graphtools Documentation

Release 1.5.2

Scott Gigante and Jay Stanley, Yale University

Aug 10, 2020

Contents

Tools for building and manipulating graphs in Python.

CHAPTER 1

Installation

1.1 Installation with *pip*

Install graphtools using:

```
pip install --user graphtools
```

1.2 Installation from source

Install from source using:

```
git clone git://github.com/KrishnaswamyLab/graphtools.git
cd graphtools
python setup.py install --user
```

CHAPTER 2

Reference

2.1 API

graphtools.api.**Graph**(*data*, *n_pca=None*, *rank_threshold=None*, *knn=5*, *decay=40*, *bandwidth=None*, *bandwidth_scale=1.0*, *knn_max=None*, *anisotropy=0*, *distance='euclidean'*, *thresh=0.0001*, *kernel_symm='+'*, *theta=None*, *precomputed=None*, *beta=1*, *sample_idx=None*, *adaptive_k=None*, *n_landmark=None*, *n_svd=100*, *n_jobs=-1*, *verbose=False*, *random_state=None*, *graphtype='auto'*, *use_pygsp=False*, *initialize=True*, ***kwargs*)

Create a graph built on data.

Automatically selects the appropriate DataGraph subclass based on chosen parameters. Selection criteria: - if *graphtype* is given, this will be respected - otherwise: – if *sample_idx* is given, an MNNGraph will be created – if *precomputed* is not given, and either *decay* is *None* or *thresh* is given, a kNNGraph will be created - otherwise, a TraditionalGraph will be created.

Incompatibilities: - MNNGraph and kNNGraph cannot be precomputed - kNNGraph and TraditionalGraph do not accept sample indices

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix*. TODO: accept pandas dataframes'
- **n_pca** ({*int*, *None*, *bool*, 'auto'}, optional (default: *None*)) – number of PC dimensions to retain for graph building. If n_pca in *[None, False, 0]*, uses the original data. If 'auto' or *True* then estimate using a singular value threshold Note: if data is sparse, uses SVD instead of PCA TODO: should we subtract and store the mean?
- **rank_threshold** (*float*, 'auto', optional (default: 'auto')) – threshold to use when estimating rank for *n_pca in [True, 'auto']*. If 'auto', this threshold is s_max * eps * max(n_samples, n_features) where s_max is the maximum singular value of the data matrix and eps is numerical precision. [press2007].

- **knn** (*int*, optional (default: 5)) – Number of nearest neighbors (including self) to use to build the graph
- **decay** (*int* or *None*, optional (default: 40)) – Rate of alpha decay to use. If *None*, alpha decay is not used and a vanilla k-Nearest Neighbors graph is returned.
- **bandwidth** (*float*, list-like,'callable', or *None*, optional (default: *None*)) – Fixed bandwidth to use. If given, overrides *knn*. Can be a single bandwidth, list-like (shape=[n_samples]) of bandwidths for each sample, or a *callable* that takes in an *n x n* distance matrix and returns a a single value or list-like of length n (shape=[n_samples])
- **bandwidth_scale** (*float*, optional (default : 1.0)) – Rescaling factor for bandwidth.
- **knn_max** (*int* or *None*, optional (default : *None*)) – Maximum number of neighbors with nonzero affinity
- **anisotropy** (*float, optional (default: 0)*) – Level of anisotropy between 0 and 1 (alpha in Coifman & Lafon, 2006)
- **distance** (*str*, optional (default: *'euclidean'*)) – Any metric from *scipy.spatial.distance* can be used distance metric for building kNN graph. TODO: actually sklearn.neighbors has even more choices
- **thresh** (*float*, optional (default: *1e-4*)) – Threshold above which to calculate alpha decay kernel. All affinities below *thresh* will be set to zero in order to save on time and memory constraints.
- **kernel_symm** (*string, optional (default: '+')*) – Defines method of kernel symmetrization. '+' : additive '*' : multiplicative 'mnn' : min-max MNN symmetrization 'none' : no symmetrization
- **theta** (*float (default: None)*) – Min-max symmetrization constant or matrix. Only used if kernel_symm='mnn'. K = *theta * min(K, K.T) + (1 - theta) * max(K, K.T)*
- **precomputed** ({'distance', 'affinity', 'adjacency', *None*}, optional (default: *None*)) – If the graph is precomputed, this variable denotes which graph matrix is provided as *data*. Only one of *precomputed* and *n_pca* can be set.
- **beta** (*float, optional(default: 1)*) – Multiply between - batch connections by beta
- **sample_idx** (*array-like*) – Batch index for MNN kernel
- **adaptive_k** (*{'min', 'mean', 'sqrt', 'none'}* (default: None)) – Weights MNN kernel adaptively using the number of cells in each sample according to the selected method.
- **n_landmark** (*int*, optional (default: 2000)) – number of landmarks to use
- **n_svd** (*int*, optional (default: 100)) – number of SVD components to use for spectral clustering
- **random_state** (*int* or *None*, optional (default: *None*)) – Random state for random PCA
- **verbose** (*bool*, optional (default: *True*)) – Verbosity. TODO: should this be an integer instead to allow multiple levels of verbosity?
- **n_jobs** (*int*, optional (default : 1)) – The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used
- **graphtype** (*{'exact', 'knn', 'mnn', 'auto'} (Default: 'auto')*) – Manually selects graph type. Only recommended for expert users

- **use_pygsp** (*bool* (Default: *False*)) – If true, inherits from *pygsp.graphs.Graph*.
- **initialize** (*bool* (Default: *True*)) – If True, initialize the kernel matrix on instantiation
- ****kwargs** (extra arguments for *pygsp.graphs.Graph*) –

Returns G

Return type *DataGraph*

Raises ValueError : if selected parameters are incompatible.

References

graphtools.api.**from_igraph**(*G*, *attribute='weight'*, ***kwargs*)

Convert an igraph.Graph to a graphtools.Graph

Creates a graphtools.graphs.TraditionalGraph with a precomputed adjacency matrix

Parameters

- **G** (*igraph.Graph*) – Graph to be converted
- **attribute** (*str, optional (default: "weight")*) – attribute containing edge weights, if any. If None, unweighted graph is built
- **kwargs** – keyword arguments for graphtools.Graph

Returns G

Return type *graphtools.graphs.TraditionalGraph*

graphtools.api.**read_pickle**(*path*)

Load pickled Graphtools object (or any object) from file.

Parameters **path** (*str*) – File path where the pickled object will be loaded.

2.2 Graph Classes

class graphtools.graphs.**LandmarkGraph**(*data*, *n_landmark=2000*, *n_svd=100*, ***kwargs*)

Bases: *graphtools.base.DataGraph*

Landmark graph

Adds landmarking feature to any data graph by taking spectral clusters and building a 'landmark operator' from clusters to samples and back to clusters. Any transformation on the landmark kernel is trivially extended to the data space by multiplying by the transition matrix.

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix.*, *pandas.DataFrame*, *pandas.SparseDataFrame*.
- **n_landmark** (*int*, optional (default: 2000)) – number of landmarks to use
- **n_svd** (*int*, optional (default: 100)) – number of SVD components to use for spectral clustering

landmark_op

Landmark operator. Can be treated as a diffusion operator between landmarks.

Type array-like, shape=[n_landmark, n_landmark]

transitions
Transition probabilities between samples and landmarks.

Type array-like, shape=[n_samples, n_landmark]

clusters
Private attribute. Cluster assignments for each sample.

Type array-like, shape=[n_samples]

Examples

```
>>> G = graphtools.Graph(data, n_landmark=1000)
>>> X_landmark = transform(G.landmark_op)
>>> X_full = G.interpolate(X_landmark)
```

K
Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P
Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()
Build the kernel matrix

Abstract method that all child classes must implement. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*)
Build a kernel from new input data *Y* to the *self.data*

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **K_yx** – kernel matrix where each row represents affinities of a single sample in *Y* to all samples in *self.data*.

Return type array-like, [n_samples_y, n_samples]

Raises

- ValueError: if this Graph is not capable of extension or
- if the supplied data is the wrong shape

build_landmark_op()
Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

clusters
Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*data*, ***kwargs*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel

Synonym for K

kernel_degree

Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

landmark_op

Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

set_params(***params*)

Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_landmark - n_svd

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)

Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]

- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions
Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

weighted

class graphtools.graphs.**MNNGraph**(*data*, *sample_idx*, *knn=5*, *beta=1*, *n_pca=None*, *decay=None*, *adaptive_k=None*, *bandwidth=None*, *distance='euclidean'*, *thresh=0.0001*, *n_jobs=1*, ***kwargs*)

Bases: *graphtools.base.DataGraph*

Mutual nearest neighbors graph

Performs batch correction by forcing connections between batches, but only when the connection is mutual (i.e. x is a neighbor of y _and_ y is a neighbor of x).

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix*, *pandas.DataFrame*, *pandas.SparseDataFrame*.
- **sample_idx** (*array-like, shape=[n_samples]*) – Batch index
- **beta** (*float*, optional (default: 1)) – Downweight between-batch affinities by beta
- **adaptive_k** ({'min', 'mean', 'sqrt', *None*} (default: None)) – Weights MNN kernel adaptively using the number of cells in each sample according to the selected method.

subgraphs

Graphs representing each batch separately

Type list of *graphtools.graphs.kNNGraph*

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()

Build the MNN kernel.

Build a mutual nearest neighbors kernel.

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *theta=None*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **theta** (array-like or *None*, optional (default: *None*)) – if *self.theta* is a matrix, theta values must be explicitly specified between *Y* and each sample in *self.data*

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*Y*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel

Synonym for K

kernel_degree

Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

set_params(***params*)

Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - adaptive_k - decay - distance - thresh - beta

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)

Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

weighted

class graphtools.graphs.**MNNLandmarkGraph**(*data*, *sample_idx*, *knn=5*, *beta=1*, *n_pca=None*, *decay=None*, *adaptive_k=None*, *bandwidth=None*, *distance='euclidean'*, *thresh=0.0001*, *n_jobs=1*, ***kwargs*)

Bases: *graphtools.graphs.MNNGraph*, *graphtools.graphs.LandmarkGraph*

K
Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()

Build the MNN kernel.

Build a mutual nearest neighbors kernel.

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *theta=None*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **theta** (array-like or *None*, optional (default: *None*)) – if *self.theta* is a matrix, theta values must be explicitly specified between *Y* and each sample in *self.data*

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

build_landmark_op()

Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

clusters

Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

diff_aff

Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*data*, ***kwargs*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

landmark_op
Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - adaptive_k - decay - distance - thresh - beta

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight', **kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions
Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

weighted

class graphtools.graphs.**MNNLandmarkPyGSPGraph**(*data*, *sample_idx*, *knn=5*, *beta=1*, *n_pca=None*, *decay=None*, *adaptive_k=None*, *bandwidth=None*, *distance='euclidean'*, *thresh=0.0001*, *n_jobs=1*, ***kwargs*)

Bases: *graphtools.graphs.MNNGraph*, *graphtools.graphs.LandmarkGraph*, *graphtools.base.PyGSPGraph*

A
Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D

Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

apply_anisotropy(*K*)

build_kernel()
Build the MNN kernel.

Build a mutual nearest neighbors kernel.

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *theta=None*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **theta** (array-like or *None*, optional (default: *None*)) – if *self.theta* is a matrix, theta values must be explicitly specified between *Y* and each sample in *self.data*

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

build_landmark_op()
Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

check_weights()
Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a "not a number" value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false
- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

clusters
Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

compute_differential_operator()
Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)
Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U \Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)
Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d

The degree (the number of neighbors) of each node.

diff_aff

Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op

Synonym for P

div(s)

Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw

The weighted degree (the sum of weighted edges) of each node.

e

Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)

Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extend_to_data(*data*, ***kwargs*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

extract_components()
Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()
Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

get_params()
Get parameters from this object

gft(*s*)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)
Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)

Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator `D`.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)

Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis `U`.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

is_connected(*recompute=False*)

Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)

Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

landmark_op
Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

lmax
Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate (*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot(***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - adaptive_k - decay - distance - thresh - beta

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

subgraph(*ind*)
Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions

Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

translate(*f*, *i*)

Translate the signal *f* to the node *i*.

Parameters

- **f** (*ndarray*) – Signal
- **i** (*int*) – Indices of vertex

Returns **ft**

Return type translate signal

weighted

class graphtools.graphs.**MNNPyGSPGraph**(*data*, *sample_idx*, *knn=5*, *beta=1*, *n_pca=None*, *decay=None*, *adaptive_k=None*, *bandwidth=None*, *distance='euclidean'*, *thresh=0.0001*, *n_jobs=1*, ***kwargs*)

Bases: *graphtools.graphs.MNNGraph*, *graphtools.base.PyGSPGraph*

A

Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D

Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

apply_anisotropy(*K*)

build_kernel()
Build the MNN kernel.

Build a mutual nearest neighbors kernel.

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *theta=None*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **theta** (array-like or *None*, optional (default: *None*)) – if *self.theta* is a matrix, theta values must be explicitly specified between *Y* and each sample in *self.data*

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

check_weights()
Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a "not a number" value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false
- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

compute_differential_operator()
Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)

Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U \Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
```

(continues on next page)

(continued from previous page)

```
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)

Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d

The degree (the number of neighbors) of each node.

diff_aff

Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

div(*s*)
Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw
The weighted degree (the sum of weighted edges) of each node.

e
Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)
Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extend_to_data(*Y*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

extract_components()

Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()

Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

get_params()
Get parameters from this object

gft(*s*)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)
Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)
Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)
Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis *U*.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

is_connected(*recompute=False*)

Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)

Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

kernel

Synonym for K

kernel_degree

Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

lmax

Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate(*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot(***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - adaptive_k - decay - distance - thresh - beta

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

subgraph(*ind*)
Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns G

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

translate(*f*, *i*)
Translate the signal *f* to the node *i*.

Parameters

- **f** (*ndarray*) – Signal
- **i** (*int*) – Indices of vertex

Returns ft

Return type translate signal

weighted

class graphtools.graphs.**TraditionalGraph**(*data*, *knn=5*, *decay=40*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *n_pca=None*, *thresh=0.0001*, *precomputed=None*, ***kwargs*)
Bases: *graphtools.base.DataGraph*

Traditional weighted adjacency graph

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix*, *pandas.DataFrame*, *pandas.SparseDataFrame*. If *precomputed* is not *None*, data should be an [n_samples, n_samples] matrix denoting pairwise distances, affinities, or edge weights.
- **knn** (*int*, optional (default: 5)) – Number of nearest neighbors (including self) to use to build the graph
- **decay** (*int* or *None*, optional (default: 40)) – Rate of alpha decay to use. If *None*, alpha decay is not used.
- **bandwidth** (*float*, list-like,'callable', or *None*, optional (default: *None*)) – Fixed bandwidth to use. If given, overrides *knn*. Can be a single bandwidth, list-like

(shape=[n_samples]) of bandwidths for each sample, or a *callable* that takes in a *n x m* matrix and returns a a single value or list-like of length n (shape=[n_samples])

- **bandwidth_scale** (*float*, optional (default : 1.0)) – Rescaling factor for bandwidth.
- **distance** (*str*, optional (default: *'euclidean'*)) – Any metric from *scipy.spatial.distance* can be used distance metric for building kNN graph. TODO: actually sklearn.neighbors has even more choices
- **n_pca** ({*int*, *None*, *bool*, 'auto'}, optional (default: *None*)) – number of PC dimensions to retain for graph building. If n_pca in *[None,False,0]*, uses the original data. If *True* then estimate using a singular value threshold Note: if data is sparse, uses SVD instead of PCA TODO: should we subtract and store the mean?
- **rank_threshold** (*float*, 'auto', optional (default: 'auto')) – threshold to use when estimating rank for *n_pca in [True, 'auto']*. Note that the default kwarg is *None* for this parameter. It is subsequently parsed to 'auto' if necessary. If 'auto', this threshold is smax * np.finfo(data.dtype).eps * max(data.shape) where smax is the maximum singular value of the data matrix. For reference, see, e.g. W. Press, S. Teukolsky, W. Vetterling and B. Flannery, "Numerical Recipes (3rd edition)", Cambridge University Press, 2007, page 795.
- **thresh** (*float*, optional (default: *1e-4*)) – Threshold above which to calculate alpha decay kernel. All affinities below *thresh* will be set to zero in order to save on time and memory constraints.
- **precomputed** ({'distance', 'affinity', 'adjacency', *None*},) – optional (default: *None*) If the graph is precomputed, this variable denotes which graph matrix is provided as *data*. Only one of *precomputed* and *n_pca* can be set.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()

Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. If *precomputed* is not *None*, the appropriate steps in the kernel building process are skipped. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

Raises ValueError: if *precomputed* is not an acceptable value

build_kernel_to_data(*Y*, *knn=None*, *bandwidth=None*, *bandwidth_scale=None*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

Raises

- ValueError: if *precomputed* is not *None*, then the graph cannot
- be extended.

diff_aff

Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op

Synonym for P

extend_to_data(*Y*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

get_params()

Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*

- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel

Synonym for K

kernel_degree

Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

set_params(***params*)

Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Invalid parameters: (these would require modifying the kernel matrix) - precomputed - distance - knn - decay - bandwidth - bandwidth_scale

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)

Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]

- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight', **kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

weighted

class graphtools.graphs.**TraditionalLandmarkGraph**(*data*, *knn=5*, *decay=40*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *n_pca=None*, *thresh=0.0001*, *precomputed=None*, ***kwargs*)

Bases: *graphtools.graphs.TraditionalGraph*, *graphtools.graphs.LandmarkGraph*

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()

Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. If *precomputed* is not *None*, the appropriate steps in the kernel building process are skipped. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

Raises ValueError: if *precomputed* is not an acceptable value

build_kernel_to_data(*Y*, *knn=None*, *bandwidth=None*, *bandwidth_scale=None*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

Raises

- ValueError: if *precomputed* is not *None*, then the graph cannot
- be extended.

build_landmark_op()

Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

clusters
Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*data*, ***kwargs*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

landmark_op
Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Invalid parameters: (these would require modifying the kernel matrix) - precomputed - distance - knn - decay - bandwidth - bandwidth_scale

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]

- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions
Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

weighted

class graphtools.graphs.**TraditionalLandmarkPyGSPGraph**(*data*, *knn=5*, *decay=40*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *n_pca=None*, *thresh=0.0001*, *precomputed=None*, ***kwargs*)

Bases: *graphtools.graphs.TraditionalGraph*, *graphtools.graphs.LandmarkGraph*, *graphtools.base.PyGSPGraph*

A

Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D

Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

apply_anisotropy(*K*)

build_kernel()

Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. If *precomputed* is not *None*, the appropriate steps in the kernel building process are skipped. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

Raises ValueError: if *precomputed* is not an acceptable value

build_kernel_to_data(*Y*, *knn=None*, *bandwidth=None*, *bandwidth_scale=None*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

Raises

- ValueError: if *precomputed* is not *None*, then the graph cannot
- be extended.

build_landmark_op()
Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

check_weights()
Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a "not a number" value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false
- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

clusters
Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

compute_differential_operator()
Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)

Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U \Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)
Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d
The degree (the number of neighbors) of each node.

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

div(*s*)
Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw
The weighted degree (the sum of weighted edges) of each node.

e
Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)
Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extend_to_data(*data*, ***kwargs*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

extract_components()

Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()

Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

get_params()
Get parameters from this object

gft(*s*)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)

Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)

Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)

Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis *U*.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

is_connected(*recompute=False*)

Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random

vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)

Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

kernel

Synonym for K

kernel_degree

Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

landmark_op

Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

lmax
Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate(*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot(***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Invalid parameters: (these would require modifying the kernel matrix) - precomputed - distance - knn - decay - bandwidth - bandwidth_scale

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)

Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

subgraph(*ind*)

Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)

Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions
Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

translate(*f*, *i*)
Translate the signal *f* to the node *i*.

Parameters

- **f** (*ndarray*) – Signal
- **i** (*int*) – Indices of vertex

Returns **ft**

Return type translate signal

weighted

class graphtools.graphs.**TraditionalPyGSPGraph**(*data*, *knn=5*, *decay=40*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *n_pca=None*, *thresh=0.0001*, *precomputed=None*, ***kwargs*)
Bases: *graphtools.graphs.TraditionalGraph*, *graphtools.base.PyGSPGraph*

A
Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D

Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

apply_anisotropy(*K*)

build_kernel()

Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. If *precomputed* is not *None*, the appropriate steps in the kernel building process are skipped. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

Raises ValueError: if *precomputed* is not an acceptable value

build_kernel_to_data(*Y*, *knn=None*, *bandwidth=None*, *bandwidth_scale=None*)

Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

Raises

- ValueError: if *precomputed* is not *None*, then the graph cannot
- be extended.

check_weights()

Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a "not a number" value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false
- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

compute_differential_operator()

Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)

Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U\Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)

Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2}WD^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d

The degree (the number of neighbors) of each node.

diff_aff

Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op

Synonym for P

div(s)

Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw
The weighted degree (the sum of weighted edges) of each node.

e
Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)
Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extend_to_data(*Y*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

extract_components()
Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()
Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

get_params()
Get parameters from this object

gft(*s*)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)
Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)

Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator `D`.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)

Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis `U`.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

is_connected(*recompute=False*)

Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)

Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

lmax
Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate (*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot (***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Invalid parameters: (these would require modifying the kernel matrix) - precomputed - distance - knn - decay - bandwidth - bandwidth_scale

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

subgraph(*ind*)
Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

translate(*f*, *i*)

Translate the signal *f* to the node *i*.

Parameters

- **f** (*ndarray*) – Signal
- **i** (*int*) – Indices of vertex

Returns **ft**

Return type translate signal

weighted

class graphtools.graphs.**kNNGraph**(*data*, *knn=5*, *decay=None*, *knn_max=None*, *search_multiplier=6*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *thresh=0.0001*, *n_pca=None*, ***kwargs*)

Bases: *graphtools.base.DataGraph*

K nearest neighbors graph

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix*, *pandas.DataFrame*, *pandas.SparseDataFrame*.
- **knn** (*int*, optional (default: 5)) – Number of nearest neighbors (including self) to use to build the graph
- **decay** (*int* or *None*, optional (default: *None*)) – Rate of alpha decay to use. If *None*, alpha decay is not used.
- **bandwidth** (*float*, list-like,'callable', or *None*,) – optional (default: *None*) Fixed bandwidth to use. If given, overrides *knn*. Can be a single bandwidth, or a list-like (shape=[n_samples]) of bandwidths for each sample
- **bandwidth_scale** (*float*, optional (default : 1.0)) – Rescaling factor for bandwidth.
- **distance** (*str*, optional (default: *'euclidean'*)) – Any metric from *scipy.spatial.distance* can be used distance metric for building kNN graph. Custom distance functions of form *f(x, y) = d* are also accepted. TODO: actually sklearn.neighbors has even more choices
- **thresh** (*float*, optional (default: *1e-4*)) – Threshold above which to calculate alpha decay kernel. All affinities below *thresh* will be set to zero in order to save on time and memory constraints.

knn_tree

The fitted KNN tree. (cached) TODO: can we be more clever than sklearn when it comes to choosing between KD tree, ball tree and brute force?

Type *sklearn.neighbors.NearestNeighbors*

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()
Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *knn=None*, *knn_max=None*, *bandwidth=None*, *bandwidth_scale=None*)
Build a kernel from new input data *Y* to the *self.data*

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **knn** (*int* or *None*, optional (default: *None*)) – If *None*, defaults to *self.knn*
- **bandwidth** (*float*, *callable*, or *None*, optional (default: *None*)) – If *None*, defaults to *self.bandwidth*
- **bandwidth_scale** (*float*, optional (default : *None*)) – Rescaling factor for bandwidth. If *None*, defaults to self.bandwidth_scale

Returns **K_yx** – kernel matrix where each row represents affinities of a single sample in *Y* to all samples in *self.data*.

Return type array-like, [n_samples_y, n_samples]

Raises ValueError: if the supplied data is the wrong shape

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*Y*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

knn_tree
KNN tree object (cached)

Builds or returns the fitted KNN tree. TODO: can we be more clever than sklearn when it comes to choosing between KD tree, ball tree and brute force?

Returns **knn_tree**

Return type *sklearn.neighbors.NearestNeighbors*

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - knn_max - decay - bandwidth - bandwidth_scale - distance - thresh

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns G

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

weighted

class graphtools.graphs.**kNNLandmarkGraph**(*data*, *knn=5*, *decay=None*, *knn_max=None*, *search_multiplier=6*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *thresh=0.0001*, *n_pca=None*, ***kwargs*)
Bases: *graphtools.graphs.kNNGraph*, *graphtools.graphs.LandmarkGraph*

K
Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P
Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()
Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *knn=None*, *knn_max=None*, *bandwidth=None*, *bandwidth_scale=None*)
Build a kernel from new input data *Y* to the *self.data*

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **knn** (*int* or *None*, optional (default: *None*)) – If *None*, defaults to *self.knn*
- **bandwidth** (*float*, *callable*, or *None*, optional (default: *None*)) – If *None*, defaults to *self.bandwidth*
- **bandwidth_scale** (*float*, optional (default : *None*)) – Rescaling factor for bandwidth. If *None*, defaults to self.bandwidth_scale

Returns **K_yx** – kernel matrix where each row represents affinities of a single sample in *Y* to all samples in *self.data*.

Return type array-like, [n_samples_y, n_samples]

Raises ValueError: if the supplied data is the wrong shape

build_landmark_op()
Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

clusters
Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*data*, ***kwargs*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

knn_tree
KNN tree object (cached)

Builds or returns the fitted KNN tree. TODO: can we be more clever than sklearn when it comes to choosing between KD tree, ball tree and brute force?

Returns **knn_tree**

Return type *sklearn.neighbors.NearestNeighbors*

landmark_op
Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - knn_max - decay - bandwidth - bandwidth_scale - distance - thresh

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions
Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

weighted

class graphtools.graphs.**kNNLandmarkPyGSPGraph**(*data*, *knn=5*, *decay=None*, *knn_max=None*, *search_multiplier=6*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *thresh=0.0001*, *n_pca=None*, ***kwargs*)
Bases: *graphtools.graphs.kNNGraph*, *graphtools.graphs.LandmarkGraph*, *graphtools.base.PyGSPGraph*

A
Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D
Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K
Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

apply_anisotropy(*K*)

build_kernel()

Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *knn=None*, *knn_max=None*, *bandwidth=None*, *bandwidth_scale=None*)

Build a kernel from new input data *Y* to the *self.data*

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **knn** (*int* or *None*, optional (default: *None*)) – If *None*, defaults to *self.knn*
- **bandwidth** (*float*, *callable*, or *None*, optional (default: *None*)) – If *None*, defaults to *self.bandwidth*
- **bandwidth_scale** (*float*, optional (default : *None*)) – Rescaling factor for bandwidth. If *None*, defaults to self.bandwidth_scale

Returns **K_yx** – kernel matrix where each row represents affinities of a single sample in *Y* to all samples in *self.data*.

Return type array-like, [n_samples_y, n_samples]

Raises ValueError: if the supplied data is the wrong shape

build_landmark_op()

Build the landmark operator

Calculates spectral clusters on the kernel, and calculates transition probabilities between cluster centers by using transition probabilities between samples assigned to each cluster.

check_weights()

Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a “not a number” value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false

- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

clusters
Cluster assignments for each sample.

Compute or return the cluster assignments

Returns **clusters** – Cluster assignments for each sample.

Return type list-like, shape=[n_samples]

compute_differential_operator()
Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)
Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U \Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)

Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d

The degree (the number of neighbors) of each node.

diff_aff

Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op

Synonym for P

div(s)

Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw
The weighted degree (the sum of weighted edges) of each node.

e
Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)
Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extend_to_data(*data*, ***kwargs*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of landmarks. Any transformation of the landmarks can be trivially applied to *Y* by performing

transform_Y = transitions.dot(transform)

Parameters **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, self.data.shape[0]]

extract_components()
Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()
Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

get_params()
Get parameters from this object

gft(*s*)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)
Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)

Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator `D`.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)

Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis `U`.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

inverse_transform(*Y*, *columns=None*)

Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

is_connected(*recompute=False*)

Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)

Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

knn_tree
KNN tree object (cached)

Builds or returns the fitted KNN tree. TODO: can we be more clever than sklearn when it comes to choosing between KD tree, ball tree and brute force?

Returns **knn_tree**

Return type *sklearn.neighbors.NearestNeighbors*

landmark_op
Landmark operator

Compute or return the landmark operator

Returns **landmark_op** – Landmark operator. Can be treated as a diffusion operator between landmarks.

Return type array-like, shape=[n_landmark, n_landmark]

lmax
Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate (*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot(***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - knn_max - decay - bandwidth - bandwidth_scale - distance - thresh

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

subgraph(*ind*)

Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

symmetrize_kernel(*K*)

to_igraph(*attribute='weight', **kwargs*)

Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)

Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)

Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)

Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

transitions

Transition matrix from samples to landmarks

Compute the landmark operator if necessary, then return the transition matrix.

Returns **transitions** – Transition probabilities between samples and landmarks.

Return type array-like, shape=[n_samples, n_landmark]

translate(*f*, *i*)

Translate the signal *f* to the node *i*.

Parameters

- **f** (*ndarray*) – Signal
- **i** (*int*) – Indices of vertex

Returns **ft**

Return type translate signal

weighted

class graphtools.graphs.**kNNPyGSPGraph**(*data*, *knn=5*, *decay=None*, *knn_max=None*, *search_multiplier=6*, *bandwidth=None*, *bandwidth_scale=1.0*, *distance='euclidean'*, *thresh=0.0001*, *n_pca=None*, ***kwargs*)

Bases: *graphtools.graphs.kNNGraph*, *graphtools.base.PyGSPGraph*

A

Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D

Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

apply_anisotropy(*K*)

build_kernel()

Build the KNN kernel.

Build a k nearest neighbors kernel, optionally with alpha decay. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*, *knn=None*, *knn_max=None*, *bandwidth=None*, *bandwidth_scale=None*)

Build a kernel from new input data *Y* to the *self.data*

Parameters

- **Y** (*array-like, [n_samples_y, n_features]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions
- **knn** (*int* or *None*, optional (default: *None*)) – If *None*, defaults to *self.knn*
- **bandwidth** (*float*, *callable*, or *None*, optional (default: *None*)) – If *None*, defaults to *self.bandwidth*
- **bandwidth_scale** (*float*, optional (default : *None*)) – Rescaling factor for bandwidth. If *None*, defaults to self.bandwidth_scale

Returns **K_yx** – kernel matrix where each row represents affinities of a single sample in *Y* to all samples in *self.data*.

Return type array-like, [n_samples_y, n_samples]

Raises ValueError: if the supplied data is the wrong shape

check_weights()

Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a "not a number" value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false
- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

compute_differential_operator()

Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)

Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U \Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)

Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d

The degree (the number of neighbors) of each node.

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

div(*s*)
Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw
The weighted degree (the sum of weighted edges) of each node.

e
Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)
Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extend_to_data(*Y*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

extract_components()
Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
```

(continues on next page)

(continued from previous page)

```
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()
Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

get_params()
Get parameters from this object

gft(*s*)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).

- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)
Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)
Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)

Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis `U`.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

interpolate(*transform*, *transitions=None*, *Y=None*)

Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

is_connected(*recompute=False*)
Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)
Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

knn_tree
KNN tree object (cached)

Builds or returns the fitted KNN tree. TODO: can we be more clever than sklearn when it comes to choosing between KD tree, ball tree and brute force?

Returns **knn_tree**

Return type *sklearn.neighbors.NearestNeighbors*

lmax
Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate(*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot(***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - random_state - verbose Invalid parameters: (these would require modifying the kernel matrix) - knn - knn_max - decay - bandwidth - bandwidth_scale - distance - thresh

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

subgraph(*ind*)
Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

translate(*f*, *i*)
Translate the signal *f* to the node *i*.

Parameters

- **f** (*ndarray*) – Signal

- **i** (*int*) – Indices of vertex

Returns **ft**

Return type translate signal

weighted

2.3 Base Classes

class graphtools.base.**Base**

Bases: object

Class that deals with key-word arguments but is otherwise just an object.

set_params(***kwargs*)

class graphtools.base.**BaseGraph**(*kernel_symm='+'*, *theta=None*, *anisotropy=0*, *gamma=None*, *initialize=True*, ***kwargs*)

Bases: *graphtools.base.Base*

Parent graph class

Parameters

- **kernel_symm** (*string, optional (default: '+')*) – Defines method of kernel symmetrization. '+' : additive '*' : multiplicative 'mnn' : min-max MNN symmetrization 'none' : no symmetrization
- **theta** (*float (default: 1)*) – Min-max symmetrization constant. K = *theta * min(K, K.T) + (1 - theta) * max(K, K.T)*
- **anisotropy** (*float, optional (default: 0)*) – Level of anisotropy between 0 and 1 (alpha in Coifman & Lafon, 2006)
- **initialize** (*bool*, optional (default : *True*)) – if false, don't create the kernel matrix.

K

kernel matrix defined as the adjacency matrix with ones down the diagonal

Type array-like, shape=[n_samples, n_samples]

kernel

Type synonym for *K*

P

diffusion operator defined as a row-stochastic form of the kernel matrix

Type array-like, shape=[n_samples, n_samples] (cached)

diff_op

Type synonym for *P*

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P
Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()
Build the kernel matrix

Abstract method that all child classes must implement. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

get_params()
Get parameters from this object

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: Invalid parameters: (these would require modifying the kernel matrix) - kernel_symm - theta

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

weighted

class graphtools.base.**Data**(*data*, *n_pca=None*, *rank_threshold=None*, *random_state=None*, ***kwargs*)
Bases: *graphtools.base.Base*

Parent class that handles the import and dimensionality reduction of data

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix*. *pandas.DataFrame*, *pandas.SparseDataFrame*.

- **n_pca** ({*int*, *None*, *bool*, 'auto'}, optional (default: *None*)) – number of PC dimensions to retain for graph building. If n_pca in *[None, False, 0]*, uses the original data. If 'auto' or *True* then estimate using a singular value threshold Note: if data is sparse, uses SVD instead of PCA TODO: should we subtract and store the mean?
- **rank_threshold** (*float*, 'auto', optional (default: 'auto')) – threshold to use when estimating rank for *n_pca in [True, 'auto']*. If 'auto', this threshold is s_max * eps * max(n_samples, n_features) where s_max is the maximum singular value of the data matrix and eps is numerical precision. [press2007].
- **random_state** (*int* or *None*, optional (default: *None*)) – Random state for random PCA

data
Original data matrix

Type array-like, shape=[n_samples,n_features]

n_pca

Type int or *None*

data_nu
Reduced data matrix

Type array-like, shape=[n_samples,n_pca]

data_pca
sklearn PCA operator

Type sklearn.decomposition.PCA or sklearn.decomposition.TruncatedSVD

get_params()
Get parameters from this object

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_pca - random_state

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

transform(*Y*)

Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

class graphtools.base.**DataGraph**(*data*, *verbose=True*, *n_jobs=1*, ***kwargs*)

Bases: *graphtools.base.Data*, *graphtools.base.BaseGraph*

Abstract class for graphs built from a dataset

Parameters

- **data** (*array-like, shape=[n_samples,n_features]*) – accepted types: *numpy.ndarray*, *scipy.sparse.spmatrix*.
- **n_pca** ({*int*, *None*, *bool*, 'auto'}, optional (default: *None*)) – number of PC dimensions to retain for graph building. If n_pca in *[None,False,0]*, uses the original data. If *True* then estimate using a singular value threshold Note: if data is sparse, uses SVD instead of PCA TODO: should we subtract and store the mean?
- **rank_threshold** (*float*, 'auto', optional (default: 'auto')) – threshold to use when estimating rank for *n_pca in [True, 'auto']*. Note that the default kwarg is *None* for this parameter. It is subsequently parsed to 'auto' if necessary. If 'auto', this threshold is smax * np.finfo(data.dtype).eps * max(data.shape) where smax is the maximum singular value of the data matrix. For reference, see, e.g. W. Press, S. Teukolsky, W. Vetterling and B. Flannery, "Numerical Recipes (3rd edition)", Cambridge University Press, 2007, page 795.
- **random_state** (*int* or *None*, optional (default: *None*)) – Random state for random PCA and graph building
- **verbose** (*bool*, optional (default: *True*)) – Verbosity.
- **n_jobs** (*int*, optional (default : 1)) – The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

P

Diffusion operator (cached)

Return or calculate the diffusion operator

Returns **P** – diffusion operator defined as a row-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

apply_anisotropy(*K*)

build_kernel()
Build the kernel matrix

Abstract method that all child classes must implement. Must return a symmetric matrix

Returns **K** – symmetric matrix with ones down the diagonal with no non-negative entries.

Return type kernel matrix, shape=[n_samples, n_samples]

build_kernel_to_data(*Y*)
Build a kernel from new input data *Y* to the *self.data*

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **K_yx** – kernel matrix where each row represents affinities of a single sample in *Y* to all samples in *self.data*.

Return type array-like, [n_samples_y, n_samples]

Raises

- ValueError: if this Graph is not capable of extension or
- if the supplied data is the wrong shape

diff_aff
Symmetric diffusion affinity matrix

Return or calculate the symmetric diffusion affinity matrix

$$A(x, y) = K(x, y)(d(x)d(y))^{-1/2}$$

where d is the degrees (row sums of the kernel.)

Returns **diff_aff** – symmetric diffusion affinity matrix defined as a doubly-stochastic form of the kernel matrix

Return type array-like, shape=[n_samples, n_samples]

diff_op
Synonym for P

extend_to_data(*Y*)
Build transition matrix from new data to the graph

Creates a transition matrix such that *Y* can be approximated by a linear combination of samples in *self.data*. Any transformation of *self.data* can be trivially applied to *Y* by performing

transform_Y = self.interpolate(transform, transitions)

Parameters **Y** (*array-like, [n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **transitions** – Transition matrix from *Y* to *self.data*

Return type array-like, shape=[n_samples_y, self.data.shape[0]]

get_params()
Get parameters from this object

interpolate(*transform*, *transitions=None*, *Y=None*)
Interpolate new data onto a transformation of the graph data

One of either transitions or Y should be provided

Parameters

- **transform** (*array-like, shape=[n_samples, n_transform_features]*) –
- **transitions** (*array-like, optional, shape=[n_samples_y, n_samples]*) – Transition matrix from *Y* (not provided) to *self.data*
- **Y** (*array-like, optional, shape=[n_samples_y, n_dimensions]*) – new data for which an affinity matrix is calculated to the existing data. *n_features* must match either the ambient or PCA dimensions

Returns **Y_transform** – Transition matrix from *Y* to *self.data*

Return type array-like, [n_samples_y, n_features or n_pca]

Raises ValueError: if neither *transitions* nor *Y* is provided

inverse_transform(*Y*, *columns=None*)
Transform input data *Y* to ambient data space defined by *self.data*

Takes data in the same reduced space as *self.data_nu* and transforms it to be in the same ambient space as *self.data*.

Parameters

- **Y** (*array-like, shape=[n_samples_y, n_pca]*) – n_features must be the same as *self.data_nu*.
- **columns** (*list-like*) – list of integers referring to column indices in the original data space to be returned. Avoids recomputing the full matrix where only a few dimensions of the ambient space are of interest

Returns

Return type Inverse transformed data, shape=[n_samples_y, n_features]

Raises ValueError : if Y.shape[1] != self.data_nu.shape[1]

kernel
Synonym for K

kernel_degree
Weighted degree vector (cached)

Return or calculate the degree vector from the affinity matrix

Returns **degrees** – Row sums of graph kernel

Return type array-like, shape=[n_samples]

set_params(***params*)
Set parameters on this object

Safe setter method - attributes should not be modified directly as some changes are not valid. Valid parameters: - n_jobs - verbose

Parameters **params** (*key-value pairs of parameter name and new values*) –

Returns

Return type self

shortest_path(*method='auto'*, *distance=None*)
Find the length of the shortest path between every pair of vertices on the graph

Parameters

- **method** (*string ['auto'|'FW'|'D']*) – method to use. Options are 'auto' : attempt to choose the best method for the current problem 'FW' : Floyd-Warshall algorithm. O[N^3] 'D' : Dijkstra's algorithm with Fibonacci stacks. O[(k+log(N))N^2]
- **distance** (*{'constant', 'data', 'affinity'}, optional (default: 'data')*) – Distances along kNN edges. 'constant' gives constant edge lengths. 'data' gives distances in ambient data space. 'affinity' gives distances as negative log affinities.

Returns **D** – D[i,j] gives the shortest distance from point i to point j along the graph. If no path exists, the distance is np.inf

Return type np.ndarray, float, shape = [N,N]

Notes

Currently, shortest paths can only be calculated on kNNGraphs with *decay=None*

symmetrize_kernel(*K*)

to_igraph(*attribute='weight'*, ***kwargs*)
Convert to an igraph Graph

Uses the igraph.Graph constructor

Parameters

- **attribute** (*str, optional (default: "weight")*) –
- **kwargs** (*additional arguments for igraph.Graph*) –

to_pickle(*path*)
Save the current Graph to a pickle.

Parameters **path** (*str*) – File path where the pickled object will be stored.

to_pygsp(***kwargs*)
Convert to a PyGSP graph

For use only when the user means to create the graph using the flag *use_pygsp=True*, and doesn't wish to recompute the kernel. Creates a graphtools.graphs.TraditionalGraph with a precomputed affinity matrix which also inherits from pygsp.graphs.Graph.

Parameters **kwargs** – keyword arguments for graphtools.Graph

Returns **G**

Return type *graphtools.base.PyGSPGraph*, *graphtools.graphs.TraditionalGraph*

transform(*Y*)
Transform input data *Y* to reduced data space defined by *self.data*

Takes data in the same ambient space as *self.data* and transforms it to be in the same reduced space as *self.data_nu*.

Parameters **Y** (*array-like, shape=[n_samples_y, n_features]*) – n_features must be the same as *self.data*.

Returns

Return type Transformed data, shape=[n_samples_y, n_pca]

Raises ValueError : if Y.shape[1] != self.data.shape[1]

weighted

class graphtools.base.**PyGSPGraph**(*lap_type='combinatorial'*, *coords=None*, *plotting=None*, ***kwargs*)

Bases: pygsp.graphs.graph.Graph, *graphtools.base.Base*

Interface between BaseGraph and PyGSP.

All graphs should possess these matrices. We inherit a lot of functionality from pygsp.graphs.Graph.

There is a lot of overhead involved in having both a weight and kernel matrix

A

Graph adjacency matrix (the binary version of W).

The adjacency matrix defines which edges exist on the graph. It is represented as an N-by-N matrix of booleans. $A_{i,j}$ is True if $W_{i,j} > 0$.

D

Differential operator (for gradient and divergence).

Is computed by *compute_differential_operator()*.

K

Kernel matrix

Returns **K** – kernel matrix defined as the adjacency matrix with ones down the diagonal

Return type array-like, shape=[n_samples, n_samples]

U

Fourier basis (eigenvectors of the Laplacian).

Is computed by *compute_fourier_basis()*.

check_weights()

Check the characteristics of the weights matrix.

Returns

- *A dict of bools containing informations about the matrix*
- **has_inf_val** (*bool*) – True if the matrix has infinite values else false
- **has_nan_value** (*bool*) – True if the matrix has a "not a number" value else false
- **is_not_square** (*bool*) – True if the matrix is not square else false
- **diag_is_not_zero** (*bool*) – True if the matrix diagonal has not only zeros else false

Examples

```
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
>>> cw = G.check_weights()
>>> cw == {'has_inf_val': False, 'has_nan_value': False,
...        'is_not_square': False, 'diag_is_not_zero': True}
True
```

compute_differential_operator()
Compute the graph differential operator (cached).

The differential operator is a matrix such that

$$L = D^T D,$$

where D is the differential operator and L is the graph Laplacian. It is used to compute the gradient and the divergence of a graph signal, see *grad()* and *div()*.

The result is cached and accessible by the *D* property.

See also:

grad() compute the gradient

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> G.compute_differential_operator()
>>> G.D.shape == (G.Ne, G.N)
True
```

compute_fourier_basis(*recompute=False*)
Compute the Fourier basis of the graph (cached).

The result is cached and accessible by the *U*, *e*, *lmax*, and *mu* properties.

Parameters **recompute** (*bool*) – Force to recompute the Fourier basis if already existing.

Notes

'G.compute_fourier_basis()' computes a full eigendecomposition of the graph Laplacian L such that:

$$L = U \Lambda U^*,$$

where Λ is a diagonal matrix of eigenvalues and the columns of U are the eigenvectors.

G.e is a vector of length *G.N* containing the Laplacian eigenvalues. The largest eigenvalue is stored in *G.lmax*. The eigenvectors are stored as column vectors of *G.U* in the same order that the eigenvalues. Finally, the coherence of the Fourier basis is found in *G.mu*.

References

See [chung1997spectral].

Examples

```
>>> G = graphs.Torus()
>>> G.compute_fourier_basis()
>>> G.U.shape
(256, 256)
>>> G.e.shape
(256,)
>>> G.lmax == G.e[-1]
True
>>> G.mu < 1
True
```

compute_laplacian(*lap_type='combinatorial'*)

Compute a graph Laplacian.

The result is accessible by the L attribute.

Parameters **lap_type** (*'combinatorial', 'normalized'*) – The type of Laplacian to compute. Default is combinatorial.

Notes

For undirected graphs, the combinatorial Laplacian is defined as

$$L = D - W,$$

where W is the weight matrix and D the degree matrix, and the normalized Laplacian is defined as

$$L = I - D^{-1/2} W D^{-1/2},$$

where I is the identity matrix.

Examples

```
>>> G = graphs.Sensor(50)
>>> G.L.shape
(50, 50)
>>>
>>> G.compute_laplacian('combinatorial')
>>> G.compute_fourier_basis()
>>> -1e-10 < G.e[0] < 1e-10  # Smallest eigenvalue close to 0.
True
>>>
>>> G.compute_laplacian('normalized')
>>> G.compute_fourier_basis(recompute=True)
>>> -1e-10 < G.e[0] < 1e-10 < G.e[-1] < 2  # Spectrum in [0, 2].
True
```

d

The degree (the number of neighbors) of each node.

div(*s*)

Compute the divergence of a graph signal.

The divergence of a signal s is defined as

$$y = D^T s,$$

where D is the differential operator `D`.

Parameters **s** (*ndarray*) – Signal of length G.Ne/2 living on the edges (non-directed graph).

Returns **s_div** – Divergence signal of length G.N living on the nodes.

Return type ndarray

See also:

compute_differential_operator()

grad() compute the gradient

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.Ne)
>>> s_div = G.div(s)
>>> s_grad = G.grad(s_div)
```

dw

The weighted degree (the sum of weighted edges) of each node.

e

Eigenvalues of the Laplacian (square of graph frequencies).

Is computed by *compute_fourier_basis()*.

estimate_lmax(*recompute=False*)

Estimate the Laplacian's largest eigenvalue (cached).

The result is cached and accessible by the *lmax* property.

Exact value given by the eigendecomposition of the Laplacian, see *compute_fourier_basis()*. That estimation is much faster than the eigendecomposition.

Parameters **recompute** (*boolean*) – Force to recompute the largest eigenvalue. Default is false.

Notes

Runs the implicitly restarted Lanczos method with a large tolerance, then increases the calculated largest eigenvalue by 1 percent. For much of the PyGSP machinery, we need to approximate wavelet kernels on an interval that contains the spectrum of L. The only cost of using a larger interval is that the polynomial approximation over the larger interval may be a slightly worse approximation on the actual spectrum. As this is a very mild effect, it is not necessary to obtain very tight bounds on the spectrum of L.

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> print('{:.2f}'.format(G.lmax))
13.78
>>> G = graphs.Logo()
>>> G.estimate_lmax(recompute=True)
>>> print('{:.2f}'.format(G.lmax))
13.92
```

extract_components()
Split the graph into connected components.

See *is_connected()* for the method used to determine connectedness.

Returns **graphs** – A list of graph structures. Each having its own node list and weight matrix. If the graph is directed, add into the info parameter the information about the source nodes and the sink nodes.

Return type list

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> W = utils.symmetrize(W)
>>> G = graphs.Graph(W=W)
>>> components = G.extract_components()
>>> has_sinks = 'sink' in components[0].info
>>> sinks_0 = components[0].info['sink'] if has_sinks else []
```

get_edge_list()
Return an edge list, an alternative representation of the graph.

The weighted adjacency matrix is the canonical form used in this package to represent a graph as it is the easiest to work with when considering spectral methods.

Returns

- **v_in** (*vector of int*)
- **v_out** (*vector of int*)
- **weights** (*vector of float*)

Examples

```
>>> G = graphs.Logo()
>>> v_in, v_out, weights = G.get_edge_list()
>>> v_in.shape, v_out.shape, weights.shape
((3131,), (3131,), (3131,))
```

gft(s)
Compute the graph Fourier transform.

The graph Fourier transform of a signal s is defined as

$$\hat{s} = U^* s,$$

where U is the Fourier basis attr:U and U^* denotes the conjugate transpose or Hermitian transpose of U.

Parameters **s** (*ndarray*) – Graph signal in the vertex domain.

Returns **s_hat** – Representation of s in the Fourier domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s = np.random.normal(size=(G.N, 5, 1))
>>> s_hat = G.gft(s)
>>> s_star = G.igft(s_hat)
>>> np.all((s - s_star) < 1e-10)
True
```

gft_windowed(*g*, *f*, *lowmemory=True*)
Windowed graph Fourier transform.

Parameters

- **g** (*ndarray or Filter*) – Window (graph signal or kernel).
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory (default=True).

Returns **C** – Coefficients.

Return type ndarray

gft_windowed_gabor(*s*, *k*)
Gabor windowed graph Fourier transform.

Parameters

- **s** (*ndarray*) – Graph signal in the vertex domain.
- **k** (*function*) – Gabor kernel. See `pygsp.filters.Gabor`.

Returns **s** – Vertex-frequency representation of the signals.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> s = np.random.normal(size=(G.N, 2))
>>> s = G.gft_windowed_gabor(s, lambda x: x/(1.-x))
>>> s.shape
(1130, 2, 1130)
```

gft_windowed_normalized(*g*, *f*, *lowmemory=True*)
Normalized windowed graph Fourier transform.

Parameters

- **g** (*ndarray*) – Window.
- **f** (*ndarray*) – Graph signal in the vertex domain.
- **lowmemory** (*bool*) – Use less memory. (default = True)

Returns **C** – Coefficients.

Return type ndarray

grad(*s*)

Compute the gradient of a graph signal.

The gradient of a signal s is defined as

$$y = Ds,$$

where D is the differential operator *D*.

Parameters **s** (*ndarray*) – Signal of length G.N living on the nodes.

Returns **s_grad** – Gradient signal of length G.Ne/2 living on the edges (non-directed graph).

Return type ndarray

See also:

compute_differential_operator()

div() compute the divergence

Examples

```
>>> G = graphs.Logo()
>>> G.N, G.Ne
(1130, 3131)
>>> s = np.random.normal(size=G.N)
>>> s_grad = G.grad(s)
>>> s_div = G.div(s_grad)
>>> np.linalg.norm(s_div - G.L.dot(s)) < 1e-10
True
```

igft(*s_hat*)

Compute the inverse graph Fourier transform.

The inverse graph Fourier transform of a Fourier domain signal $\hat{s}$ is defined as

$$s = U\hat{s},$$

where U is the Fourier basis *U*.

Parameters **s_hat** (*ndarray*) – Graph signal in the Fourier domain.

Returns **s** – Representation of s_hat in the vertex domain.

Return type ndarray

Examples

```
>>> G = graphs.Logo()
>>> G.compute_fourier_basis()
>>> s_hat = np.random.normal(size=(G.N, 5, 1))
>>> s = G.igft(s_hat)
>>> s_hat_star = G.gft(s)
>>> np.all((s_hat - s_hat_star) < 1e-10)
True
```

is_connected(*recompute=False*)
Check the strong connectivity of the graph (cached).

It uses DFS travelling on graph to ensure that each node is visited. For undirected graphs, starting at any vertex and trying to access all others is enough. For directed graphs, one needs to check that a random vertex is accessible by all others and can access all others. Thus, we can transpose the adjacency matrix and compute again with the same starting point in both phases.

Parameters **recompute** (*bool*) – Force to recompute the connectivity if already known.

Returns **connected** – True if the graph is connected.

Return type bool

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> connected = G.is_connected()
```

is_directed(*recompute=False*)
Check if the graph has directed edges (cached).

In this framework, we consider that a graph is directed if and only if its weight matrix is non symmetric.

Parameters **recompute** (*bool*) – Force to recompute the directedness if already known.

Returns **directed** – True if the graph is directed.

Return type bool

Notes

Can also be used to check if a matrix is symmetrical

Examples

```
>>> from scipy import sparse
>>> W = sparse.rand(10, 10, 0.2)
>>> G = graphs.Graph(W=W)
>>> directed = G.is_directed()
```

lmax
Largest eigenvalue of the graph Laplacian.

Can be exactly computed by *compute_fourier_basis()* or approximated by *estimate_lmax()*.

modulate(*f*, *k*)
Modulate the signal *f* to the frequency *k*.

Parameters

- **f** (*ndarray*) – Signal (column)
- **k** (*int*) – Index of frequencies

Returns **fm** – Modulated signal

Return type ndarray

mu
Coherence of the Fourier basis.

Is computed by *compute_fourier_basis()*.

plot(***kwargs*)
Plot the graph.

See `pygsp.plotting.plot_graph()`.

plot_signal(*signal*, ***kwargs*)
Plot a signal on that graph.

See `pygsp.plotting.plot_signal()`.

plot_spectrogram(***kwargs*)
Plot the graph's spectrogram.

See `pygsp.plotting.plot_spectrogram()`.

set_coordinates(*kind='spring'*, ***kwargs*)
Set node's coordinates (their position when plotting).

Parameters

- **kind** (*string or array-like*) – Kind of coordinates to generate. It controls the position of the nodes when plotting the graph. Can either pass an array of size Nx2 or Nx3 to set the coordinates manually or the name of a layout algorithm. Available algorithms: community2D, random2D, random3D, ring2D, line1D, spring. Default is 'spring'.
- **kwargs** (*dict*) – Additional parameters to be passed to the Fruchterman-Reingold force-directed algorithm when kind is spring.

Examples

```
>>> G = graphs.ErdosRenyi()
>>> G.set_coordinates()
>>> G.plot()
```

set_params(***kwargs*)

subgraph(*ind*)
Create a subgraph given indices.

Parameters **ind** (*list*) – Nodes to keep

Returns **sub_G** – Subgraph

Return type Graph

Examples

```
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [1, 3]
>>> sub_G = G.subgraph(ind)
```

translate(*f*, *i*)
Translate the signal f to the node i.

Parameters

- **f** (*ndarray*) – Signal
- **i** (*int*) – Indices of vertex

Returns ft

Return type translate signal

2.4 Utilities

graphtools.utils.**check_between**(*v_min*, *v_max*, ***params*)
Checks parameters are in a specified range

Parameters

- **v_min** (*float, minimum allowed value (inclusive)*) –
- **v_max** (*float, maximum allowed value (inclusive)*) –
- **params** (*object*) – Named arguments, parameters to be checked

Raises ValueError : unacceptable choice of parameters

graphtools.utils.**check_greater**(*x*, ***params*)
Check that parameters are greater than x as expected

Parameters **x** (*excepted boundary*) – Checks not run if parameters are greater than x

Raises ValueError : unacceptable choice of parameters

graphtools.utils.**check_if_not**(*x*, **checks*, ***params*)
Run checks only if parameters are not equal to a specified value

Parameters

- **x** (*excepted value*) – Checks not run if parameters equal x
- **checks** (*function*) – Unnamed arguments, check functions to be run
- **params** (*object*) – Named arguments, parameters to be checked

Raises ValueError : unacceptable choice of parameters

graphtools.utils.**check_in**(*choices*, ***params*)
Checks parameters are in a list of allowed parameters

Parameters

- **choices** (*array-like, accepted values*) –
- **params** (*object*) – Named arguments, parameters to be checked

Raises ValueError : unacceptable choice of parameters

graphtools.utils.**check_int**(***params*)
Check that parameters are integers as expected

Raises ValueError : unacceptable choice of parameters

graphtools.utils.**check_positive**(***params*)
Check that parameters are positive as expected

Raises ValueError : unacceptable choice of parameters

graphtools.utils.**dense_nonzero_discrete**(**args*, ***kwargs*)

graphtools.utils.**dense_set_diagonal**(**args*, ***kwargs*)

graphtools.utils.**elementwise_maximum**(**args*, ***kwargs*)

graphtools.utils.**elementwise_minimum**(**args*, ***kwargs*)

graphtools.utils.**if_sparse**(**args*, ***kwargs*)

graphtools.utils.**is_Anndata**(*X*)

graphtools.utils.**is_DataFrame**(*X*)

graphtools.utils.**is_SparseDataFrame**(*X*)

graphtools.utils.**matrix_is_equivalent**(**args*, ***kwargs*)

graphtools.utils.**nonzero_discrete**(**args*, ***kwargs*)

graphtools.utils.**set_diagonal**(**args*, ***kwargs*)

graphtools.utils.**set_submatrix**(**args*, ***kwargs*)

graphtools.utils.**sparse_maximum**(**args*, ***kwargs*)

graphtools.utils.**sparse_minimum**(**args*, ***kwargs*)

graphtools.utils.**sparse_nonzero_discrete**(**args*, ***kwargs*)

graphtools.utils.**sparse_set_diagonal**(**args*, ***kwargs*)

graphtools.utils.**to_array**(**args*, ***kwargs*)

CHAPTER 3

Quick Start

To use *graphtools*, create a *graphtools.Graph* class:

```
from sklearn import datasets
import graphtools
digits = datasets.load_digits()
G = graphtools.Graph(digits['data'])
K = G.kernel
P = G.diff_op
G = graphtools.Graph(digits['data'], n_landmark=300)
L = G.landmark_op
```

To use *graphtools* with *pygsp*, create a *graphtools.Graph* class with *use_pygsp=True*:

```
from sklearn import datasets
import graphtools
digits = datasets.load_digits()
G = graphtools.Graph(digits['data'], use_pygsp=True)
N = G.N
W = G.W
basis = G.compute_fourier_basis()
```

CHAPTER 4

Help

If you have any questions or require assistance using graphtools, please contact us at https://krishnaswamylab.org/get-help

Bibliography

[press2007] W. Press, S. Teukolsky, W. Vetterling and B. Flannery, “Numerical Recipes (3rd edition)”, Cambridge University Press, 2007, page 795.

Python Module Index

g

Index

A

B

C

D

E

F

G

I

K

L

M

N

P

R

S

T

U

W