pre-sampled data. The data consists of 400 short trajectories, each with 30 datapoints. The precise sampling procedure is described in "Galerkin Approximation of Dynamical Quantities using Trajectory Data" by Thiede et al. Note that this is a smaller dataset than in the paper. We use a smallar dataset to ensure the diffusion map basis construction runs in a reasonably short time.

Set Hyperparameters

Here we specify a few hyperparameters. Thes can be varied to study the behavior of the scheme in various limits by the user.

```
ntraj = 700
trajectory_length = 40
lag_values = np.arange(1, 37, 2)
embedding_values = lag_values[1:] - 1
```

Load and format the data

```
trajs_2d = np.load('data/muller_brown_trajs.npy')[:ntraj, :trajectory_length] # Raw␣
↪trajectory
trajs = trajs_2d[:, :, 1] # Only keep y coordinate
stateA = (trajs > 1.15).astype('float')
stateB = (trajs < 0.15).astype('float')

# Convert to list of trajectories format
trajs = [traj_i.reshape(-1, 1) for traj_i in trajs]
stateA = [A_i for A_i in stateA]
stateB = [B_i for B_i in stateB]

# Load the true results
true_mfpt = np.load('data/htAB_1_0_0_1.npy')
```

We also convert the data into the flattened format. This converts the data into a 2D array, which allows the data to be passed into many ML packages that require a two-dimensional dataset. In particular, this is the format accepted by the Diffusion Atlas object. Trajectory start/stop points are then stored in the traj_edges array.

```
flattened_trajs, traj_edges = tlist_to_flat(trajs)
flattened_stateA = np.hstack(stateA)
flattened_stateB = np.hstack(stateB)
print("Flattened Shapes are: ", flattened_trajs.shape, flattened_stateA.shape,␣
↪flattened_stateB.shape,)
```

```
Flattened Shapes are:  (28000, 1) (28000,) (28000,)
```

4.2.2 Construct DGA MFPT by increasing lag times

We first construct the MFPT with increasing lag times.

```
# Build the basis set
diff_atlas = pyedgar.basis.DiffusionAtlas.from_sklearn(alpha=0, k=500, bandwidth_type=
↪'-1/d', epsilon='bgh_generous')
diff_atlas.fit(flattened_trajs)
flat_basis = diff_atlas.make_dirichlet_basis(200, in_domain=(1. - flattened_stateA))
basis = flat_to_tlist(flat_basis, traj_edges)
flat_basis_no_boundaries = diff_atlas.make_dirichlet_basis(200)
basis_no_boundaries = flat_to_tlist(flat_basis_no_boundaries, traj_edges)
```

```
# Perform DGA calculation
mfpt_BA_lags = []
for lag in lag_values:
    mfpt = pyedgar.galerkin.compute_mfpt(basis, stateA, lag=lag)
    pi = pyedgar.galerkin.compute_change_of_measure(basis_no_boundaries, lag=lag)
    flat_pi = np.array(pi).ravel()
    flat_mfpt = np.array(mfpt).ravel()
    mfpt_BA = np.mean(flat_mfpt * flat_pi * np.array(stateB).ravel()) / np.mean(flat_
↪pi * np.array(stateB).ravel())
    mfpt_BA_lags.append(mfpt_BA)
```

4.2.3 Construct DGA MFPT with increasing Delay Embedding

We now construct the MFPT using delay embedding. To accelerate the process, we will only use every fifth value of the delay length.

```
mfpt_BA_embeddings = []
for lag in embedding_values:
    # Perform delay embedding
    debbed_traj = delay_embed(trajs, n_embed=lag)
    lifted_A = lift_function(stateA, n_embed=lag)
    lifted_B = lift_function(stateB, n_embed=lag)

    flat_debbed_traj, embed_edges = tlist_to_flat(debbed_traj)
    flat_lifted_A = np.hstack(lifted_A)

    # Build the basis
    diff_atlas = pyedgar.basis.DiffusionAtlas.from_sklearn(alpha=0, k=500, bandwidth_
↪type='-1/d',
                                                           epsilon='bgh_generous',␣
↪neighbor_params={'algorithm':'brute'})
    diff_atlas.fit(flat_debbed_traj)
    flat_deb_basis = diff_atlas.make_dirichlet_basis(200, in_domain=(1. - flat_lifted_
↪A))
    deb_basis = flat_to_tlist(flat_deb_basis, embed_edges)

    flat_pi_basis = diff_atlas.make_dirichlet_basis(200)
    pi_basis = flat_to_tlist(flat_deb_basis, embed_edges)
```

(continues on next page)

(continued from previous page)

```
    # Construct the Estimate
    deb_mfpt = pyedgar.galerkin.compute_mfpt(deb_basis, lifted_A, lag=1)
    pi = pyedgar.galerkin.compute_change_of_measure(pi_basis)
    flat_pi = np.array(pi).ravel()
    flat_mfpt = np.array(deb_mfpt).ravel()
    deb_mfpt_BA = np.mean(flat_mfpt * flat_pi * np.array(lifted_B).ravel()) / np.
↪mean(flat_pi * np.array(lifted_B).ravel())
    mfpt_BA_embeddings.append(deb_mfpt_BA)
```

4.2.4 Plot the Results

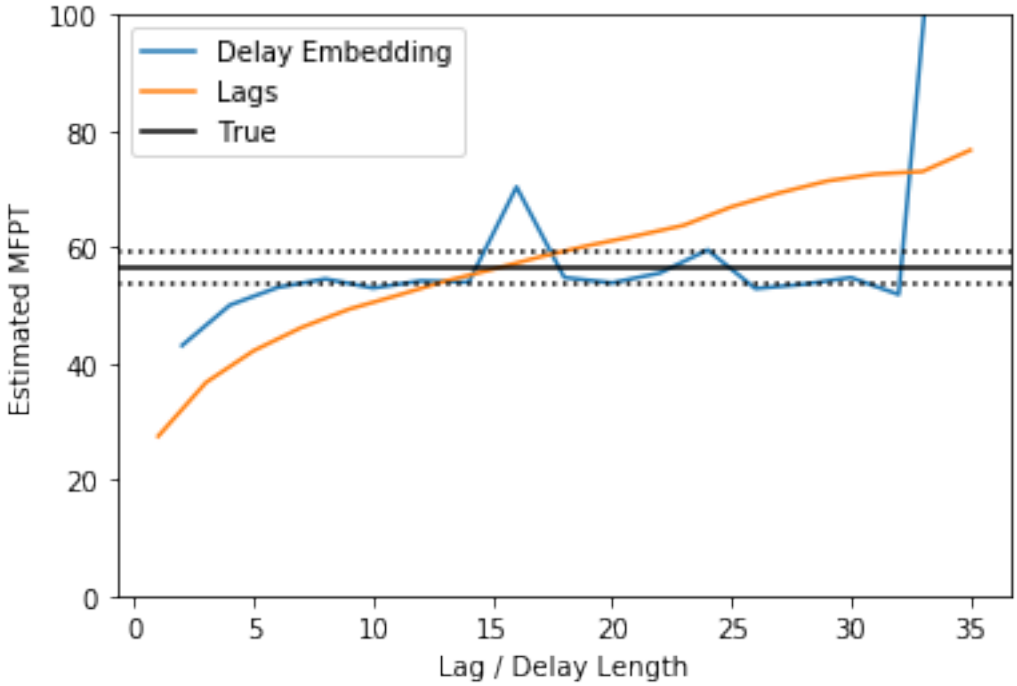
We plot the results of our calculation, against the true value (black line, with the standard deviation in stateB given by the dotted lines). We see that increasing the lag time causes the mean-first-passage time to grow unboundedly. In contrast, with delay embedding the mean-first-passage time converges. We do, however, see one bad fluction at a delay length of 16, and that as the the delay length gets sufficiently long, the calculation blows up.

```
plt.plot(embedding_values, mfpt_BA_embeddings, label="Delay Embedding")
plt.plot(lag_values, mfpt_BA_lags, label="Lags")
plt.axhline(true_mfpt[0] * 10, color='k', label='True')
plt.axhline((true_mfpt[0] + true_mfpt[1]) * 10., color='k', linestyle=':')
plt.axhline((true_mfpt[0] - true_mfpt[1]) * 10., color='k', linestyle=':')

plt.legend()
plt.ylim(0, 100)

plt.xlabel("Lag / Delay Length")
plt.ylabel("Estimated MFPT")
```

```
Text(0,0.5,'Estimated MFPT')
```


Delay Embedding
Lags
True
Estimated MFPT
Lag / Delay Length
0
20
40
60
80
100
0
5
10
15
20
25
30
35

CHAPTER 5

Reference

5.1 Basis

Routines and Class definitions for constructing basis sets using the diffusion maps algorithm.

@author: Erik

class pyedgar.basis.**DiffusionAtlas**(*dmap_object=None*)

The diffusion atlas is a factory object for constructing diffusion map bases with various boundary conditions.

extend_FK_soln(*soln*, *Y*, *b*, *in_domain*)

Extends the values of the Feynman-Kac solution onto new points. In the DGA framework, this is intended to be used to extend guess functions onto new datapoints.

Parameters

- **soln** (*Dataset of same type as the data.*) – Solution to the Feynman-Kac problem on the original type.
- **Y** (*2D array-like OR list of trajectories OR flat data format*) – Data for which to perform the out-of-sample extension.
- **b** (*1D array-like, OR list of such arrays, OR flat data format.*) – Values of the right hand-side for the OOS points.
- **in_domain** (*1D array-like, OR list of such arrays, OR flat data format.*) – Dataset of the same shape as the input datapoints, where each element is 1 or True if that datapoint is inside the domain, and 0 or False if it is in the domain.

Returns **extended_soln** (*Dataset of same type as the data.*) – Solution to the Feynman-Kac problem.

extend_dirichlet_basis(*Y*, *in_domain*, *basis*, *evals*)

Performs out-of-sample extension an a dirichlet basis set.

Parameters

- **Y** (*2D array-like OR list of trajectories OR flat data format*) – Data for which to perform the out-of-sample extension.
- **in_domain** (*1D array-like, OR list of such arrays, OR flat data format*) – Dataset of the same shape as the input datapoints, where each element is 1 or True if that datapoint is inside the domain, and 0 or False if it is in the domain.
- **basis** (*2D array-like OR list of trajectories OR Flat data format*) – The basis functions.
- **evals** (*1D numpy array*) – The eigenvalues corresponding to each basis vector.

Returns **basis_extended** (*Dataset of same type as the data*) – Transformed value of the given values.

fit (*data*)

Constructs the diffusion map on the dataset.

Parameters **data** (*2D array-like OR list of trajectories OR Flat data format*) – Dataset on which to construct the diffusion map.

classmethod from_kernel (*kernel_object*, *alpha=0.5*, *weight_fxn=None*, *density_fxn=None*, *bandwidth_normalize=False*, *oos='nystroem'*)

Builds the Diffusion Atlas using a pyDiffMap kernel. See the pyDiffMap.DiffusionMap constructor for a description of arguments.

classmethod from_sklearn (*alpha=0.5*, *k=64*, *kernel_type='gaussian'*, *epsilon='bgh'*, *neighbor_params=None*, *metric='euclidean'*, *metric_params=None*, *weight_fxn=None*, *density_fxn=None*, *bandwidth_type=None*, *bandwidth_normalize=False*, *oos='nystroem'*)

Builds the Diffusion Atlas using the standard pyDiffMap kernel. See the pyDiffMap.DiffusionMap.from_sklearn for a description of arguments.

make_FK_soln (*b*, *in_domain*)

Solves a Feynman-Kac problem on the data. Specifically, solves Lx = b on the domain and x=b off of the domain. In the DGA framework, this is intended to be used to solve for guess functions.

Parameters

- **b** (*1D array-like, OR list of such arrays, OR flat data format.*) – Dataset of the same shape as the input datapoints. Right hand side of the Feynman-Kac equation.
- **in_domain** (*1D array-like, OR list of such arrays, OR flat data format.*) – Dataset of the same shape as the input datapoints, where each element is 1 or True if that datapoint is inside the domain, and 0 or False if it is in the domain.

Returns **soln** (*Dataset of same type as the data.*) – Solution to the Feynman-Kac problem.

make_dirichlet_basis (*k*, *in_domain=None*, *return_evals=False*)

Creates a diffusion map basis set that obeys the homogeneous Dirichlet boundary conditions on the domain. This is done by taking the eigenfunctions of the diffusion map submatrix on the domain.

Parameters

- **k** (*int*) – Number of basis functions to create.
- **in_domain** (*1D array-like, OR list of such arrays, OR flat data format, optional*) – Array of the same shape as the data, where each element is 1 or True if that datapoint is inside the domain, and 0 or False if it is in the domain. Naturally, this must be the length as the current dataset. If None (default), all points assumed to be in the domain.
- **return_evals** (*Boolean, optional*) – Whether or not to return the eigenvalues as well. These are useful for out of sample extension.

Returns

- **basis** (*Dataset of same type as the data*) – The basis functions evaluated on each datapoint. Of the same type as the input data.
- **evals** (*1D numpy array, optional*) – The eigenvalues corresponding to each basis vector. Only returned if return_evals is True.

pyedgar.basis.**nystroem_oos**(*dmap_object*, *Y*, *evecs*, *evals*)

Performs Nystroem out-of-sample extension to calculate the values of the diffusion coordinates at each given point.

Parameters

- **dmap_object** (*DiffusionMap object*) – Diffusion map upon which to perform the out-of-sample extension.
- **Y** (*array-like, shape (n_query, n_features)*) – Data for which to perform the out-of-sample extension.

Returns **phi** (*numpy array, shape (n_query, n_eigenvectors)*) – Transformed value of the given values.

pyedgar.basis.**power_oos**(*dmap_object*, *Y*, *evecs*, *evals*)

Performs out-of-sample extension to calculate the values of the diffusion coordinates at each given point using the power-like method.

Parameters

- **dmap_object** (*DiffusionMap object*) – Diffusion map upon which to perform the out-of-sample extension.
- **Y** (*array-like, shape (n_query, n_features)*) – Data for which to perform the out-of-sample extension.

Returns **phi** (*numpy array, shape (n_query, n_eigenvectors)*) – Transformed value of the given values.

5.2 Data Manipulation

A collection of useful functions for manipulating trajectory data and dynamical basis set objects.

@author: Erik

pyedgar.data_manipulation.**delay_embed**(*traj_data*, *n_embed*, *lag=1*, *verbosity=0*)

Performs delay embedding on the trajectory data. Takes in trajectory data of format types, and returns the delay embedded data in the same type.

Parameters

- **traj_data** (*list of arrays OR tuple of two arrays OR single numpy array*) – Dynamical data on which to perform the delay embedding. This can be of multiple types, and the type dictates the format of the data. Specifically, it can be either a list of trajectories, the internal flattened format, or a single trajectory in the form of an array.
- **n_embed** (*int*) – The number of delay embeddings to perform.
- **lag** (*int, optional*) – The number of timesteps to look back in time for each delay. Default is 1.
- **verbosity** (*int, optional*) – The level of status messages that are output. Default is 0 (no messages).

Returns **embedded_data** (*list of arrays OR tuple of two arrays OR single numpy array*) – Dynamical data with delay embedding performed, of the same type as the trajectory data.

pyedgar.data_manipulation.**flat_to_tlist**(*traj_2d*, *traj_edges*)
Takes a flattened trajectory with stop and start points and reformats it into a list of separate trajectories.

Parameters

- **traj2D** (*2D numpy array*) – Numpy array containing the flattened trajectory information.
- **traj_edges** (*1D numpy array*) – Numpy array where each element is the start of each trajectory: the n'th trajectory runs from traj_edges[n] to traj_edges[n+1]

Returns **trajs** (*list of array-likes*) – List where each element n is a array-like object of shape N_n x d, where N_n is the number of data points in that trajectory and d is the number of coordinates for each datapoint.

pyedgar.data_manipulation.**get_initial_final_split**(*traj_edges*, *lag=1*)
Returns the incides of the points in the flat trajectory of the initial and final sample points. In this context, initial means the first N-lag points, and final means the last N-lag points.

Parameters **lag** (*int, optional*) – Number of timepoints in the future to look into the future for the transfer operator. Default is 1.

Returns

- **t_0_indices** (*1D numpy array*) – Indices in the flattened trajectory data of all the points at the initial times.
- **t_0_indices** (*1D numpy array*) – Indices in the flattened trajectory data of all the points at the final times.

pyedgar.data_manipulation.**lift_function**(*function*, *n_embed*, *lag=1*)
Lift a function into the delay-embedded space.

pyedgar.data_manipulation.**tlist_to_flat**(*trajs*)
Flattens a list of two dimensional trajectories into a single two dimensional datastructure, and returns it along with a list of tuples giving the locations of each trajectory.

Parameters **trajs** (*list of array-likes*) – List where each element n is a array-like object of shape N_n x d, where N_n is the number of data points in that trajectory and d is the number of coordinates for each datapoint.

Returns

- **traj2D** (*2D numpy array*) – Numpy array containing the flattened trajectory information.
- **traj_edges** (*1D numpy array*) – Numpy array where each element is the start of each trajectory: the n'th trajectory runs from traj_edges[n] to traj_edges[n+1]

5.3 Galerkin

Routines for constructing estimates of dynamical quantities on trajectory data using Galerkin expansion.

@author: Erik

pyedgar.galerkin.**compute_FK**(*basis*, *h*, *r=None*, *lag=1*, *dt=1.0*, *return_coeffs=False*)
Solves the forward Feynman-Kac problem Lg=h on a domain D, with boundary conditions g=b on the complement of D. To account for the boundary conditions, we solve the homogeneous problem Lg = h - Lr, where r is the provided guess.

Parameters

- **traj_data** (*list of arrays OR single numpy array*) – Value of the basis functions at every time point. Should only be nonzero for points on the domain.
- **h** (*list of 1d arrays or single 1d array*) – Value of the RHS of the FK formula. This should only be nonzero at points on the domain, Domain.
- **r** (*list of 1d arrays or single 1d array, optional*) – Value of the guess function. Should be equal to b every point off of the domain. IF not provided, the boundary conditions are assumed to be homogeneous.
- **lag** (*int*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator. If not provided, defaults to 1.
- **timestep** (*scalar, optional*) – Time between timepoints in the trajectory data. Defaults to 1.

Returns

- **g** (*list of arrays*) – Estimated solution to the Feynman-Kac problem.
- **coeffs** (*ndarray*) – Coefficients for the solution, only returned if return_coeffs is True.

pyedgar.galerkin.**compute_adj_FK**(*basis*, *h*, *com=None*, *r=None*, *lag=1*, *dt=1.0*, *return_coeffs=False*)

Solves the Feynman-Kac problem L^t dagger g=h on a domain D, with boundary conditions g=b on the complement of D. Here L^t is the adjoint of the generator with respect to the provided change of measure. To account for the boundary conditions, we solve the homogeneous problem L^t g = h - L^t r, where r is the provided guess.

Parameters

- **traj_data** (*list of arrays OR single numpy array*) – Value of the basis functions at every time point. Should only be nonzero for points on the domain.
- **h** (*list of 1d arrays or single 1d array*) – Value of the RHS of the FK formula. This should only be nonzero at points on the domain, Domain.
- **com** (*list of 1d arrays or single 1d array, optional*) – Values of the change of measure against which to take the desired adjoint. If not provided, takes the adjoint against the sampled meaure
- **r** (*list of 1d arrays or single 1d array, optional*) – Value of the guess function. Should be equal to b every point off of the domain. If not provided, the boundary conditions are assumed to be homogeneous.
- **lag** (*int*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator. If not provided, defaults to 1.
- **timestep** (*scalar, optional*) – Time between timepoints in the trajectory data. Defaults to 1.

Returns

- **g** (*list of arrays*) – Estimated solution to the Feynman-Kac problem.
- **coeffs** (*ndarray*) – Coefficients for the solution, only returned if return_coeffs is True.

pyedgar.galerkin.**compute_bwd_committor**(*basis*, *guess_committor*, *stationary_com*, *lag=1*)

Calculates the backward into state A as a function of each point.

Parameters

- **basis** (*list of trajectories*) – Basis for the Galerkin expansion. Must be zero in state A and B
- **guess_committor** (*list of trajectories, optional*) – The value of the guess function obeying the inhomogenous boundary conditions.
- **stationary_com** (*list of trajectories*) – Values of the change of measure to the stationary distribution.

- **lag** (*int, optional*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator.

Returns **bwd_committor** (*dynamical basis object*) – List of trajectories containing the values of the backward_committor at each point.

pyedgar.galerkin.**compute_change_of_measure**(*basis*, *lag=1*)

Calculates the value of the change of measure to the stationary distribution for each datapoint.

Parameters

- **basis** (*list of trajectories*) – Basis for the Galerkin expansion. Must be zero in state A and B
- **lag** (*int, optional*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator.

Returns **change_of_measure** (*dynamical basis object*) – List of trajectories containing the values of the change of measure to the stationary distribution at each point.

pyedgar.galerkin.**compute_committor**(*basis*, *guess_committor*, *lag=1*)

Calculates the forward committor into state A as a function of each point.

Parameters

- **basis** (*list of trajectories*) – Basis for the Galerkin expansion. Must be zero in state A and B
- **guess_committor** (*list of trajectories, optional*) – The value of the guess function obeying the inhomogenous boundary conditions.
- **lag** (*int, optional*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator.

Returns **committor** (*dynamical basis object*) – List of trajectories containing the values of the forward committor at each point.

pyedgar.galerkin.**compute_correlation_mat**(*Xs*, *Ys=None*, *lag=1*, *com=None*)

Computes the time-lagged correlation matrix between two sets of observables.

Parameters

- **Xs** (*list of trajectories*) – List of trajectories for the first set of observables.
- **Ys** (*list of trajectories, optional*) – List of trajectories for the second set of observables. If None, set to be X.
- **lag** (*int, optional*) – Lag to use in the correlation matrix. Default is one step.
- **com** (*list of trajectories*) – Values of the change of measure against which to compute the average

Returns **K** (*numpy array*) – The time-lagged correlation matrix between X and Y.

pyedgar.galerkin.**compute_esystem**(*basis*, *lag=1*, *dt=1.0*, *left=False*, *right=True*)

Calculates the eigenvectors and eigenvalues of the generator through Galerkin expansion.

Parameters

- **basis** (*list of trajectories*) – List of trajectories containing the basis for the Galerkin expansion. This method works much better if the basis set is zero on states A and B, however this is not a necessity.
- **lag** (*int, optional*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator.
- **left** (*bool, optional*) – Whether or not to calculate the left eigenvectors of the system.

- **right** (*bool, optional*) – Whether or not to calculate the right eigenvectors of the system.

Returns

- **eigenvalues** (*numpy array*) – Numpy array containing the eigenvalues of the generator.
- **left_eigenvectors** (*list of trajectories, optional*) – If left was set to true, the left eigenvectors are returned as a list of trajectories.
- **right_eigenvectors** (*list of trajectories, optional*) – If right was set to true, the right eigenvectors are returned as a list of trajectories.

pyedgar.galerkin.**compute_generator**(*Xs*, *Ys=None*, *lag=1*, *dt=1.0*, *com=None*)

Computes the matrix of inner product elements against the generator.

Parameters

- **Xs** (*list of trajectories*) – List of trajectories for the first set of observables.
- **Ys** (*list of trajectories, optional*) – List of trajectories for the second set of observables. If None, set to be X.
- **lag** (*int, optional*) – Lag to use in the correlation matrix. Default is one step.
- **dt** (*float, optional*) – time per step of dynamics. Default is one time unit.
- **com** (*list of trajectories*) – Values of the change of measure against which to compute the average.

Returns **L** (*numpy array*) – The approximation to the inner product <X, L Y>.

pyedgar.galerkin.**compute_mfpt**(*basis*, *stateA*, *lag=1*, *dt=1.0*)

Calculates the mean first passage time into state A as a function of each point.

Parameters

- **basis** (*list of trajectories*) – Basis for the Galerkin expansion. Must be zero in state A.
- **state_A** (*list of trajectories*) – List of trajectories where each element is 1 or 0, corresponding to whether or not the datapoint is in state A.
- **lag** (*int, optional*) – Number of timepoints in the future to use for the finite difference in the discrete-time generator. If not provided, uses value in the generator.
- **timestep** (*scalar, optional*) – Time between timepoints in the trajectory data.

Returns **mfpt** (*list of trajectories*) – List of trajectories containing the values of the mean first passage time at each timepoint.

pyedgar.galerkin.**compute_stiffness_mat**(*Xs*, *Ys=None*, *lag=1*, *com=None*)

Computes the stiffness matrix between two sets of observables.

Parameters

- **Xs** (*list of trajectories*) – List of trajectories for the first set of observables.
- **Ys** (*list of trajectories, optional*) – List of trajectories for the second set of observables. If None, set to be X.
- **lag** (*int, optional*) – Lag to use in the correlation matrix. Default is one step. This is required as the stiffness is only evaluated over the initial points.
- **com** (*list of trajectories*) – Values of the change of measure against which to compute the average

Returns **S** (*numpy array*) – The time-lagged stiffness matrix between X and Y.

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

6.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.2 Documentation improvements

PyEDGAR could always use more documentation, whether as part of the official PyEDGAR docs, in docstrings, or even on the web in blog posts, articles, and such.

6.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/ehthiede/PyEDGAR/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

6.4 Development

To set up *PyEDGAR* for local development:

1. Fork PyEDGAR (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/PyEDGAR.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

6.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

6.4.2 Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 7

Authors

- Erik Henning Thiede - NA

CHAPTER 8

Changelog

8.1 0.2.0 (2019-02-25)

- Abandoned Dataset object and moved to list of trajectories convention.
- Removed dataset interface, moved to list of trajectories
- Factorized code to use generic solvers for FK formulas
- Added support for calculation of backwards committors
- Added jupyter notebook tutorials in the examples
- Updated Documentation

8.2 0.1.0 (2017-09-19)

- First release.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index