pydiffmap

Release 0.1.0

Jan 30, 2019

Contents

This is the home of the documentation for pyDiffMap, an open-source project to develop a robust and accessible diffusion map code for public use. Code can be found on our github page. Our documentation is currently under construction, please bear with us.

CHAPTER 1

Contents

1.1 Overview

docs	
tests	

This is the home of the documentation for pyDiffMap, an open-source project to develop a robust and accessible diffusion map code for public use. Our documentation is currently under construction, please bear with us.

- Free software: GPL. If you need to use this software in a non-GPL compatible environment, please contact the authors and we can work something out.

1.1.1 Installation

Pydiffmap is installable using pip. You can install it using the command

```
pip install pyDiffMap
```

In the meantime, you can install the package directly from the source directly by downloading the package from github and running the command below, optionally with the "-e" flag for an editable install.

```
pip install [source_directory]
```

1.1.2 Documentation

https://pyDiffMap.readthedocs.io/

1.1.3 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`

If you don't have tox installed, you can also run the python tests directly with

```
pytest
```

1.2 Installation

At the command line:

```
pip install [source_dir]
```

1.3 Theory

Diffusion maps is a dimension reduction technique that can be used to discover low dimensional structure in high dimensional data. It assumes that the data points, which are given as points in a high dimensional metric space, actually live on a lower dimensional structure. To uncover this structure, diffusion maps builds a neighborhood graph on the data based on the distances between nearby points. Then a graph Laplacian $\mathbf{L}$ is constructed on the neighborhood graph. Many variants exist that approximate different differential operators. For example, *standard* diffusion maps approximates the differential operator

$$\mathcal{L}f = \Delta f - 2(1-\alpha)\nabla f \cdot \frac{\nabla q}{q}$$

where Δ is the Laplace Beltrami operator, ∇ is the gradient operator and q is the sampling density. The normalization parameter α, which is typically between 0.0 and 1.0, determines how much q is allowed to bias the operator $\mathcal{L}$. Standard diffusion maps on a dataset `X`, which has to given as a numpy array with different rows corresponding to different observations, is implemented in pydiffmap as:

```
mydmap = diffusion_map.DiffusionMap(epsilon = my_epsilon, alpha = my_alpha)
mydmap.fit(X)
```

Here `epsilon` is a scale parameter used to rescale distances between data points. We can also choose `epsilon` automatically due to an an algorithm by Berry, Harlim and Giannakis:

```
mydmap = dm.DiffusionMap(alpha = my_alpha, epsilon = 'bgh')
```

For additional optional arguments of the DiffusionMap class, see usage and documentation.

A variant of diffusion maps, 'TMDmap', unbiases with respect to q and approximates the differential operator

$$\mathcal{L}f = \Delta f + \nabla(\log \pi) \cdot \nabla f$$

where π is a 'target distribution' that defines the drift term and has to be known up to a normalization constant. TMDmap is implemented in pydiffmap as:

```
mydmap = diffusion_map.DiffusionMap(epsilon = my_epsilon, alpha = 1.0)
mydmap.fit(X, weights = myDistribution)
```

where `myDistribution` is an array that represents the target distribution π evaluated at the data points `X`.

1.4 Usage

To use pyDiffMap in a project:

```
import pyDiffMap
```

To initialize a diffusion map object:

```
mydmap = diffusion_map.DiffusionMap(n_evecs = 1, epsilon = 1.0, alpha = 0.5, k=64)
```

where `n_evecs` is the number of eigenvectors that are computed, `epsilon` is a scale parameter used to rescale distances between data points, `alpha` is a normalization parameter (typically between 0.0 and 1.0) that influences the effect of the sampling density, and `k` is the number of nearest neighbors considered when the kernel is computed. A larger `k` means increased accuracy but larger computation time. For additional optional arguments, see documentation.

We can also employ automatic epsilon detection due to an algorithm by Berry, Harlim and Giannakis:

```
mydmap = dm.DiffusionMap(n_evecs = 1, alpha = 0.5, epsilon = 'bgh', k=64)
```

To fit to a dataset `X` (array-like, shape (n_query, n_features)):

```
mydmap.fit(X)
```

The diffusion map coordinates can also be accessed directly via:

```
dmap = mydmap.fit_transform(X)
```

This returns an array `dmap` with shape (n_query, n_evecs). E.g. `dmap[:,0]` is the first diffusion coordinate evaluated on the data `X`.

In order to compute diffusion coordinates at the out of sample location(s) `Y`:

```
dmap_Y = mydmap.transform(Y)
```

1.5 Jupyter notebook tutorials

1.5.1 The classic swiss roll data set

author: Ralf Banisch

We demonstrate the usage of the diffusion_map class on a two-dimensional manifold embedded in $\mathbb{R}^3$.

```
# import some necessary functions for plotting as well as the diffusion_map class␣
↪from pydiffmap.
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d import Axes3D
from pydiffmap import diffusion_map as dm

%matplotlib inline
```

Create Data

We create the dataset: A noisy sampling of the twodimensional "swiss roll" embedded in $\mathbb{R}^3$. The sampling is such that the density of samples decreases with the distance from the origin (non-uniform sampling).

In order to be handled correctly by the diffusion_map class, we must ensure the data is a numpy array of shape (n_points, n_features).

```
# set parameters
length_phi = 15   #length of swiss roll in angular direction
length_Z = 15     #length of swiss roll in z direction
sigma = 0.1       #noise strength
m = 10000         #number of samples

# create dataset
phi = length_phi*np.random.rand(m)
xi = np.random.rand(m)
Z = length_Z*np.random.rand(m)
X = 1./6*(phi + sigma*xi)*np.sin(phi)
Y = 1./6*(phi + sigma*xi)*np.cos(phi)

swiss_roll = np.array([X, Y, Z]).transpose()

# check that we have the right shape
print(swiss_roll.shape)
```

```
(10000, 3)
```

Run pydiffmap

Now we initialize the diffusion map object and fit it to the dataset. Since we are interested in only the first two diffusion coordinates we set n_evecs = 2, and since we want to unbias with respect to the non-uniform sampling density we set alpha = 1.0. The epsilon parameter controls the scale and needs to be adjusted to the data at hand. The k parameter controls the neighbour lists, a smaller k will increase performance but decrease accuracy.

```
# initialize Diffusion map object.
neighbor_params = {'n_jobs': -1, 'algorithm': 'ball_tree'}

mydmap = dm.DiffusionMap(n_evecs=2, k=200, epsilon='bgh', alpha=1.0, neighbor_
↪params=neighbor_params)
# fit to data and return the diffusion map.
dmap = mydmap.fit_transform(swiss_roll)
```

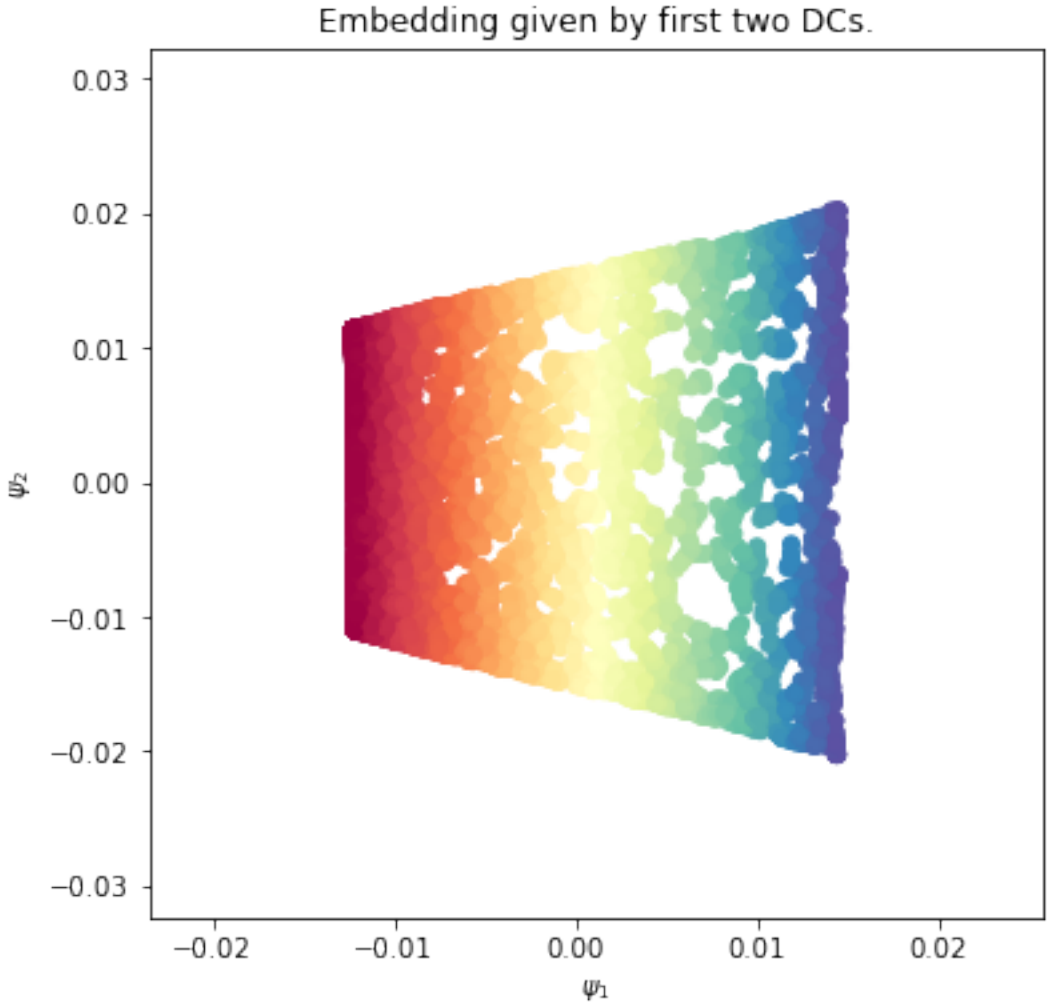
```
mydmap.epsilon_fitted
```

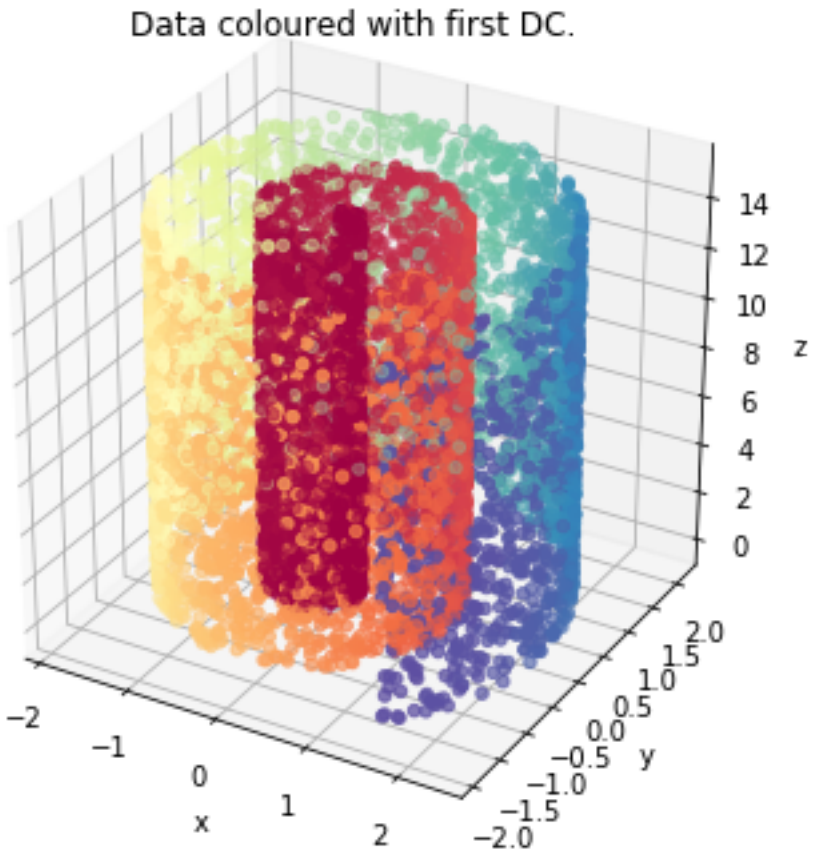
```
0.0625
```

Visualization

We show the original data set on the right, with points colored according to the first diffusion coordinate. On the left, we show the diffusion map embedding given by the first two diffusion coordinates. Points are again colored according to the first diffusion coordinate, which seems to parameterize the ϕ direction. We can see that the diffusion map embedding 'unwinds' the swiss roll.

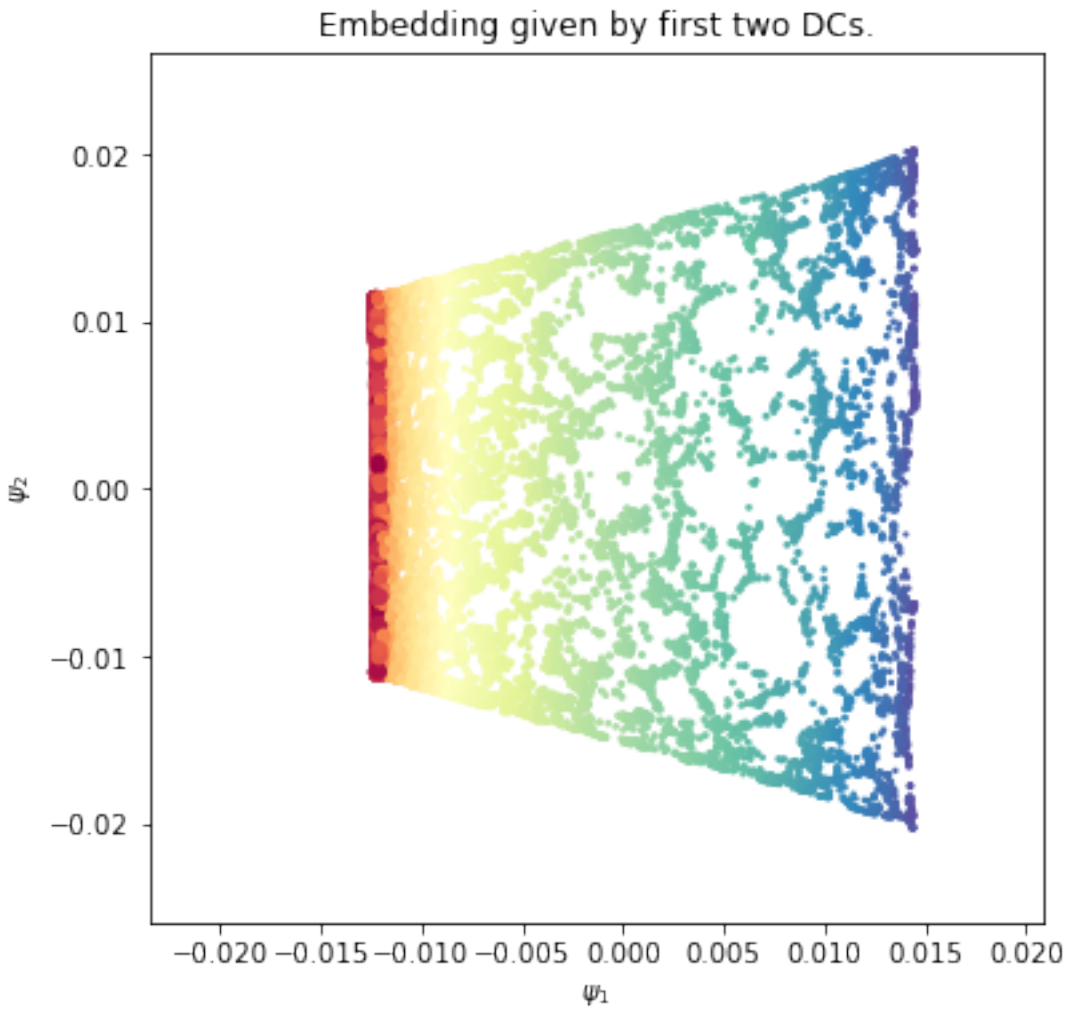
```
from pydiffmap.visualization import embedding_plot, data_plot

embedding_plot(mydmap, scatter_kwargs = {'c': dmap[:,0], 'cmap': 'Spectral'})
data_plot(mydmap, dim=3, scatter_kwargs = {'cmap': 'Spectral'})

plt.show()
```

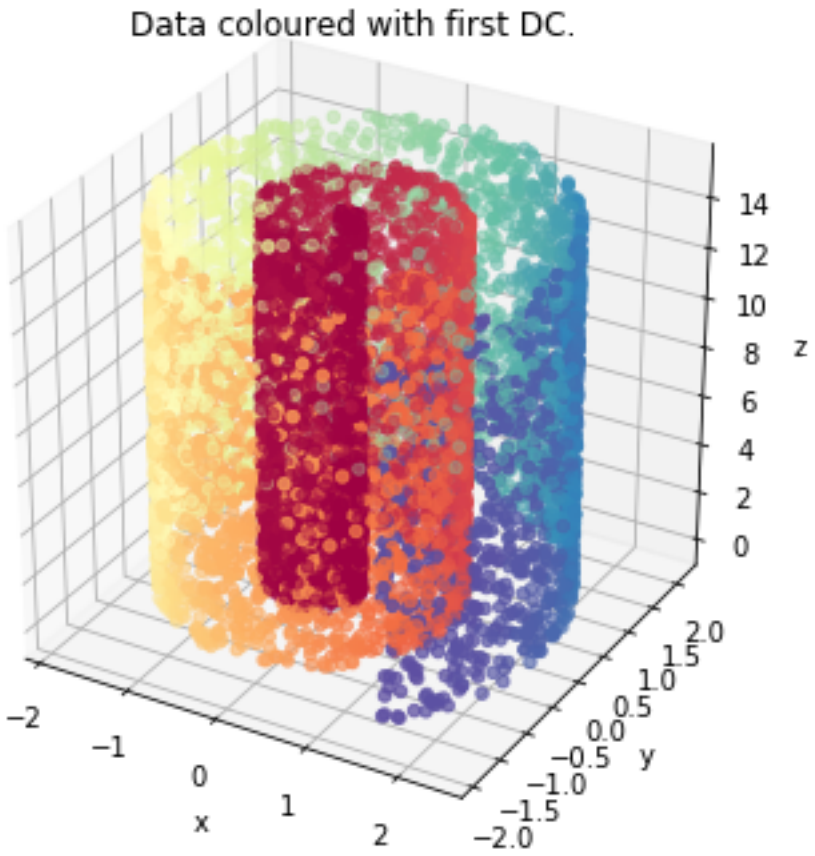




To get a bit more information out of the embedding, we can scale the points according to the numerical estimate of the sampling density (mydmap.q), and color them according to their location in the phi direction. For comparison, we color the original data set according to ϕ this time.

```
from pydiffmap.visualization import embedding_plot, data_plot

embedding_plot(mydmap, scatter_kwargs = {'c': phi, 's': mydmap.q, 'cmap': 'Spectral'})
data_plot(mydmap, dim=3, scatter_kwargs = {'cmap': 'Spectral'})
plt.show()
```

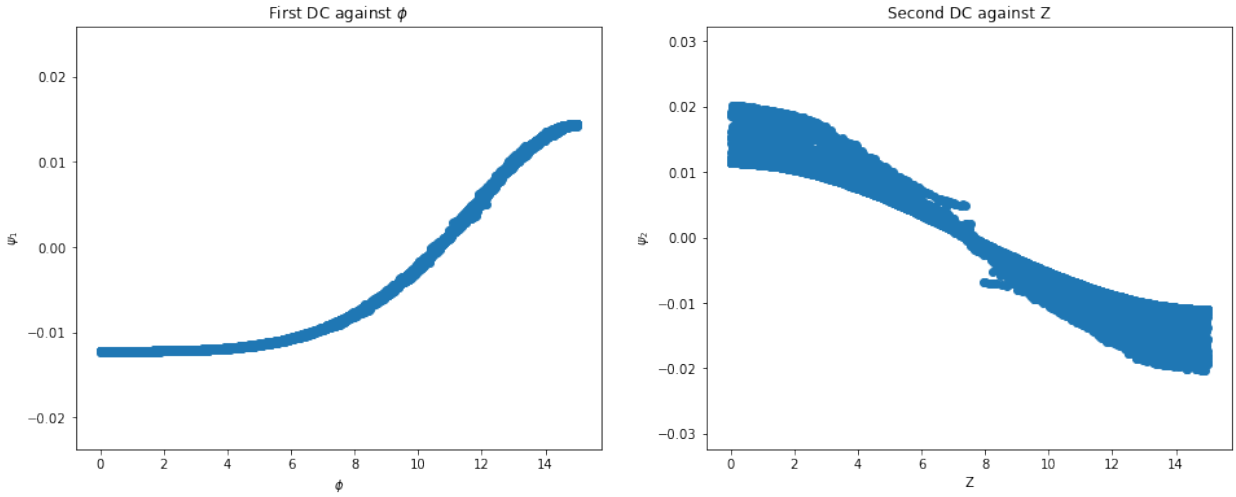

Embedding given by first two DCs.
0.02
0.01
0.00
−0.01
−0.02
ψ2
−0.020 −0.015 −0.010 −0.005 0.000 0.005 0.010 0.015 0.020
ψ1



We can see that points near the center of the swiss roll, where the winding is tight, are closer together in the embedding, while points further away from the center are more spaced out. Let's check how the first two diffusion coordinates correlate with ϕ and Z.

```
print('Correlation between \phi and \psi_1')
print(np.corrcoef(dmap[:,0], phi))

plt.figure(figsize=(16,6))
ax = plt.subplot(121)
ax.scatter(phi, dmap[:,0])
ax.set_title('First DC against $\phi$')
ax.set_xlabel(r'$\phi$')
ax.set_ylabel(r'$\psi_1$')
ax.axis('tight')

print('Correlation between Z and \psi_2')
print(np.corrcoef(dmap[:,1], Z))

ax2 = plt.subplot(122)
ax2.scatter(Z, dmap[:,1])
ax2.set_title('Second DC against Z')
ax2.set_xlabel('Z')
ax2.set_ylabel(r'$\psi_2$')

plt.show()
```

```
Correlation between phi and psi_1
[[ 1.          0.92202403]
 [ 0.92202403  1.        ]]
Correlation between Z and psi_2
[[ 1.         -0.96896173]
 [-0.96896173  1.        ]]
```



1.5.2 Spherical Harmonics

In this notebook we try to reproduce the eigenfunctions of the Laplacian on the 2D sphere embedded in $\mathbb{R}^3$. The eigenfunctions are the spherical harmonics $Y_l^m(\theta, \phi)$.

```
import numpy as np

from pydiffmap import diffusion_map as dm
from scipy.sparse import csr_matrix

np.random.seed(100)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
%matplotlib inline
```

generate data on a Sphere

we sample longitude and latitude uniformly and then transform to $\mathbb{R}^3$ using geographical coordinates (latidude is measured from the equator).

```
m = 10000
Phi = 2*np.pi*np.random.rand(m) - np.pi
Theta = np.pi*np.random.rand(m) - 0.5*np.pi
X = np.cos(Theta)*np.cos(Phi)
Y = np.cos(Theta)*np.sin(Phi)
Z = np.sin(Theta)
data = np.array([X, Y, Z]).transpose()
```

run diffusion maps

Now we initialize the diffusion map object and fit it to the dataset. We set n_evecs = 4, and since we want to unbias with respect to the non-uniform sampling density we set alpha = 1.0. The epsilon parameter controls the scale and is set here by hand. The k parameter controls the neighbour lists, a smaller k will increase performance but decrease accuracy.

```
eps = 0.01
mydmap = dm.DiffusionMap(n_evecs=4, epsilon=eps, alpha=1.0, k=400)
mydmap.fit_transform(data)
test_evals = -4./eps*(mydmap.evals - 1)
print(test_evals)
```

```
[ 1.87120055  1.89337561  1.91161358  5.58725164]
```

The true eigenfunctions here are spherical harmonics $Y_l^m(\theta, \phi)$ and the true eigenvalues are $\lambda_l = l(l+1)$. The eigenfunction corresponding to $l = 0$ is the constant function, which we ommit. Since $l = 1$ has multiplicity three, this gives the benchmark eigenvalues [2, 2, 2, 6].

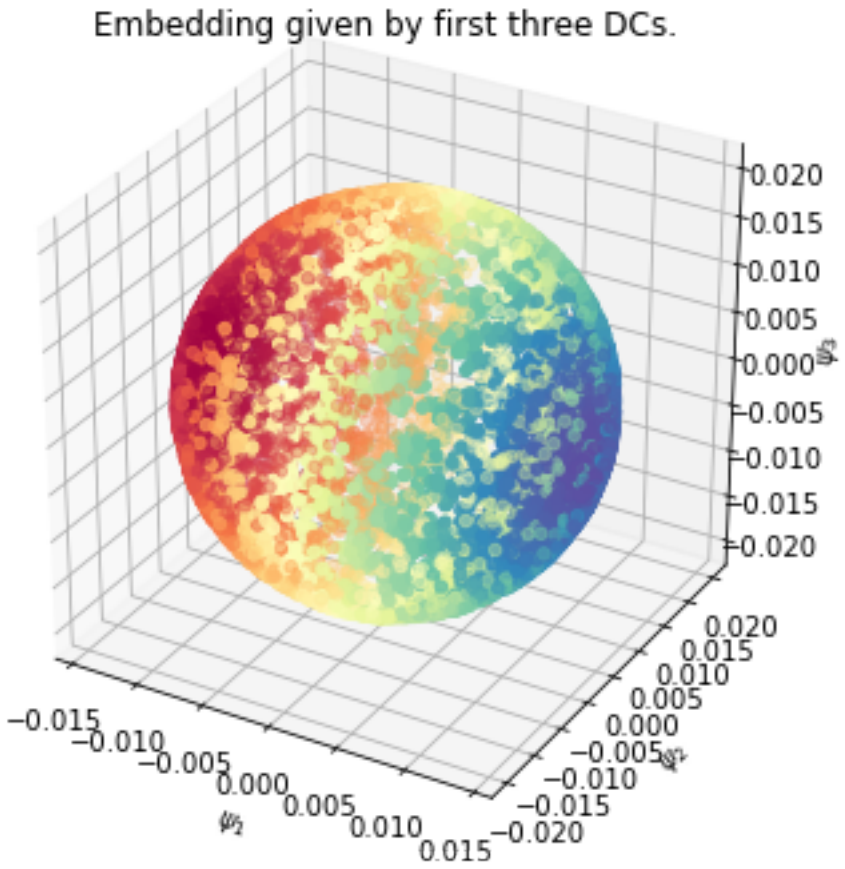
```
real_evals = np.array([2, 2, 2, 6])
test_evals = -4./eps*(mydmap.evals - 1)
eval_error = np.abs(test_evals-real_evals)/real_evals
print(test_evals)
print(eval_error)
```

```
[ 1.87120055  1.89337561  1.91161358  5.58725164]
[ 0.06439973  0.05331219  0.04419321  0.06879139]
```

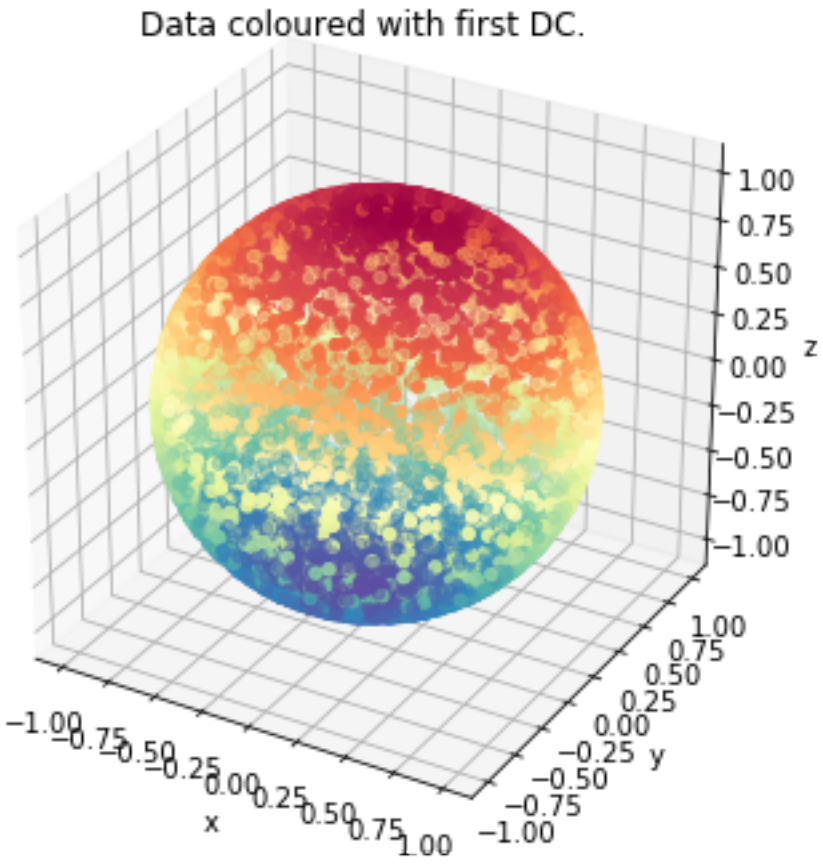
visualisation

With pydiffmap's visualization toolbox, we can get a quick look at the embedding produced by the first two diffusion coordinates and the data colored by the first eigenfunction.

```
from pydiffmap.visualization import embedding_plot, data_plot

embedding_plot(mydmap, dim=3, scatter_kwargs = {'c': mydmap.dmap[:,0], 'cmap':
↪'Spectral'})

plt.show()
```




```
data_plot(mydmap, dim=3, scatter_kwargs = {'cmap': 'Spectral'})
plt.show()
```



Rotating the dataset

There is rotational symmetry in this dataset. To remove it, we define the 'north pole' to be the point where the first diffusion coordinate attains its maximum value.

```
northpole = np.argmax(mydmap.dmap[:,0])
north = data[northpole,:]
phi_n = Phi[northpole]
theta_n = Theta[northpole]
R = np.array([[np.sin(theta_n)*np.cos(phi_n), np.sin(theta_n)*np.sin(phi_n), -np.
↪cos(theta_n)],
              [-np.sin(phi_n), np.cos(phi_n), 0],
             [np.cos(theta_n)*np.cos(phi_n), np.cos(theta_n)*np.sin(phi_n), np.
↪sin(theta_n)]])
```

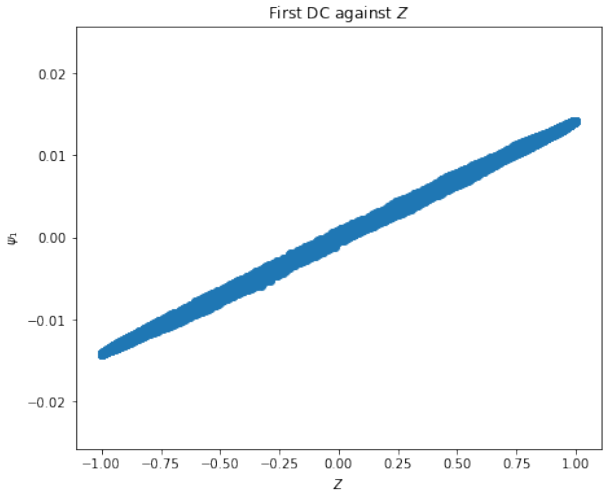
```
data_rotated = np.dot(R,data.transpose())
data_rotated.shape
```

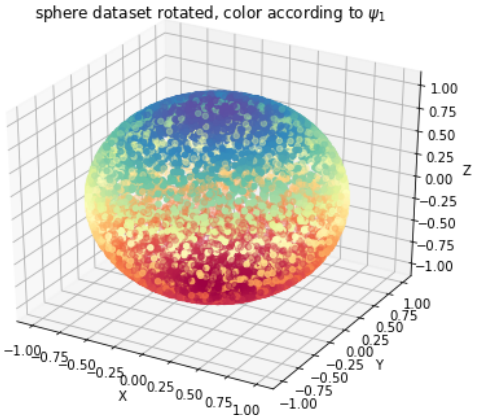
```
(3, 10000)
```

Now that the dataset is rotated, we can check how well the first diffusion coordinate approximates the first spherical harmonic $Y_1^1(\theta, \phi) = \sin(\theta) = Z$.

```
print('Correlation between \phi and \psi_1')
print(np.corrcoef(mydmap.dmap[:,0], data_rotated[2,:]))

plt.figure(figsize=(16,6))
ax = plt.subplot(121)
ax.scatter(data_rotated[2,:], mydmap.dmap[:,0])
ax.set_title('First DC against $Z$')
ax.set_xlabel(r'$Z$')
ax.set_ylabel(r'$\psi_1$')
ax.axis('tight')

ax2 = plt.subplot(122,projection='3d')
ax2.scatter(data_rotated[0,:],data_rotated[1,:],data_rotated[2,:], c=mydmap.dmap[:,0],
↪ cmap=plt.cm.Spectral)
#ax2.view_init(75, 10)
ax2.set_title('sphere dataset rotated, color according to $\psi_1$')
ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_zlabel('Z')

plt.show()
```

```
Correlation between phi and psi_1
[[ 1.          0.99939606]
 [ 0.99939606  1.        ]]
```





1.5.3 2D Four-well potential

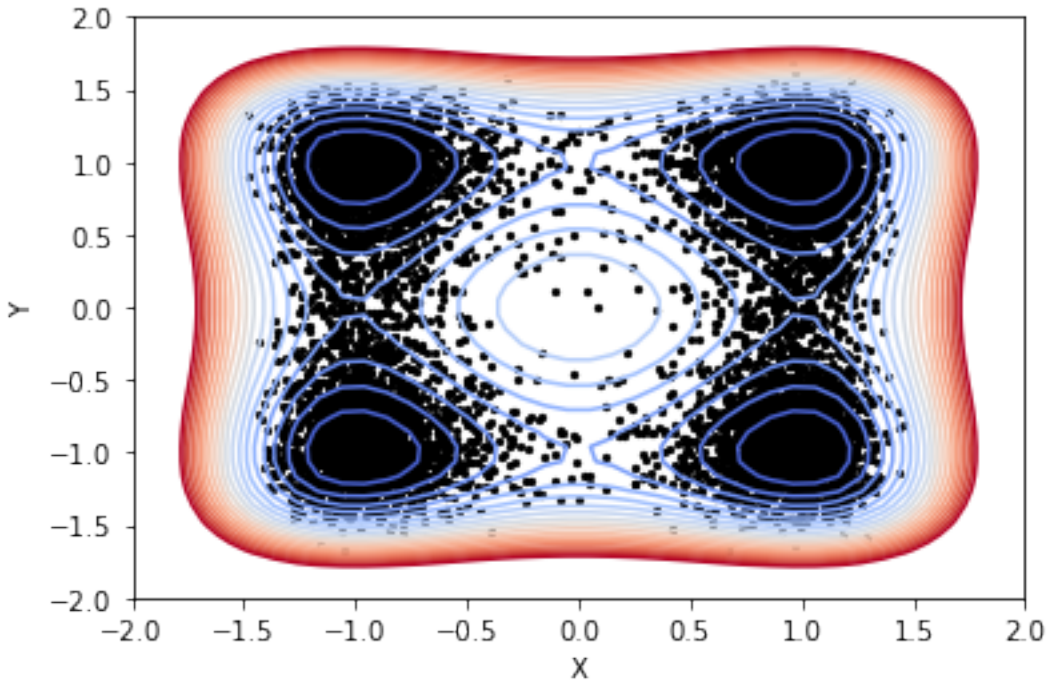
```
import matplotlib.pyplot as plt
import numpy as np

from mpl_toolkits.mplot3d import Axes3D
from pydiffmap import diffusion_map as dm

%matplotlib inline
```

Load sampled data: discretized Langevin dynamics at temperature T=1, friction 1, and time step size dt=0.01, with double-well potentials in x and y, with higher barrier in y.

```
X=np.load('Data/4wells_traj.npy')
print(X.shape)
```

```
(9900, 2)
```

```
def DW1(x):
        return 2.0*(np.linalg.norm(x)**2-1.0)**2

def DW2(x):
        return 4.0*(np.linalg.norm(x)**2-1.0)**2

def DW(x):
    return DW1(x[0]) + DW1(x[1])

from matplotlib import cm

mx=5

xe=np.linspace(-mx, mx, 100)
ye=np.linspace(-mx, mx, 100)
energyContours=np.zeros((100, 100))
for i in range(0,len(xe)):
            for j in range(0,len(ye)):
                xtmp=np.array([xe[i], ye[j]] )
                energyContours[j,i]=DW(xtmp)

levels = np.arange(0, 10, 0.5)
plt.contour(xe, ye, energyContours, levels, cmap=cm.coolwarm)
plt.scatter(X[:,0], X[:,1], s=5, c='k')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim([-2,2])
plt.ylim([-2,2])
plt.show()
```



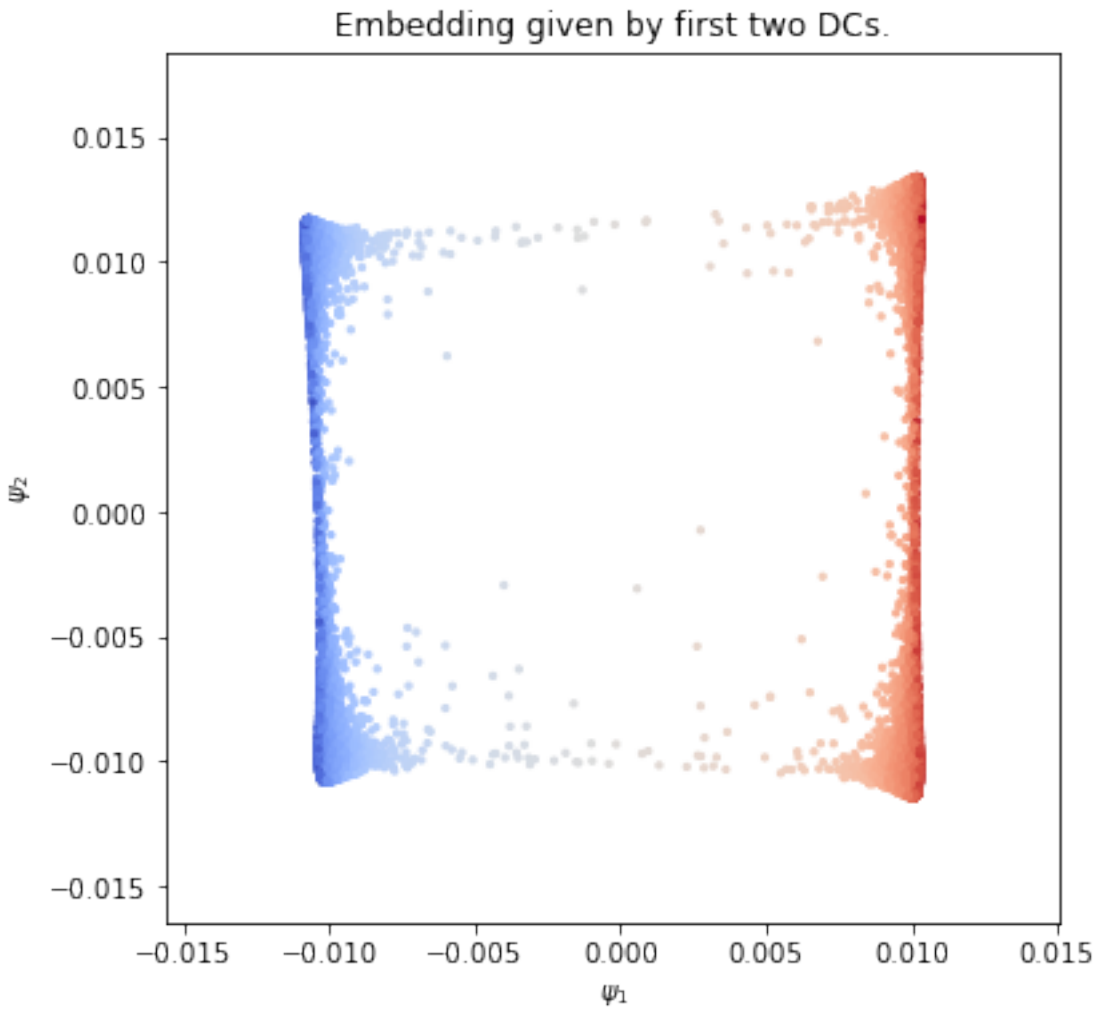
Compute diffusion map embedding

```
mydmap = dm.DiffusionMap(n_evecs = 2, epsilon = .2, alpha = 0.5, k=400, metric=
↪'euclidean')
dmap = mydmap.fit_transform(X)
```

Visualization

We plot the first two diffusion coordinates against each other, colored by the x coordinate

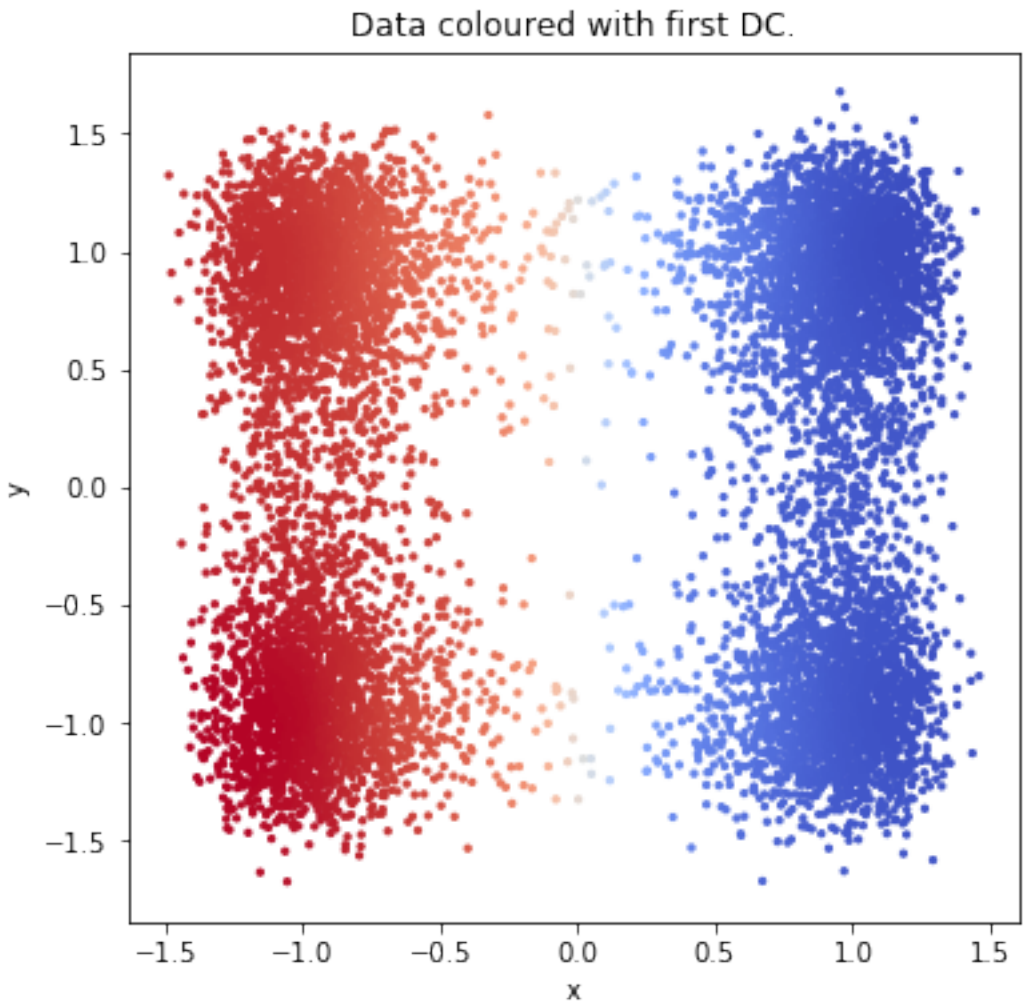
```
from pydiffmap.visualization import embedding_plot

embedding_plot(mydmap, scatter_kwargs = {'c': X[:,0], 's': 5, 'cmap': 'coolwarm'})

plt.show()
```




```
#from matplotlib import cm
#plt.scatter(dmap[:,0], dmap[:,1], c=X[:,0], s=5, cmap=cm.coolwarm)

#clb=plt.colorbar()
#clb.set_label('X coordinate')
#plt.xlabel('First dominant eigenvector')
#plt.ylabel('Second dominant eigenvector')
#plt.title('Diffusion Map Embedding')

#plt.show()
```

We visualize the data again, colored by the first eigenvector this time.

```
from pydiffmap.visualization import data_plot

data_plot(mydmap, scatter_kwargs = {'s': 5, 'cmap': 'coolwarm'})
plt.show()
```

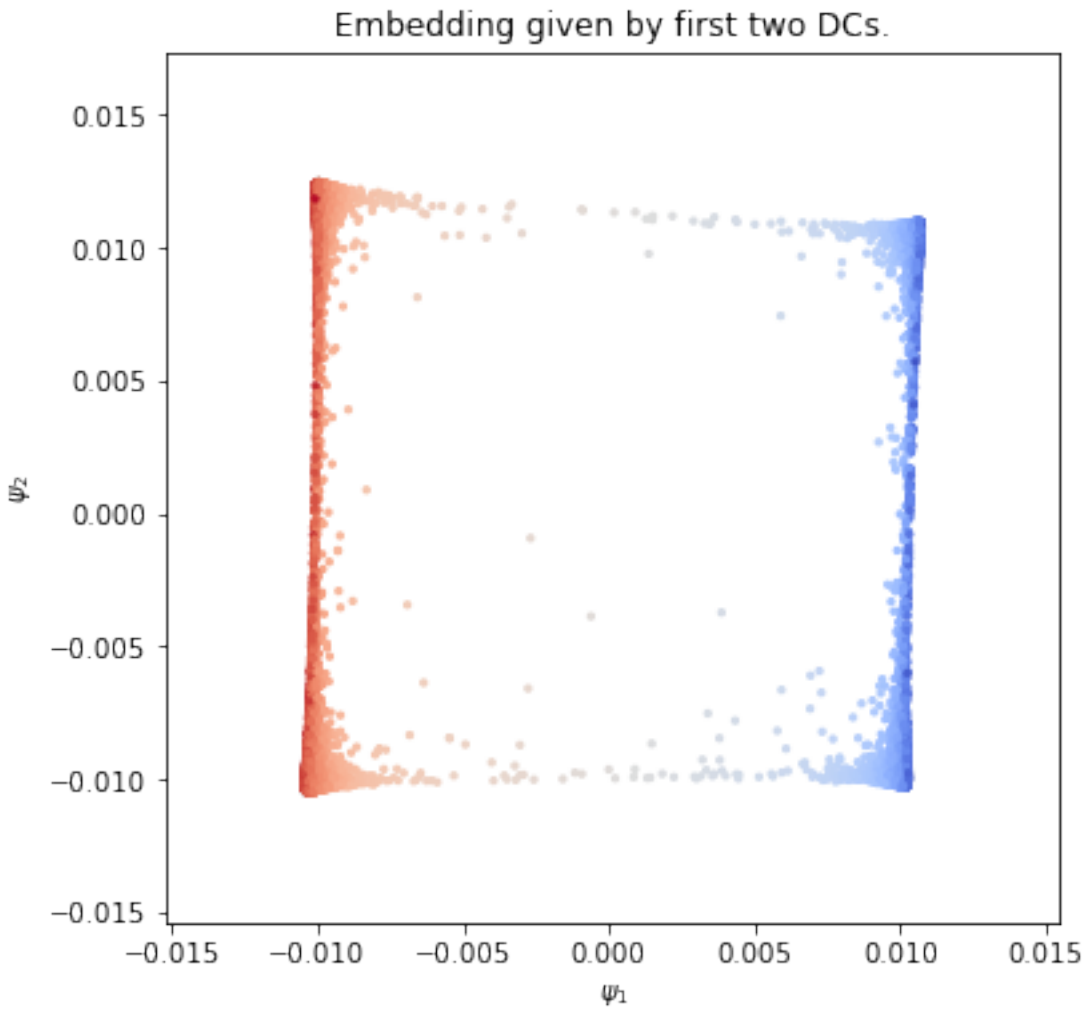


Target measure diffusion map

Compute Target Measure Diffusion Map with target distribution pi(q) = exp(-beta V(q)) with inverse temperature beta = 1. TMDmap can be seen as a special case where the weights are the target distribution, and alpha=1.

```
V=DW
beta=1
target_distribution=np.zeros(len(X))
for i in range(len(X)):
    target_distribution[i]=np.exp(-beta*V(X[i]))
mytdmap = dm.DiffusionMap(alpha=1.0, n_evecs = 2, epsilon = .2, k=400)
tmdmap = mytdmap.fit_transform(X, weights=target_distribution)
```

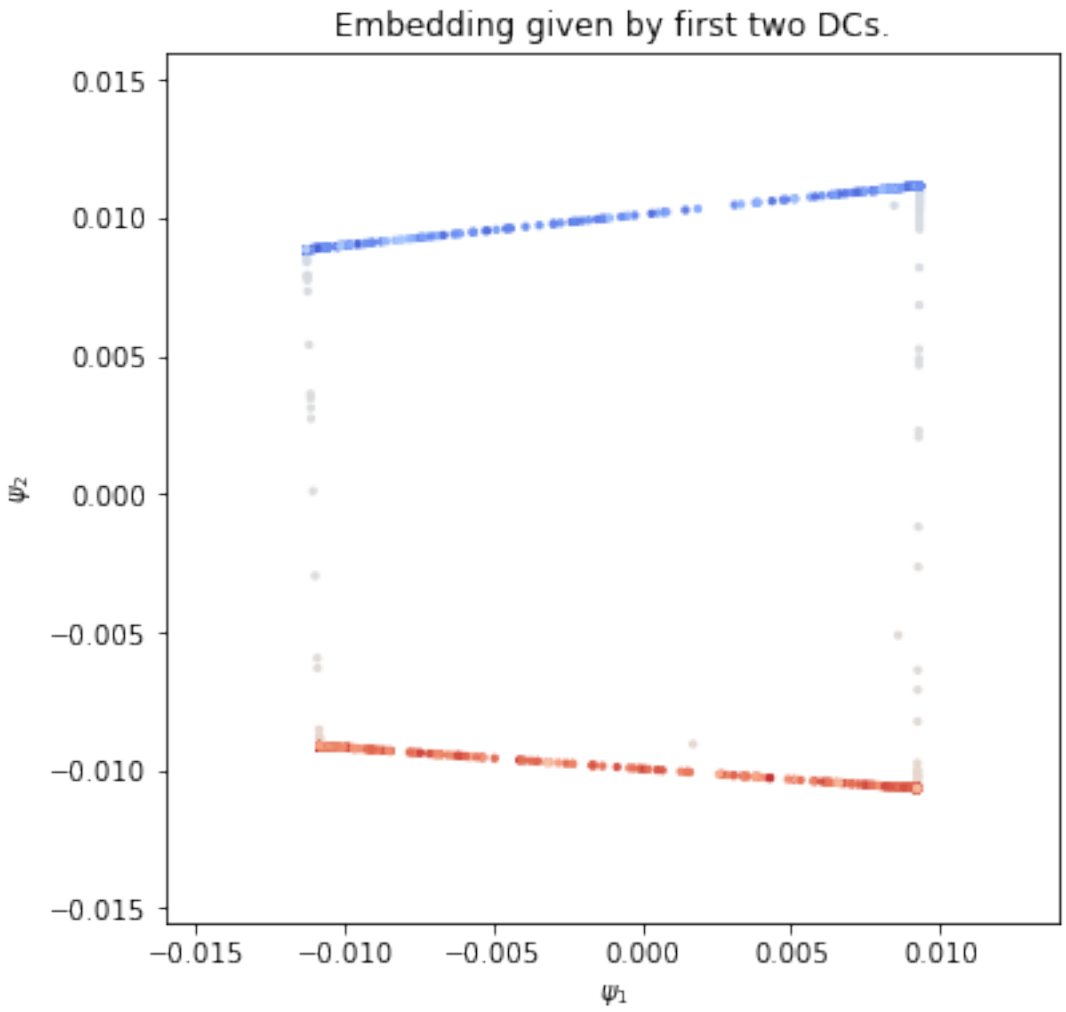
```
embedding_plot(mytdmap, scatter_kwargs = {'c': X[:,0], 's': 5, 'cmap': 'coolwarm'})

plt.show()
```



From the sampling at temperature 1/beta =1, we can compute diffusion map embedding at lower temperature T_low = 1/beta_low using TMDmap with target measure pi(q) = exp(-beta_low V(q)). Here we set beta_low = 10, and use the data obtained from sampling at higher temperature, i.e. pi(q) = exp(-beta V(q)) with beta = 1.

```
V=DW
beta=10
target_distribution2=np.zeros(len(X))
for i in range(len(X)):
    target_distribution2[i]=np.exp(-beta*V(X[i]))
mytdmap2 = dm.DiffusionMap( alpha = 1.0, n_evecs = 2, epsilon = .2,  k=400)
tmdmap2 = mytdmap2.fit_transform(X, weights=target_distribution2)
```

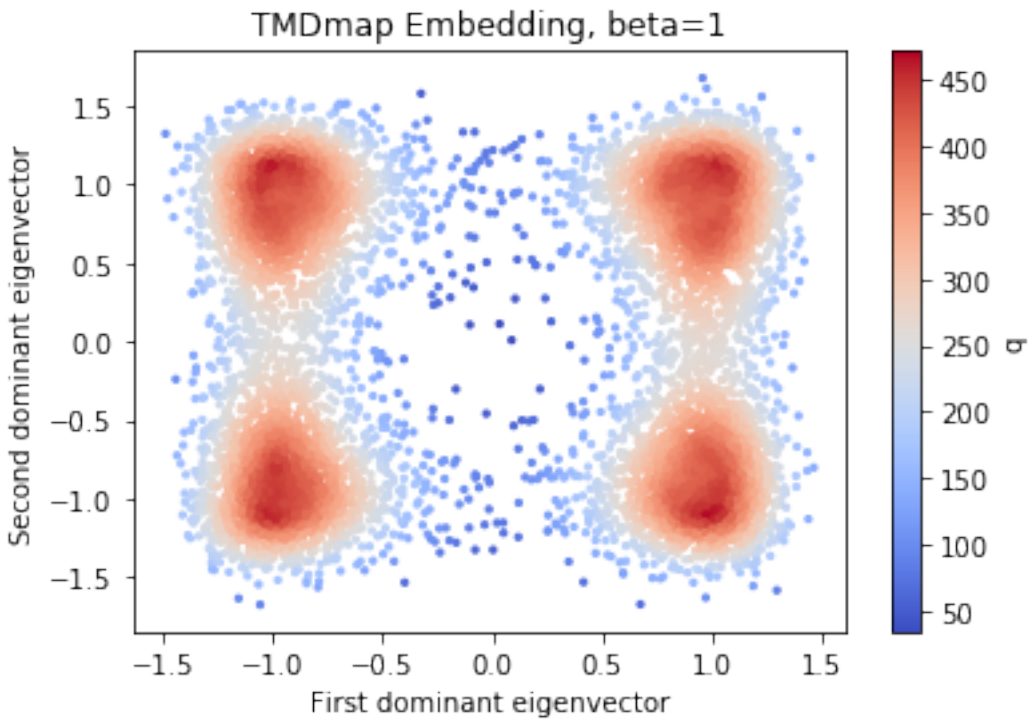
```
embedding_plot(mytdmap2, scatter_kwargs = {'c': X[:,0], 's': 5, 'cmap': 'coolwarm'})

plt.show()
```



Kernel density estimate

We can compute kernel density estimate using kde used in the diffusion map computation.

```
plt.scatter(X[:,0], X[:,1], c = mytdmap.q, s=5, cmap=cm.coolwarm)

clb=plt.colorbar()
clb.set_label('q')
plt.xlabel('First dominant eigenvector')
plt.ylabel('Second dominant eigenvector')
plt.title('TMDmap Embedding, beta=1')

plt.show()
```



Now we check how well we can approximate the target distribution by the formula in the paper (left dominant eigenvector times KDE).

```
import scipy.sparse.linalg as spsl
P = mytdmap.P
[evals, evecs] = spsl.eigs(P.transpose(),k=1, which='LM')

phi = np.real(evecs.ravel())
```

```
q_est = phi*mytdmap.q
q_est = q_est/sum(q_est)
q_exact = target_distribution/sum(target_distribution)
print(np.linalg.norm(q_est - q_exact,1))
```

```
0.0238580958123
```

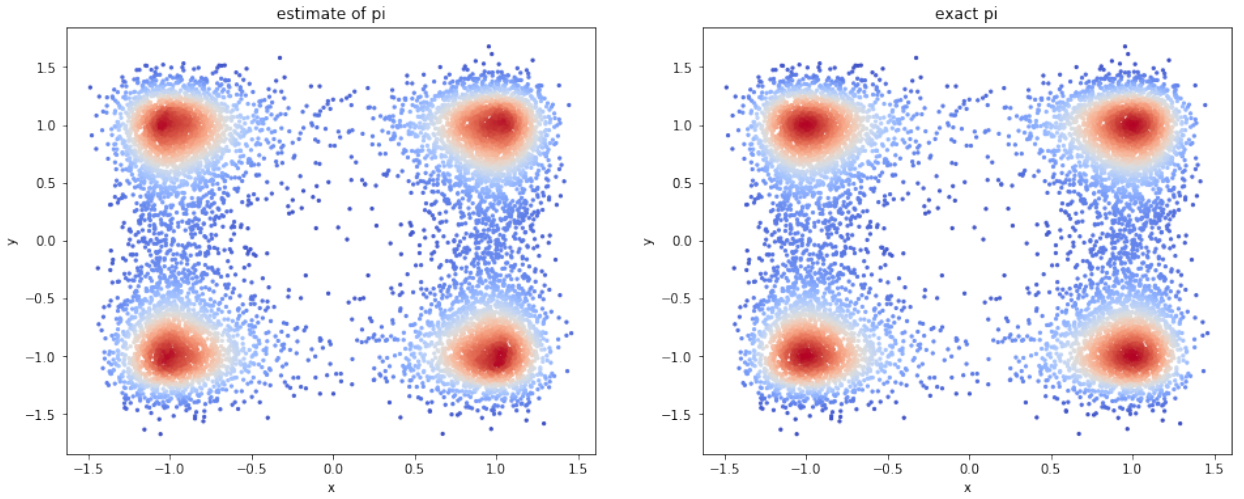
visualize both. there is no visible difference.

```
plt.figure(figsize=(16,6))

ax = plt.subplot(121)
ax.scatter(X[:,0], X[:,1], c = q_est, s=5, cmap=cm.coolwarm)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('estimate of pi')

ax2 = plt.subplot(122)
ax2.scatter(X[:,0], X[:,1], c = q_exact, s=5, cmap=cm.coolwarm)

ax2.set_xlabel('x')
```

(continues on next page)

(continued from previous page)

```
ax2.set_ylabel('y')
ax2.set_title('exact pi')

plt.show()
```



1.5.4 Diffusion maps with general metric

In this notebook, we illustrate how to use an optional metric in the diffusion maps embedding.

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

from mpl_toolkits.mplot3d import Axes3D
from pydiffmap import diffusion_map as dm

%matplotlib inline
```

2D Four-well potential

Load sampled data: discretized Langevin dynamics at temperature T=1, friction 1, and time step size dt=0.01, with double-well potentials in x and y, with higher barrier in y.

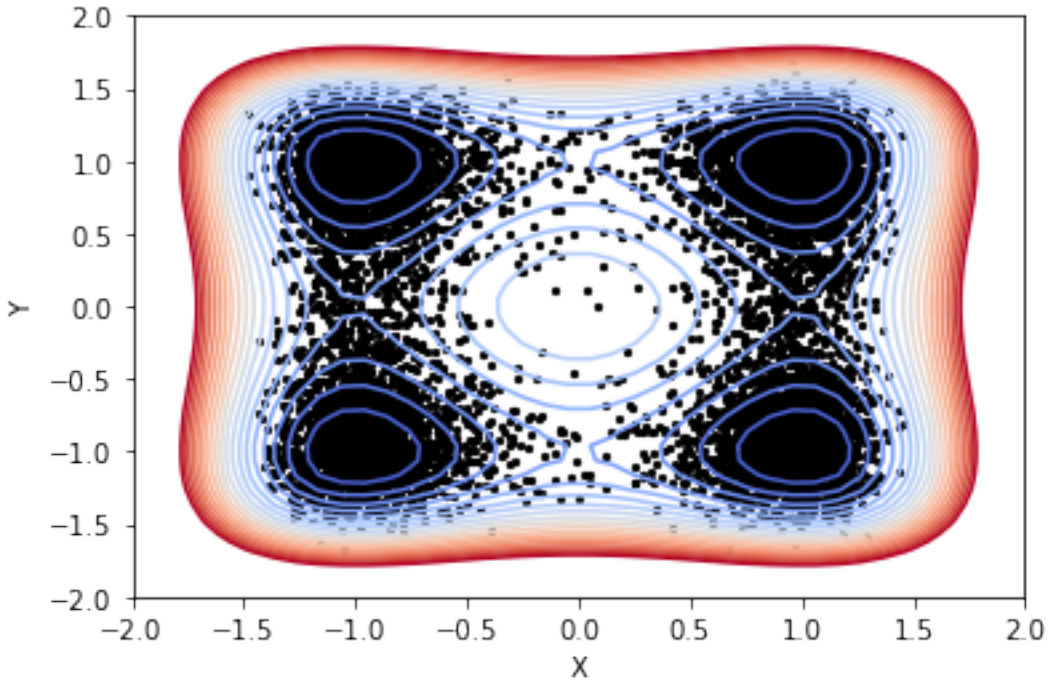
```
X=np.load('Data/4wells_traj.npy')
```

```
def DW1(x):
        return 2.0*(np.linalg.norm(x)**2-1.0)**2

def DW2(x):
        return 4.0*(np.linalg.norm(x)**2-1.0)**2

def DW(x):
    return DW1(x[0]) + DW1(x[1])

from matplotlib import cm
```

(continues on next page)

(continued from previous page)

```
mx=5

xe=np.linspace(-mx, mx, 100)
ye=np.linspace(-mx, mx, 100)
energyContours=np.zeros((100, 100))
for i in range(0,len(xe)):
            for j in range(0,len(ye)):
                xtmp=np.array([xe[i], ye[j]] )
                energyContours[j,i]=DW(xtmp)

levels = np.arange(0, 10, 0.5)
plt.contour(xe, ye, energyContours, levels, cmap=cm.coolwarm)
plt.scatter(X[:,0], X[:,1], s=5, c='k')
plt.xlabel('X')
plt.ylabel('Y')
plt.xlim([-2,2])
plt.ylim([-2,2])
plt.show()
```

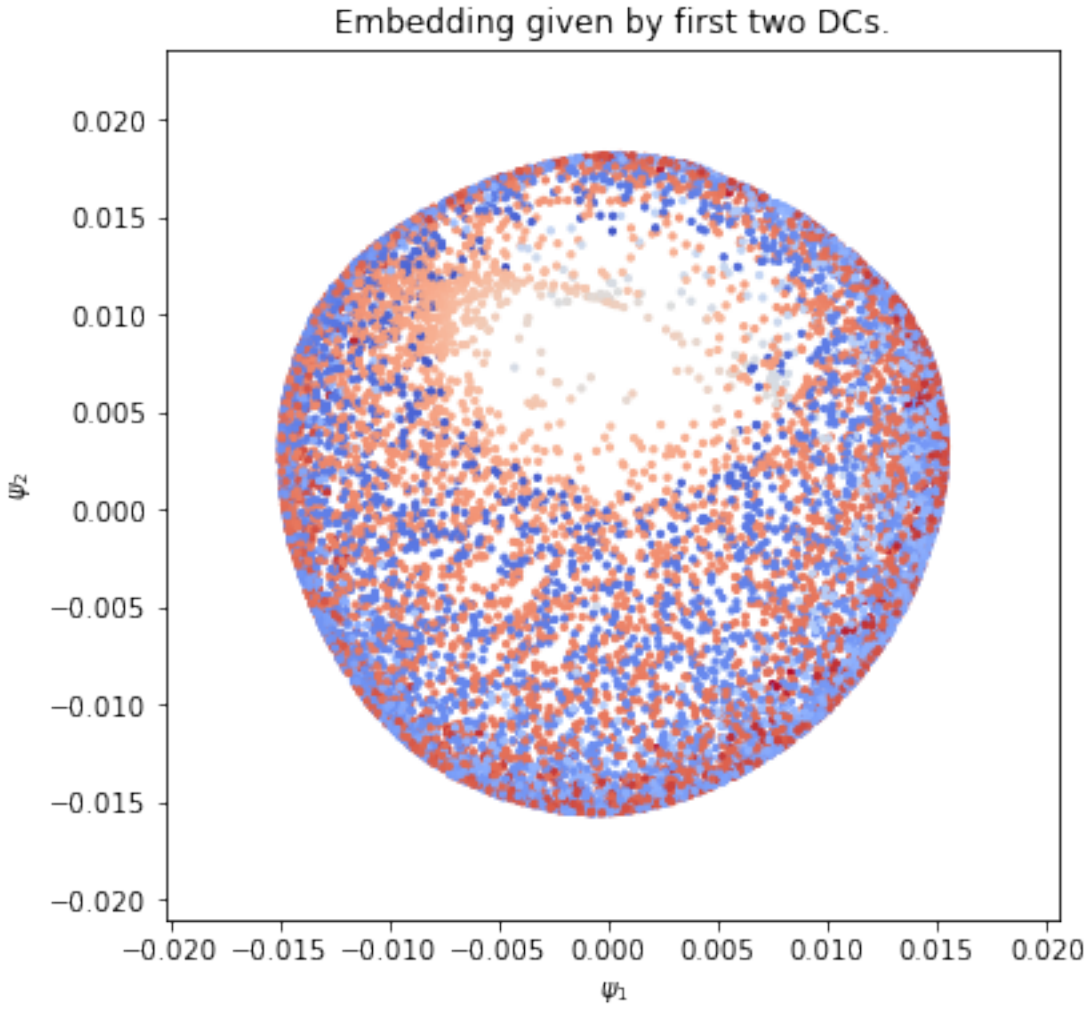



```
def periodicMetric(v1, v2):

    upperLimitPBC=1
    lowerLimitPBC=-1

    BoxLength = upperLimitPBC - lowerLimitPBC

    v = v1 - v2

    v = v - BoxLength*np.floor(v / BoxLength)


    return np.linalg.norm(v)
```

Compute diffusion map embedding

```
mydmap = dm.DiffusionMap(n_evecs = 2, epsilon = .2, alpha = 0.5, k=200,␣
↪metric=periodicMetric)

dmap = mydmap.fit_transform(X)
```

```
from pydiffmap.visualization import embedding_plot, data_plot

embedding_plot(mydmap, scatter_kwargs = {'s': 5, 'c': X[:,0], 'cmap': 'coolwarm'})
plt.show()
```



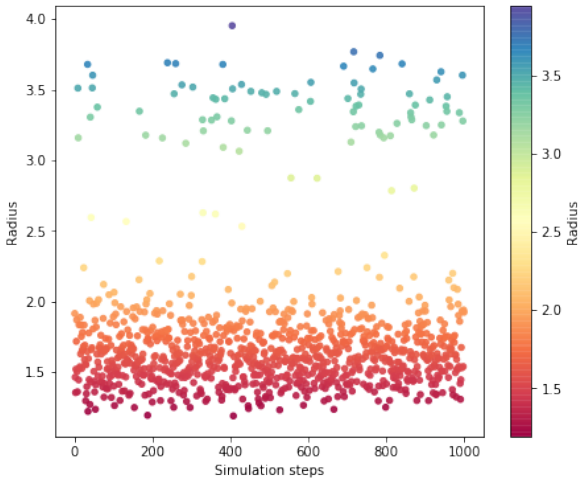
Dimer trajectory

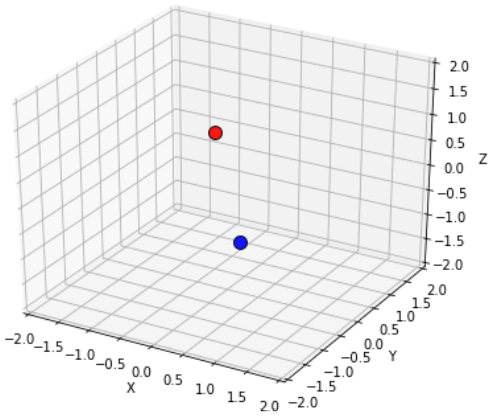
We import trajectory of two particles connected by a double-well potential, which is a function of a radius: V(r) = V_DW(r). The dimer was simulated at 300K with Langevin dynamics using OpenMM. The obvious collective variable is the radius case and we demonstrate how the first dominant eigenvector obtained from the diffusion map clearly correlates with this reaction coordinate. As a metric, we use the root mean square deviation (RMSD) from the package https://pypi.python.org/pypi/rmsd/1.2.5.

```
traj=np.load('Data/dimer_trajectory.npy')
energy=np.load('Data/dimer_energy.npy')
print('Loaded trajectory of '+repr(len(traj))+' steps of dimer molecule: '+repr(traj.
↪shape[1])+' particles in dimension '+repr(traj.shape[2])+'.')
```

```
Loaded trajectory of 1000 steps of dimer molecule: 2 particles in dimension 3.
```

```
def compute_radius(X):
    return np.linalg.norm(X[:,0,:]-X[:,1,:], 2, axis=1)

fig = plt.figure(figsize=[16,6])
ax = fig.add_subplot(121)

radius= compute_radius(traj)
cax2 = ax.scatter(range(len(radius)), radius, c=radius, s=20,alpha=0.90,cmap=plt.cm.
↪Spectral)
cbar = fig.colorbar(cax2)
cbar.set_label('Radius')
ax.set_xlabel('Simulation steps')
ax.set_ylabel('Radius')


ax2 = fig.add_subplot(122, projection='3d')

L=2

i=0

ax2.scatter(traj[i,0,0], traj[i,0,1], traj[i,0,2], c='b', s=100, alpha=0.90,␣
↪edgecolors='none', depthshade=True,)
ax2.scatter(traj[i,1,0], traj[i,1,1], traj[i,1,2], c='r', s=100, alpha=0.90,␣
↪edgecolors='none',  depthshade=True,)

ax2.set_xlim([-L, L])
ax2.set_ylim([-L, L])
ax2.set_zlim([-L, L])

ax2.set_xlabel('X')
ax2.set_ylabel('Y')
ax2.set_zlabel('Z')

plt.show()
```






```
# download from https://pypi.python.org/pypi/rmsd/1.2.5
import rmsd


def myRMSDmetric(arr1, arr2):
    """
    This function is built under the assumption that the space dimension is 3!!!
    Requirement from sklearn radius_neighbors_graph: The callable should take two␣
↪arrays as input and return one value indicating the distance between them.
     Input: One row from reshaped XYZ trajectory as number of steps times nDOF
     Inside: Reshape to XYZ format and apply rmsd as r=rmsd(X[i], X[j])
     Output: rmsd distance
    """

    nParticles = len(arr1) / 3;
    assert (nParticles == int(nParticles))

    X1 = arr1.reshape(int(nParticles), 3 )
    X2 = arr2.reshape(int(nParticles), 3 )

    X1 = X1 -  rmsd.centroid(X1)
    X2 = X2 -  rmsd.centroid(X2)

    return rmsd.kabsch_rmsd(X1, X2)
```

Compute diffusion map embedding using the rmsd metric from above.

```
epsilon=0.1

Xresh=traj.reshape(traj.shape[0], traj.shape[1]*traj.shape[2])
mydmap = dm.DiffusionMap(n_evecs = 1, epsilon = epsilon, alpha = 0.5, k=1000,␣
↪metric=myRMSDmetric)
dmap = mydmap.fit_transform(Xresh)
```

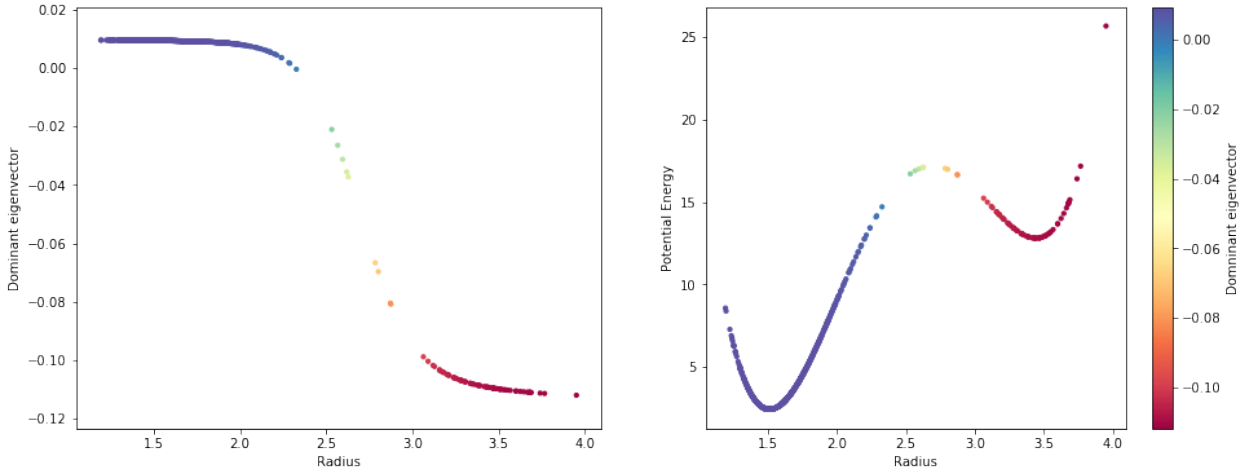
Plot the dominant eigenvector over radius, to show the correlation with this collective variable.

```
evecs = mydmap.evecs

fig = plt.figure(figsize=[16,6])
ax = fig.add_subplot(121)
```

(continues on next page)

(continued from previous page)

```
ax.scatter(compute_radius(traj), evecs[:,0], c=evecs[:,0], s=10, cmap=plt.cm.Spectral)
ax.set_xlabel('Radius')
ax.set_ylabel('Dominant eigenvector')

ax2 = fig.add_subplot(122)
#
cax2 = ax2.scatter(compute_radius(traj), energy, c=evecs[:,0], s=10, cmap=plt.cm.
↪Spectral)
ax2.set_xlabel('Radius')
ax2.set_ylabel('Potential Energy')
cbar = fig.colorbar(cax2)
cbar.set_label('Domninant eigenvector')
plt.show()
```



1.6 Reference

1.6.1 diffusion_map

Routines and Class definitions for the diffusion maps algorithm.

class pydiffmap.diffusion_map.**DiffusionMap**(*alpha=0.5*, *k=64*, *kernel_type='gaussian'*, *epsilon='bgh'*, *n_evecs=1*, *neighbor_params=None*, *metric='euclidean'*, *metric_params=None*)

Diffusion Map object to be used in data analysis for fun and profit.

Parameters

- **alpha** (*scalar, optional*) – Exponent to be used for the left normalization in constructing the diffusion map.
- **k** (*int, optional*) – Number of nearest neighbors over which to construct the kernel.
- **kernel_type** (*string, optional*) – Type of kernel to construct. Currently the only option is 'gaussian', but more will be implemented.
- **epsilon** (*string or scalar, optional*) – Method for choosing the epsilon. Currently, the only

options are to provide a scalar (epsilon is set to the provided scalar) or 'bgh' (Berry, Giannakis and Harlim).

- **n_evecs** (*int, optional*) – Number of diffusion map eigenvectors to return
- **neighbor_params** (*dict or None, optional*) – Optional parameters for the nearest Neighbor search. See scikit-learn NearestNeighbors class for details.
- **metric** (*string, optional*) – Metric for distances in the kernel. Default is 'euclidean'. The callable should take two arrays as input and return one value indicating the distance between them.
- **metric_params** (*dict or None, optional*) – Optional parameters required for the metric given.

Examples

setup neighbor_params list with as many jobs as CPU cores and kd_tree neighbor search. >>> neighbor_params = {'n_jobs': -1, 'algorithm': 'kd_tree'} # initialize diffusion map object with the top two eigenvalues being computed, epsilon set to 0.1 # and alpha set to 1.0. >>> mydmap = DiffusionMap(n_evecs = 2, epsilon = .1, alpha = 1.0, neighbor_params = neighbor_params)

fit(*X*, *weights=None*)

Fits the data.

Parameters

- **X** (*array-like, shape (n_query, n_features)*) – Data upon which to construct the diffusion map.
- **weights** (*array-like, optional, shape(n_query)*) – Values of a weight function for the data. This effectively adds a drift term equivalent to the gradient of the log of weighting function to the final operator.

Returns **self** (*the object itself*)

fit_transform(*X*, *weights=None*)

Fits the data and returns diffusion coordinates. equivalent to calling dmap.fit(X).transform(x).

Parameters

- **X** (*array-like, shape (n_query, n_features)*) – Data upon which to construct the diffusion map.
- **weights** (*array-like, optional, shape (n_query)*) – Values of a weight function for the data. This effectively adds a drift term equivalent to the gradient of the log of weighting function to the final operator.

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

transform(*Y*)

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

Parameters **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.

1.6.2 kernel

A class to implement diffusion kernels.

class pydiffmap.kernel.**Kernel**(*kernel_type='gaussian'*, *epsilon='bgh'*, *k=64*, *neighbor_params=None*, *metric='euclidean'*, *metric_params=None*)

Class abstracting the evaluation of kernel functions on the dataset.

Parameters

- **type** (*string, optional*) – Type of kernel to construct. Currently the only option is 'gaussian', but more will be implemented.
- **epsilon** (*string, optional*) – Method for choosing the epsilon. Currently, the only options are to provide a scalar (epsilon is set to the provided scalar) or 'bgh' (Berry, Giannakis and Harlim).
- **k** (*int, optional*) – Number of nearest neighbors over which to construct the kernel.
- **neighbor_params** (*dict or None, optional*) – Optional parameters for the nearest Neighbor search. See scikit-learn NearestNeighbors class for details.
- **metric** (*string, optional*) – Distance metric to use in constructing the kernel. This can be selected from any of the scipy.spatial.distance metrics, or a callable function returning the distance.
- **metric_params** (*dict or None, optional*) – Optional parameters required for the metric given.

choose_optimal_epsilon(*epsilon=None*)

Chooses the optimal value of epsilon and automatically detects the dimensionality of the data.

Parameters **epsilon** (*string or scalar, optional*) – Method for choosing the epsilon. Currently, the only options are to provide a scalar (epsilon is set to the provided scalar) or 'bgh' (Berry, Giannakis and Harlim).

Returns **self** (*the object itself*)

compute(*Y=None*)

Computes the sparse kernel matrix.

Parameters **Y** (*array-like, shape (n_query, n_features), optional.*) – Data against which to calculate the kernel values. If not provided, calculates against the data provided in the fit.

Returns **K** (*array-like, shape (n_query_X, n_query_Y)*) – Values of the kernel matrix.

fit(*X*)

Fits the kernel to the data X, constructing the nearest neighbor tree.

Parameters **X** (*array-like, shape (n_query, n_features)*) – Data upon which to fit the nearest neighbor tree.

Returns **self** (*the object itself*)

pydiffmap.kernel.**choose_optimal_epsilon_BGH**(*scaled_distsq*, *epsilons=None*)

Calculates the optimal epsilon for kernel density estimation according to the criteria in Berry, Giannakis, and Harlim.

Parameters

- **scaled_distsq** (*numpy array*) – Values for scaled distance squared values, in no particular order or shape. (This is the exponent in the Gaussian Kernel, aka the thing that gets divided by epsilon).

- **epsilons** (*array-like, optional*) – Values of epsilon from which to choose the optimum. If not provided, uses all powers of 2. from 2^-40 to 2^40

Returns

- **epsilon** (*float*) – Estimated value of the optimal length-scale parameter.
- **d** (*int*) – Estimated dimensionality of the system.

Notes

Erik sez : I have a suspicion that the derivation here explicitly assumes that the kernel is Gaussian. However, I'm not sure. Also, we should perhaps replace this with some more intelligent optimization routine. Here, I'm just picking from several values and choosin the best.

References

The algorithm given is based on[1]. If you use this code, please cite them.

1.6.3 visualization

Some convenient visalisation routines.

`pydiffmap.visualization.`**`data_plot`**(*dmap_instance*, *n_evec=1*, *dim=2*, *scatter_kwargs=None*, *show=True*)

Creates diffusion map embedding scatterplot. By default, the first two diffusion coordinates are plotted against each other. This only plots against the first two or three (as controlled by 'dim' parameter) dimensions of the data, however: effectively this assumes the data is two resp. three dimensional.

Parameters

- **dmap_instance** (*DiffusionMap Instance*) – An instance of the DiffusionMap class.
- **n_evec** (*int, optional*) – The eigenfunction that should be used to color the plot.
- **dim** (*int, optional, 2 or 3.*) – Optional argument that controls if a two- or three dimensional plot is produced.
- **scatter_kwargs** (*dict, optional*) – Optional arguments to be passed to the scatter plot, e.g. point color, point size, colormap, etc.
- **show** (*boolean, optional*) – If true, calls plt.show()

Returns **fig** (*pyplot figure object*) – Figure object where everything is plotted on.

`pydiffmap.visualization.`**`embedding_plot`**(*dmap_instance*, *dim=2*, *scatter_kwargs=None*, *show=True*)

Creates diffusion map embedding scatterplot. By default, the first two diffusion coordinates are plotted against each other.

Parameters

- **dmap_instance** (*DiffusionMap Instance*) – An instance of the DiffusionMap class.
- **dim** (*int, optional, 2 or 3.*) – Optional argument that controls if a two- or three dimensional plot is produced.

[1] T. Berry, D. Giannakis, and J. Harlim, Physical Review E 91, 032915 (2015).

- **scatter_kwargs** (*dict, optional*) – Optional arguments to be passed to the scatter plot, e.g. point color, point size, colormap, etc.
- **show** (*boolean, optional*) – If true, calls plt.show()

Returns **fig** (*pyplot figure object*) – Figure object where everything is plotted on.

Examples

Plots the top two diffusion coords, colored by the first coord. >>> scatter_kwargs = {'s': 2, 'c': mydmap.dmap[:,0], 'cmap': 'viridis'} >>> embedding_plot(mydmap, scatter_kwargs)

1.7 Contributing

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

1.7.1 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.

1.7.2 Documentation improvements

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

1.7.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/DiffusionMapsAcademics/pyDiffMap/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 :)

1.7.4 Development

To set up *python-pydiffmap* for local development:

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

```
git clone git@github.com:your_name_here/python-pydiffmap.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.

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`.

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
```

1.8 Authors

- Ralf Banisch
- Erik Henning Thiede
- Zofia Trstanova

[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 ...

1.9 Acknowledgements

This work was partially funded by grant EPSR EP/P006175/1 as well as the Molecular Sciences Software Institute (MolSSI). Computing resources were provided in part by the University of Chicago Research Computing Center (RCC). We also want to thank the following scientists for their input and advice:

- Prof. Dimitris Giannakis for help in implementing the automatic bandwidth selection algorithm.

1.10 Changelog

1.10.1 0.1.0 (2017-12-06)

- Fixed setup.py issues.

1.10.2 0.1.0 (2017-12-06)

- Added base functionality to the code.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index