

---

# **graphtools Documentation**

*Release 1.5.2*

**Scott Gigante and Jay Stanley, Yale University**

**Aug 10, 2020**



---

## Contents

---

<b>1</b>	<b>Installation</b>	<b>3</b>
1.1	Installation with <i>pip</i> . . . . .	3
1.2	Installation from source . . . . .	3
<b>2</b>	<b>Reference</b>	<b>5</b>
2.1	API . . . . .	5
2.2	Graph Classes . . . . .	7
2.3	Base Classes . . . . .	112
2.4	Utilities . . . . .	129
<b>3</b>	<b>Quick Start</b>	<b>131</b>
<b>4</b>	<b>Help</b>	<b>133</b>
	<b>Bibliography</b>	<b>135</b>
	<b>Python Module Index</b>	<b>137</b>
	<b>Index</b>	<b>139</b>



Tools for building and manipulating graphs in Python.



### 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
```





## 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  $s_{\max} * \epsilon * \max(n_{\text{samples}}, n_{\text{features}})$  where  $s_{\max}$  is the maximum singular value of the data matrix and  $\epsilon$  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  $n \times n$  distance matrix and returns 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<sub>yx</sub>** – 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` (*dict* (`{'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

**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, band-
                                          width=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  $Y.shape[1] \neq self.data\_nu.shape[1]$

**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, adap-
                                             tive_k=None, bandwidth=None, dis-
                                             tance='euclidean', thresh=0.0001,
                                             n_jobs=1, **kwargs)
```

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

**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  $\Lambda$  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** (*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  $N \times 2$  or  $N \times 3$  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  $\Lambda$  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

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

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. 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** (*dict* {'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  $n \times m$  matrix and returns 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  $\text{smax} * \text{np.finfo}(\text{data.dtype}).\text{eps} * \text{max}(\text{data.shape})$  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

**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  $\text{np.inf}$

**Return type**  $\text{np.ndarray}$ , 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

**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  $Y.shape[1] \neq self.data.shape[1]$

**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  $\Lambda$  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** (*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  $\Lambda$  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  $N \times 2$  or  $N \times 3$  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  $Y.shape[1] \neq self.data.shape[1]$

**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<sub>yx</sub>** – 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<sub>yx</sub>** – 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  $\text{np.inf}$

**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.base.PyGSPGraph`

`graphtools.graphs.LandmarkGraph`,

**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<sub>yx</sub>** – 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  $\Lambda$  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** (*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  $Y.shape[1] \neq self.data_nu.shape[1]$

**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  $N \times 2$  or  $N \times 3$  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<sub>yx</sub>** – 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  $\Lambda$  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()
```

(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**  $s_{\text{hat}}$  – 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  $N \times 2$  or  $N \times 3$  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 `graphtools.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  $s_{\max} * \epsilon * \max(n_{\text{samples}}, n_{\text{features}})$  where  $s_{\max}$  is the maximum singular value of the data matrix and  $\epsilon$  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  $Y.shape[1] \neq self.data_nu.shape[1]$ **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** (*dict* {'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  $\Lambda$  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.

**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  $N \times 2$  or  $N \times 3$  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)
```

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.



**g**

`graphtools.api`, 5  
`graphtools.base`, 112  
`graphtools.graphs`, 7  
`graphtools.utils`, 129



## A

A (*graphtools.base.PyGSPGraph* attribute), 120  
 A (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 86  
 A (*graphtools.graphs.kNNPyGSPGraph* attribute), 99  
 A (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 19  
 A (*graphtools.graphs.MNNPyGSPGraph* attribute), 32  
 A (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 53  
 A (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 65  
`apply_anisotropy()` (*graphtools.base.BaseGraph* method), 113  
`apply_anisotropy()` (*graphtools.base.DataGraph* method), 116  
`apply_anisotropy()` (*graphtools.graphs.kNNGraph* method), 79  
`apply_anisotropy()` (*graphtools.graphs.kNNLandmarkGraph* method), 82  
`apply_anisotropy()` (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 87  
`apply_anisotropy()` (*graphtools.graphs.kNNPyGSPGraph* method), 100  
`apply_anisotropy()` (*graphtools.graphs.LandmarkGraph* method), 8  
`apply_anisotropy()` (*graphtools.graphs.MNNGraph* method), 12  
`apply_anisotropy()` (*graphtools.graphs.MNNLandmarkGraph* method), 16  
`apply_anisotropy()` (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 20  
`apply_anisotropy()` (*graphtools.graphs.MNNPyGSPGraph* method),

33  
`apply_anisotropy()` (*graphtools.graphs.TraditionalGraph* method), 45  
`apply_anisotropy()` (*graphtools.graphs.TraditionalLandmarkGraph* method), 49  
`apply_anisotropy()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 53  
`apply_anisotropy()` (*graphtools.graphs.TraditionalPyGSPGraph* method), 66

## B

Base (class in *graphtools.base*), 112  
 BaseGraph (class in *graphtools.base*), 112  
`build_kernel()` (*graphtools.base.BaseGraph* method), 113  
`build_kernel()` (*graphtools.base.DataGraph* method), 117  
`build_kernel()` (*graphtools.graphs.kNNGraph* method), 79  
`build_kernel()` (*graphtools.graphs.kNNLandmarkGraph* method), 82  
`build_kernel()` (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 87  
`build_kernel()` (*graphtools.graphs.kNNPyGSPGraph* method), 100  
`build_kernel()` (*graphtools.graphs.LandmarkGraph* method), 8  
`build_kernel()` (*graphtools.graphs.MNNGraph* method), 12  
`build_kernel()` (*graphtools.graphs.MNNLandmarkGraph* method), 16

<code>build_kernel()</code>	( <i>graphtools.graphs.MNNLandmarkPyGSPGraph method</i> ), 20	<code>build_landmark_op()</code>	( <i>graphtools.graphs.TraditionalPyGSPGraph method</i> ), 66
<code>build_kernel()</code>	( <i>graphtools.graphs.MNNPyGSPGraph method</i> ), 33	<code>build_landmark_op()</code>	( <i>graphtools.graphs.kNNLandmarkGraph method</i> ), 83
<code>build_kernel()</code>	( <i>graphtools.graphs.TraditionalGraph method</i> ), 45	<code>build_landmark_op()</code>	( <i>graphtools.graphs.kNNLandmarkPyGSPGraph method</i> ), 87
<code>build_kernel()</code>	( <i>graphtools.graphs.TraditionalLandmarkGraph method</i> ), 49	<code>build_landmark_op()</code>	( <i>graphtools.graphs.LandmarkGraph method</i> ), 8
<code>build_kernel()</code>	( <i>graphtools.graphs.TraditionalLandmarkPyGSPGraph method</i> ), 53	<code>build_landmark_op()</code>	( <i>graphtools.graphs.MNNLandmarkGraph method</i> ), 16
<code>build_kernel()</code>	( <i>graphtools.graphs.TraditionalPyGSPGraph method</i> ), 66	<code>build_landmark_op()</code>	( <i>graphtools.graphs.MNNLandmarkPyGSPGraph method</i> ), 20
<code>build_kernel_to_data()</code>	( <i>graphtools.base.DataGraph method</i> ), 117	<code>build_landmark_op()</code>	( <i>graphtools.graphs.TraditionalLandmarkGraph method</i> ), 49
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.kNNGraph method</i> ), 79	<code>build_landmark_op()</code>	( <i>graphtools.graphs.TraditionalLandmarkPyGSPGraph method</i> ), 54
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.kNNLandmarkGraph method</i> ), 82		
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.kNNLandmarkPyGSPGraph method</i> ), 87	<b>C</b>	
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.kNNPyGSPGraph method</i> ), 100	<code>check_between()</code>	(in module <i>graphtools.utils</i> ), 129
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.LandmarkGraph method</i> ), 8	<code>check_greater()</code>	(in module <i>graphtools.utils</i> ), 129
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.MNNGraph method</i> ), 12	<code>check_if_not()</code>	(in module <i>graphtools.utils</i> ), 129
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.MNNLandmarkGraph method</i> ), 16	<code>check_in()</code>	(in module <i>graphtools.utils</i> ), 129
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.MNNLandmarkPyGSPGraph method</i> ), 20	<code>check_int()</code>	(in module <i>graphtools.utils</i> ), 129
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.MNNPyGSPGraph method</i> ), 33	<code>check_positive()</code>	(in module <i>graphtools.utils</i> ), 129
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.TraditionalGraph method</i> ), 45	<code>check_weights()</code>	( <i>graphtools.base.PyGSPGraph method</i> ), 120
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.TraditionalLandmarkGraph method</i> ), 49	<code>check_weights()</code>	( <i>graphtools.graphs.kNNLandmarkPyGSPGraph method</i> ), 87
<code>build_kernel_to_data()</code>	( <i>graphtools.graphs.TraditionalLandmarkPyGSPGraph method</i> ), 53	<code>check_weights()</code>	( <i>graphtools.graphs.kNNPyGSPGraph method</i> ), 100
<code>build_kernel_to_data()</code>	( <i>graphtools</i> ), 53	<code>check_weights()</code>	( <i>graphtools.graphs.MNNLandmarkPyGSPGraph method</i> ), 20
		<code>check_weights()</code>	( <i>graphtools.graphs.MNNPyGSPGraph method</i> ), 33
		<code>check_weights()</code>	( <i>graphtools.graphs.TraditionalLandmarkPyGSPGraph method</i> ), 54
		<code>check_weights()</code>	( <i>graphtools.graphs.TraditionalPyGSPGraph method</i> ), 66
		<code>clusters</code>	( <i>graphtools.graphs.kNNLandmarkGraph attribute</i> ), 83
		<code>clusters</code>	( <i>graphtools.graphs.kNNLandmarkPyGSPGraph attribute</i> ), 88



- clusters (*graphtools.graphs.LandmarkGraph* attribute), 8, 9
- clusters (*graphtools.graphs.MNNLandmarkGraph* attribute), 16
- clusters (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 21
- clusters (*graphtools.graphs.TraditionalLandmarkGraph* attribute), 49
- clusters (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 54
- compute\_differential\_operator() (*graphtools.base.PyGSPGraph* method), 120
- compute\_differential\_operator() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 88
- compute\_differential\_operator() (*graphtools.graphs.kNNPyGSPGraph* method), 101
- compute\_differential\_operator() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 21
- compute\_differential\_operator() (*graphtools.graphs.MNNPyGSPGraph* method), 33
- compute\_differential\_operator() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 54
- compute\_differential\_operator() (*graphtools.graphs.TraditionalPyGSPGraph* method), 67
- compute\_fourier\_basis() (*graphtools.base.PyGSPGraph* method), 121
- compute\_fourier\_basis() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 88
- compute\_fourier\_basis() (*graphtools.graphs.kNNPyGSPGraph* method), 101
- compute\_fourier\_basis() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 21
- compute\_fourier\_basis() (*graphtools.graphs.MNNPyGSPGraph* method), 34
- compute\_fourier\_basis() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 55
- compute\_fourier\_basis() (*graphtools.graphs.TraditionalPyGSPGraph* method), 67
- compute\_laplacian() (*graphtools.base.PyGSPGraph* method), 122
- compute\_laplacian() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 89
- compute\_laplacian() (*graphtools.graphs.kNNPyGSPGraph* method), 102
- compute\_laplacian() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 22
- compute\_laplacian() (*graphtools.graphs.MNNPyGSPGraph* method), 35
- compute\_laplacian() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 55
- compute\_laplacian() (*graphtools.graphs.TraditionalPyGSPGraph* method), 68
- ## D
- D (*graphtools.base.PyGSPGraph* attribute), 120
- d (*graphtools.base.PyGSPGraph* attribute), 122
- D (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 86
- d (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 90
- D (*graphtools.graphs.kNNPyGSPGraph* attribute), 99
- d (*graphtools.graphs.kNNPyGSPGraph* attribute), 102
- D (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 20
- d (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 23
- D (*graphtools.graphs.MNNPyGSPGraph* attribute), 32
- d (*graphtools.graphs.MNNPyGSPGraph* attribute), 35
- D (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 53
- d (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 56
- D (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 66
- d (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 69
- Data (class in *graphtools.base*), 114
- data (*graphtools.base.Data* attribute), 115
- data\_nu (*graphtools.base.Data* attribute), 115
- data\_pca (*graphtools.base.Data* attribute), 115
- DataGraph (class in *graphtools.base*), 116
- dense\_nonzero\_discrete() (in module *graphtools.utils*), 129
- dense\_set\_diagonal() (in module *graphtools.utils*), 130
- diff\_aff (*graphtools.base.BaseGraph* attribute), 113
- diff\_aff (*graphtools.base.DataGraph* attribute), 117
- diff\_aff (*graphtools.graphs.kNNGraph* attribute), 79
- diff\_aff (*graphtools.graphs.kNNLandmarkGraph* attribute), 83

- diff\_aff (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 90
  - diff\_aff (*graphtools.graphs.kNNPyGSPGraph* attribute), 103
  - diff\_aff (*graphtools.graphs.LandmarkGraph* attribute), 9
  - diff\_aff (*graphtools.graphs.MNNGraph* attribute), 13
  - diff\_aff (*graphtools.graphs.MNNLandmarkGraph* attribute), 16
  - diff\_aff (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 23
  - diff\_aff (*graphtools.graphs.MNNPyGSPGraph* attribute), 35
  - diff\_aff (*graphtools.graphs.TraditionalGraph* attribute), 46
  - diff\_aff (*graphtools.graphs.TraditionalLandmarkGraph* attribute), 50
  - diff\_aff (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 56
  - diff\_aff (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 69
  - diff\_op (*graphtools.base.BaseGraph* attribute), 112, 113
  - diff\_op (*graphtools.base.DataGraph* attribute), 117
  - diff\_op (*graphtools.graphs.kNNGraph* attribute), 79
  - diff\_op (*graphtools.graphs.kNNLandmarkGraph* attribute), 83
  - diff\_op (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 90
  - diff\_op (*graphtools.graphs.kNNPyGSPGraph* attribute), 103
  - diff\_op (*graphtools.graphs.LandmarkGraph* attribute), 9
  - diff\_op (*graphtools.graphs.MNNGraph* attribute), 13
  - diff\_op (*graphtools.graphs.MNNLandmarkGraph* attribute), 17
  - diff\_op (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 23
  - diff\_op (*graphtools.graphs.MNNPyGSPGraph* attribute), 36
  - diff\_op (*graphtools.graphs.TraditionalGraph* attribute), 46
  - diff\_op (*graphtools.graphs.TraditionalLandmarkGraph* attribute), 50
  - diff\_op (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 56
  - diff\_op (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 69
  - div () (*graphtools.base.PyGSPGraph* method), 122
  - div () (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 90
  - div () (*graphtools.graphs.kNNPyGSPGraph* method), 103
  - div () (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 23
  - div () (*graphtools.graphs.MNNPyGSPGraph* method), 36
  - div () (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 56
  - div () (*graphtools.graphs.TraditionalPyGSPGraph* method), 69
  - dw (*graphtools.base.PyGSPGraph* attribute), 123
  - dw (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 91
  - dw (*graphtools.graphs.kNNPyGSPGraph* attribute), 103
  - dw (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 24
  - dw (*graphtools.graphs.MNNPyGSPGraph* attribute), 36
  - dw (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 57
  - dw (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 70
- E**
- e (*graphtools.base.PyGSPGraph* attribute), 123
  - e (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 91
  - e (*graphtools.graphs.kNNPyGSPGraph* attribute), 103
  - e (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 24
  - e (*graphtools.graphs.MNNPyGSPGraph* attribute), 36
  - e (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 57
  - e (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 70
  - elementwise\_maximum () (in module *graphtools.utils*), 130
  - elementwise\_minimum () (in module *graphtools.utils*), 130
  - estimate\_lmax () (*graphtools.base.PyGSPGraph* method), 123
  - estimate\_lmax () (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 91
  - estimate\_lmax () (*graphtools.graphs.kNNPyGSPGraph* method), 103
  - estimate\_lmax () (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 24
  - estimate\_lmax () (*graphtools.graphs.MNNPyGSPGraph* method), 36
  - estimate\_lmax () (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 57

- `estimate_lmax()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 70
- `extend_to_data()` (*graphtools.base.DataGraph method*), 117
- `extend_to_data()` (*graphtools.graphs.kNNGraph method*), 79
- `extend_to_data()` (*graphtools.graphs.kNNLandmarkGraph method*), 83
- `extend_to_data()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 91
- `extend_to_data()` (*graphtools.graphs.kNNPyGSPGraph method*), 104
- `extend_to_data()` (*graphtools.graphs.LandmarkGraph method*), 9
- `extend_to_data()` (*graphtools.graphs.MNNGraph method*), 13
- `extend_to_data()` (*graphtools.graphs.MNNLandmarkGraph method*), 17
- `extend_to_data()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 24
- `extend_to_data()` (*graphtools.graphs.MNNPyGSPGraph method*), 37
- `extend_to_data()` (*graphtools.graphs.TraditionalGraph method*), 46
- `extend_to_data()` (*graphtools.graphs.TraditionalLandmarkGraph method*), 50
- `extend_to_data()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 58
- `extend_to_data()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 70
- `extract_components()` (*graphtools.base.PyGSPGraph method*), 123
- `extract_components()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 92
- `extract_components()` (*graphtools.graphs.kNNPyGSPGraph method*), 104
- `extract_components()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 25
- `extract_components()` (*graphtools.graphs.MNNPyGSPGraph method*),
- `extract_components()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 58
- `extract_components()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 71
- F**
- `from_igraph()` (*in module graphtools.api*), 7
- G**
- `get_edge_list()` (*graphtools.base.PyGSPGraph method*), 124
- `get_edge_list()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 92
- `get_edge_list()` (*graphtools.graphs.kNNPyGSPGraph method*), 105
- `get_edge_list()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 25
- `get_edge_list()` (*graphtools.graphs.MNNPyGSPGraph method*), 37
- `get_edge_list()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 58
- `get_edge_list()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 71
- `get_params()` (*graphtools.base.BaseGraph method*), 113
- `get_params()` (*graphtools.base.Data method*), 115
- `get_params()` (*graphtools.base.DataGraph method*), 117
- `get_params()` (*graphtools.graphs.kNNGraph method*), 80
- `get_params()` (*graphtools.graphs.kNNLandmarkGraph method*), 84
- `get_params()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 92
- `get_params()` (*graphtools.graphs.kNNPyGSPGraph method*), 105
- `get_params()` (*graphtools.graphs.LandmarkGraph method*), 9
- `get_params()` (*graphtools.graphs.MNNGraph method*), 13
- `get_params()` (*graphtools.graphs.MNNLandmarkGraph method*), 17

`get_params()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 25  
`get_params()` (*graphtools.graphs.MNNPyGSPGraph method*), 38  
`get_params()` (*graphtools.graphs.TraditionalGraph method*), 46  
`get_params()` (*graphtools.graphs.TraditionalLandmarkGraph method*), 50  
`get_params()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 59  
`get_params()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 71  
`gft()` (*graphtools.base.PyGSPGraph method*), 124  
`gft()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 92  
`gft()` (*graphtools.graphs.kNNPyGSPGraph method*), 105  
`gft()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 25  
`gft()` (*graphtools.graphs.MNNPyGSPGraph method*), 38  
`gft()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 59  
`gft()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 71  
`gft_windowed()` (*graphtools.base.PyGSPGraph method*), 125  
`gft_windowed()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 93  
`gft_windowed()` (*graphtools.graphs.kNNPyGSPGraph method*), 105  
`gft_windowed()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 26  
`gft_windowed()` (*graphtools.graphs.MNNPyGSPGraph method*), 38  
`gft_windowed()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 59  
`gft_windowed()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 72  
`gft_windowed_gabor()` (*graphtools.base.PyGSPGraph method*), 125  
`gft_windowed_gabor()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 93  
`gft_windowed_gabor()` (*graphtools.graphs.kNNPyGSPGraph method*), 106  
`gft_windowed_gabor()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 26  
`gft_windowed_gabor()` (*graphtools.graphs.MNNPyGSPGraph method*), 38  
`gft_windowed_gabor()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 59  
`gft_windowed_gabor()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 72  
`gft_windowed_normalized()` (*graphtools.base.PyGSPGraph method*), 125  
`gft_windowed_normalized()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 93  
`gft_windowed_normalized()` (*graphtools.graphs.kNNPyGSPGraph method*), 106  
`gft_windowed_normalized()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 26  
`gft_windowed_normalized()` (*graphtools.graphs.MNNPyGSPGraph method*), 39  
`gft_windowed_normalized()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 60  
`gft_windowed_normalized()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 72  
`grad()` (*graphtools.base.PyGSPGraph method*), 125  
`grad()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 93  
`grad()` (*graphtools.graphs.kNNPyGSPGraph method*), 106  
`grad()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 26  
`grad()` (*graphtools.graphs.MNNPyGSPGraph method*), 39  
`grad()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 60  
`grad()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 72  
`Graph()` (*in module graphtools.api*), 5  
`graphtools.api` (*module*), 5  
`graphtools.base` (*module*), 112  
`graphtools.graphs` (*module*), 7  
`graphtools.utils` (*module*), 129

## I

- `if_sparse()` (in module `graphtools.utils`), 130
- `igft()` (`graphtools.base.PyGSPGraph` method), 126
- `igft()` (`graphtools.graphs.kNNLandmarkPyGSPGraph` method), 94
- `igft()` (`graphtools.graphs.kNNPyGSPGraph` method), 107
- `igft()` (`graphtools.graphs.MNNLandmarkPyGSPGraph` method), 27
- `igft()` (`graphtools.graphs.MNNPyGSPGraph` method), 39
- `igft()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` method), 60
- `igft()` (`graphtools.graphs.TraditionalPyGSPGraph` method), 73
- `interpolate()` (`graphtools.base.DataGraph` method), 117
- `interpolate()` (`graphtools.graphs.kNNGraph` method), 80
- `interpolate()` (`graphtools.graphs.kNNLandmarkGraph` method), 84
- `interpolate()` (`graphtools.graphs.kNNLandmarkPyGSPGraph` method), 94
- `interpolate()` (`graphtools.graphs.kNNPyGSPGraph` method), 107
- `interpolate()` (`graphtools.graphs.LandmarkGraph` method), 9
- `interpolate()` (`graphtools.graphs.MNNGraph` method), 13
- `interpolate()` (`graphtools.graphs.MNNLandmarkGraph` method), 17
- `interpolate()` (`graphtools.graphs.MNNLandmarkPyGSPGraph` method), 27
- `interpolate()` (`graphtools.graphs.MNNPyGSPGraph` method), 40
- `interpolate()` (`graphtools.graphs.TraditionalGraph` method), 46
- `interpolate()` (`graphtools.graphs.TraditionalLandmarkGraph` method), 50
- `interpolate()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` method), 61
- `interpolate()` (`graphtools.graphs.TraditionalPyGSPGraph` method), 73
- `inverse_transform()` (`graphtools.base.Data` method), 115
- `inverse_transform()` (`graphtools.base.DataGraph` method), 118
- `inverse_transform()` (`graphtools.graphs.kNNGraph` method), 80
- `inverse_transform()` (`graphtools.graphs.kNNLandmarkGraph` method), 84
- `inverse_transform()` (`graphtools.graphs.kNNLandmarkPyGSPGraph` method), 95
- `inverse_transform()` (`graphtools.graphs.kNNPyGSPGraph` method), 107
- `inverse_transform()` (`graphtools.graphs.LandmarkGraph` method), 10
- `inverse_transform()` (`graphtools.graphs.MNNGraph` method), 14
- `inverse_transform()` (`graphtools.graphs.MNNLandmarkGraph` method), 17
- `inverse_transform()` (`graphtools.graphs.MNNLandmarkPyGSPGraph` method), 28
- `inverse_transform()` (`graphtools.graphs.MNNPyGSPGraph` method), 40
- `inverse_transform()` (`graphtools.graphs.TraditionalGraph` method), 47
- `inverse_transform()` (`graphtools.graphs.TraditionalLandmarkGraph` method), 50
- `inverse_transform()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` method), 61
- `inverse_transform()` (`graphtools.graphs.TraditionalPyGSPGraph` method), 74
- `is_Anndata()` (in module `graphtools.utils`), 130
- `is_connected()` (`graphtools.base.PyGSPGraph` method), 126
- `is_connected()` (`graphtools.graphs.kNNLandmarkPyGSPGraph` method), 95
- `is_connected()` (`graphtools.graphs.kNNPyGSPGraph` method), 108
- `is_connected()` (`graphtools.graphs.MNNLandmarkPyGSPGraph` method), 28
- `is_connected()` (`graphtools.graphs.MNNPyGSPGraph` method), 40

`is_connected()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 61  
`is_connected()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 74  
`is_DataFrame()` (*in module graphtools.utils*), 130  
`is_directed()` (*graphtools.base.PyGSPGraph method*), 127  
`is_directed()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 95  
`is_directed()` (*graphtools.graphs.kNNPyGSPGraph method*), 108  
`is_directed()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 28  
`is_directed()` (*graphtools.graphs.MNNPyGSPGraph method*), 41  
`is_directed()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 62  
`is_directed()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 74  
`is_SparseDataFrame()` (*in module graphtools.utils*), 130

**K**

`K` (*graphtools.base.BaseGraph attribute*), 112  
`K` (*graphtools.base.DataGraph attribute*), 116  
`K` (*graphtools.base.PyGSPGraph attribute*), 120  
`K` (*graphtools.graphs.kNNGraph attribute*), 78  
`K` (*graphtools.graphs.kNNLandmarkGraph attribute*), 82  
`K` (*graphtools.graphs.kNNLandmarkPyGSPGraph attribute*), 86  
`K` (*graphtools.graphs.kNNPyGSPGraph attribute*), 99  
`K` (*graphtools.graphs.LandmarkGraph attribute*), 8  
`K` (*graphtools.graphs.MNNGraph attribute*), 12  
`K` (*graphtools.graphs.MNNLandmarkGraph attribute*), 15  
`K` (*graphtools.graphs.MNNLandmarkPyGSPGraph attribute*), 20  
`K` (*graphtools.graphs.MNNPyGSPGraph attribute*), 32  
`K` (*graphtools.graphs.TraditionalGraph attribute*), 45  
`K` (*graphtools.graphs.TraditionalLandmarkGraph attribute*), 49  
`K` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph attribute*), 53  
`K` (*graphtools.graphs.TraditionalPyGSPGraph attribute*), 66  
`kernel` (*graphtools.base.BaseGraph attribute*), 112, 113  
`kernel` (*graphtools.base.DataGraph attribute*), 118  
`kernel` (*graphtools.graphs.kNNGraph attribute*), 80  
`kernel` (*graphtools.graphs.kNNLandmarkGraph attribute*), 84  
`kernel` (*graphtools.graphs.kNNLandmarkPyGSPGraph attribute*), 96  
`kernel` (*graphtools.graphs.kNNPyGSPGraph attribute*), 109  
`kernel` (*graphtools.graphs.LandmarkGraph attribute*), 10  
`kernel` (*graphtools.graphs.MNNGraph attribute*), 14  
`kernel` (*graphtools.graphs.MNNLandmarkGraph attribute*), 18  
`kernel` (*graphtools.graphs.MNNLandmarkPyGSPGraph attribute*), 29  
`kernel` (*graphtools.graphs.MNNPyGSPGraph attribute*), 41  
`kernel` (*graphtools.graphs.TraditionalGraph attribute*), 47  
`kernel` (*graphtools.graphs.TraditionalLandmarkGraph attribute*), 51  
`kernel` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph attribute*), 62  
`kernel` (*graphtools.graphs.TraditionalPyGSPGraph attribute*), 75  
`kernel_degree` (*graphtools.base.BaseGraph attribute*), 113  
`kernel_degree` (*graphtools.base.DataGraph attribute*), 118  
`kernel_degree` (*graphtools.graphs.kNNGraph attribute*), 80  
`kernel_degree` (*graphtools.graphs.kNNLandmarkGraph attribute*), 84  
`kernel_degree` (*graphtools.graphs.kNNLandmarkPyGSPGraph attribute*), 96  
`kernel_degree` (*graphtools.graphs.kNNPyGSPGraph attribute*), 109  
`kernel_degree` (*graphtools.graphs.LandmarkGraph attribute*), 10  
`kernel_degree` (*graphtools.graphs.MNNGraph attribute*), 14  
`kernel_degree` (*graphtools.graphs.MNNLandmarkGraph attribute*), 18  
`kernel_degree` (*graphtools.graphs.MNNLandmarkPyGSPGraph attribute*), 29  
`kernel_degree` (*graphtools.graphs.MNNPyGSPGraph attribute*), 41  
`kernel_degree` (*graph-*)

- `tools.graphs.TraditionalGraph` attribute), 47
- `kernel_degree` (`graphtools.graphs.TraditionalLandmarkGraph` attribute), 51
- `kernel_degree` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` attribute), 62
- `kernel_degree` (`graphtools.graphs.TraditionalPyGSPGraph` attribute), 75
- `knn_tree` (`graphtools.graphs.kNNGraph` attribute), 78, 80
- `knn_tree` (`graphtools.graphs.kNNLandmarkGraph` attribute), 84
- `knn_tree` (`graphtools.graphs.kNNLandmarkPyGSPGraph` attribute), 96
- `knn_tree` (`graphtools.graphs.kNNPyGSPGraph` attribute), 109
- `kNNGraph` (class in `graphtools.graphs`), 78
- `kNNLandmarkGraph` (class in `graphtools.graphs`), 82
- `kNNLandmarkPyGSPGraph` (class in `graphtools.graphs`), 86
- `kNNPyGSPGraph` (class in `graphtools.graphs`), 99
- ## L
- `landmark_op` (`graphtools.graphs.kNNLandmarkGraph` attribute), 85
- `landmark_op` (`graphtools.graphs.kNNLandmarkPyGSPGraph` attribute), 96
- `landmark_op` (`graphtools.graphs.LandmarkGraph` attribute), 7, 10
- `landmark_op` (`graphtools.graphs.MNNLandmarkGraph` attribute), 18
- `landmark_op` (`graphtools.graphs.MNNLandmarkPyGSPGraph` attribute), 29
- `landmark_op` (`graphtools.graphs.TraditionalLandmarkGraph` attribute), 51
- `landmark_op` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` attribute), 62
- `LandmarkGraph` (class in `graphtools.graphs`), 7
- `lmax` (`graphtools.base.PyGSPGraph` attribute), 127
- `lmax` (`graphtools.graphs.kNNLandmarkPyGSPGraph` attribute), 96
- `lmax` (`graphtools.graphs.kNNPyGSPGraph` attribute), 109
- `lmax` (`graphtools.graphs.MNNLandmarkPyGSPGraph` attribute), 29
- `lmax` (`graphtools.graphs.MNNPyGSPGraph` attribute), 41
- `lmax` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` attribute), 62
- `lmax` (`graphtools.graphs.TraditionalPyGSPGraph` attribute), 75
- ## M
- `matrix_is_equivalent()` (in module `graphtools.utils`), 130
- `MNNGraph` (class in `graphtools.graphs`), 12
- `MNNLandmarkGraph` (class in `graphtools.graphs`), 15
- `MNNLandmarkPyGSPGraph` (class in `graphtools.graphs`), 19
- `MNNPyGSPGraph` (class in `graphtools.graphs`), 32
- `modulate()` (`graphtools.base.PyGSPGraph` method), 127
- `modulate()` (`graphtools.graphs.kNNLandmarkPyGSPGraph` method), 96
- `modulate()` (`graphtools.graphs.kNNPyGSPGraph` method), 109
- `modulate()` (`graphtools.graphs.MNNLandmarkPyGSPGraph` method), 29
- `modulate()` (`graphtools.graphs.MNNPyGSPGraph` method), 41
- `modulate()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` method), 63
- `modulate()` (`graphtools.graphs.TraditionalPyGSPGraph` method), 75
- `mu` (`graphtools.base.PyGSPGraph` attribute), 128
- `mu` (`graphtools.graphs.kNNLandmarkPyGSPGraph` attribute), 97
- `mu` (`graphtools.graphs.kNNPyGSPGraph` attribute), 109
- `mu` (`graphtools.graphs.MNNLandmarkPyGSPGraph` attribute), 29
- `mu` (`graphtools.graphs.MNNPyGSPGraph` attribute), 42
- `mu` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph` attribute), 63
- `mu` (`graphtools.graphs.TraditionalPyGSPGraph` attribute), 75
- ## N
- `n_pca` (`graphtools.base.Data` attribute), 115
- `nonzero_discrete()` (in module `graphtools.utils`), 130
- ## P
- `P` (`graphtools.base.BaseGraph` attribute), 112
- `P` (`graphtools.base.DataGraph` attribute), 116
- `P` (`graphtools.graphs.kNNGraph` attribute), 78
- `P` (`graphtools.graphs.kNNLandmarkGraph` attribute), 82
- `P` (`graphtools.graphs.kNNLandmarkPyGSPGraph` attribute), 86
- `P` (`graphtools.graphs.kNNPyGSPGraph` attribute), 99

- P (*graphtools.graphs.LandmarkGraph* attribute), 8
  - P (*graphtools.graphs.MNNGraph* attribute), 12
  - P (*graphtools.graphs.MNNLandmarkGraph* attribute), 16
  - P (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 20
  - P (*graphtools.graphs.MNNPyGSPGraph* attribute), 32
  - P (*graphtools.graphs.TraditionalGraph* attribute), 45
  - P (*graphtools.graphs.TraditionalLandmarkGraph* attribute), 49
  - P (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 53
  - P (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 66
  - plot() (*graphtools.base.PyGSPGraph* method), 128
  - plot() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 97
  - plot() (*graphtools.graphs.kNNPyGSPGraph* method), 109
  - plot() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 29
  - plot() (*graphtools.graphs.MNNPyGSPGraph* method), 42
  - plot() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 63
  - plot() (*graphtools.graphs.TraditionalPyGSPGraph* method), 75
  - plot\_signal() (*graphtools.base.PyGSPGraph* method), 128
  - plot\_signal() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 97
  - plot\_signal() (*graphtools.graphs.kNNPyGSPGraph* method), 109
  - plot\_signal() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 30
  - plot\_signal() (*graphtools.graphs.MNNPyGSPGraph* method), 42
  - plot\_signal() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 63
  - plot\_signal() (*graphtools.graphs.TraditionalPyGSPGraph* method), 75
  - plot\_spectrogram() (*graphtools.base.PyGSPGraph* method), 128
  - plot\_spectrogram() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 97
  - plot\_spectrogram() (*graphtools.graphs.kNNPyGSPGraph* method), 109
  - plot\_spectrogram() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 30
  - plot\_spectrogram() (*graphtools.graphs.MNNPyGSPGraph* method), 42
  - plot\_spectrogram() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 63
  - plot\_spectrogram() (*graphtools.graphs.TraditionalPyGSPGraph* method), 76
  - PyGSPGraph (class in *graphtools.base*), 120
- ## R
- read\_pickle() (in module *graphtools.api*), 7
- ## S
- set\_coordinates() (*graphtools.base.PyGSPGraph* method), 128
  - set\_coordinates() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 97
  - set\_coordinates() (*graphtools.graphs.kNNPyGSPGraph* method), 110
  - set\_coordinates() (*graphtools.graphs.MNNLandmarkPyGSPGraph* method), 30
  - set\_coordinates() (*graphtools.graphs.MNNPyGSPGraph* method), 42
  - set\_coordinates() (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* method), 63
  - set\_coordinates() (*graphtools.graphs.TraditionalPyGSPGraph* method), 76
  - set\_diagonal() (in module *graphtools.utils*), 130
  - set\_params() (*graphtools.base.Base* method), 112
  - set\_params() (*graphtools.base.BaseGraph* method), 113
  - set\_params() (*graphtools.base.Data* method), 115
  - set\_params() (*graphtools.base.DataGraph* method), 118
  - set\_params() (*graphtools.base.PyGSPGraph* method), 128
  - set\_params() (*graphtools.graphs.kNNGraph* method), 81
  - set\_params() (*graphtools.graphs.kNNLandmarkGraph* method), 85
  - set\_params() (*graphtools.graphs.kNNLandmarkPyGSPGraph* method), 97



- `method`), 97
- `set_params()` (`graphtools.graphs.kNNPyGSPGraph method`), 110
- `set_params()` (`graphtools.graphs.LandmarkGraph method`), 10
- `set_params()` (`graphtools.graphs.MNNGraph method`), 14
- `set_params()` (`graphtools.graphs.MNNLandmarkGraph method`), 18
- `set_params()` (`graphtools.graphs.MNNLandmarkPyGSPGraph method`), 30
- `set_params()` (`graphtools.graphs.MNNPyGSPGraph method`), 42
- `set_params()` (`graphtools.graphs.TraditionalGraph method`), 47
- `set_params()` (`graphtools.graphs.TraditionalLandmarkGraph method`), 51
- `set_params()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph method`), 63
- `set_params()` (`graphtools.graphs.TraditionalPyGSPGraph method`), 76
- `set_submatrix()` (in module `graphtools.utils`), 130
- `shortest_path()` (`graphtools.base.BaseGraph method`), 113
- `shortest_path()` (`graphtools.base.DataGraph method`), 119
- `shortest_path()` (`graphtools.graphs.kNNGraph method`), 81
- `shortest_path()` (`graphtools.graphs.kNNLandmarkGraph method`), 85
- `shortest_path()` (`graphtools.graphs.kNNLandmarkPyGSPGraph method`), 97
- `shortest_path()` (`graphtools.graphs.kNNPyGSPGraph method`), 110
- `shortest_path()` (`graphtools.graphs.LandmarkGraph method`), 10
- `shortest_path()` (`graphtools.graphs.MNNGraph method`), 14
- `shortest_path()` (`graphtools.graphs.MNNLandmarkGraph method`), 18
- `shortest_path()` (`graphtools.graphs.MNNLandmarkPyGSPGraph method`), 30
- `shortest_path()` (`graphtools.graphs.MNNPyGSPGraph method`), 43
- `shortest_path()` (`graphtools.graphs.TraditionalGraph method`), 47
- `shortest_path()` (`graphtools.graphs.TraditionalLandmarkGraph method`), 51
- `shortest_path()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph method`), 64
- `shortest_path()` (`graphtools.graphs.TraditionalPyGSPGraph method`), 76
- `sparse_maximum()` (in module `graphtools.utils`), 130
- `sparse_minimum()` (in module `graphtools.utils`), 130
- `sparse_nonzero_discrete()` (in module `graphtools.utils`), 130
- `sparse_set_diagonal()` (in module `graphtools.utils`), 130
- `subgraph()` (`graphtools.base.PyGSPGraph method`), 128
- `subgraph()` (`graphtools.graphs.kNNLandmarkPyGSPGraph method`), 98
- `subgraph()` (`graphtools.graphs.kNNPyGSPGraph method`), 110
- `subgraph()` (`graphtools.graphs.MNNLandmarkPyGSPGraph method`), 31
- `subgraph()` (`graphtools.graphs.MNNPyGSPGraph method`), 43
- `subgraph()` (`graphtools.graphs.TraditionalLandmarkPyGSPGraph method`), 64
- `subgraph()` (`graphtools.graphs.TraditionalPyGSPGraph method`), 77
- `subgraphs` (`graphtools.graphs.MNNGraph attribute`), 12
- `symmetrize_kernel()` (`graphtools.base.BaseGraph method`), 114
- `symmetrize_kernel()` (`graphtools.base.DataGraph method`), 119
- `symmetrize_kernel()` (`graphtools.graphs.kNNGraph method`), 81
- `symmetrize_kernel()` (`graphtools.graphs.kNNLandmarkGraph method`), 85
- `symmetrize_kernel()` (`graphtools.graphs.kNNLandmarkPyGSPGraph method`), 98
- `symmetrize_kernel()` (`graphtools.graphs.kNNPyGSPGraph method`), 111
- `symmetrize_kernel()` (`graphtools.graphs.LandmarkGraph method`), 11
- `symmetrize_kernel()` (`graphtools.graphs.MNNGraph method`), 15

`symmetrize_kernel()` (*graphtools.graphs.MNNLandmarkGraph method*), 18  
`symmetrize_kernel()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 31  
`symmetrize_kernel()` (*graphtools.graphs.MNNPyGSPGraph method*), 43  
`symmetrize_kernel()` (*graphtools.graphs.TraditionalGraph method*), 48  
`symmetrize_kernel()` (*graphtools.graphs.TraditionalLandmarkGraph method*), 52  
`symmetrize_kernel()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 64  
`symmetrize_kernel()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 77

**T**

`to_array()` (*in module graphtools.utils*), 130  
`to_igraph()` (*graphtools.base.BaseGraph method*), 114  
`to_igraph()` (*graphtools.base.DataGraph method*), 119  
`to_igraph()` (*graphtools.graphs.kNNGraph method*), 81  
`to_igraph()` (*graphtools.graphs.kNNLandmarkGraph method*), 85  
`to_igraph()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 98  
`to_igraph()` (*graphtools.graphs.kNNPyGSPGraph method*), 111  
`to_igraph()` (*graphtools.graphs.LandmarkGraph method*), 11  
`to_igraph()` (*graphtools.graphs.MNNGraph method*), 15  
`to_igraph()` (*graphtools.graphs.MNNLandmarkGraph method*), 18  
`to_igraph()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 31  
`to_igraph()` (*graphtools.graphs.MNNPyGSPGraph method*), 43  
`to_igraph()` (*graphtools.graphs.TraditionalGraph method*), 48  
`to_igraph()` (*graphtools.graphs.TraditionalLandmarkGraph method*), 52  
`to_igraph()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 64  
`to_igraph()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 77  
`to_pygsp()` (*graphtools.base.BaseGraph method*), 114  
`to_pygsp()` (*graphtools.base.DataGraph method*), 119  
`to_pygsp()` (*graphtools.graphs.kNNGraph method*), 81  
`to_pygsp()` (*graphtools.graphs.kNNLandmarkGraph method*), 86  
`to_pygsp()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 98  
`to_igraph()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 64  
`to_igraph()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 77  
`to_pickle()` (*graphtools.base.BaseGraph method*), 114  
`to_pickle()` (*graphtools.base.DataGraph method*), 119  
`to_pickle()` (*graphtools.graphs.kNNGraph method*), 81  
`to_pickle()` (*graphtools.graphs.kNNLandmarkGraph method*), 85  
`to_pickle()` (*graphtools.graphs.kNNLandmarkPyGSPGraph method*), 98  
`to_pickle()` (*graphtools.graphs.kNNPyGSPGraph method*), 111  
`to_pickle()` (*graphtools.graphs.LandmarkGraph method*), 11  
`to_pickle()` (*graphtools.graphs.MNNGraph method*), 15  
`to_pickle()` (*graphtools.graphs.MNNLandmarkGraph method*), 19  
`to_pickle()` (*graphtools.graphs.MNNLandmarkPyGSPGraph method*), 31  
`to_pickle()` (*graphtools.graphs.MNNPyGSPGraph method*), 43  
`to_pickle()` (*graphtools.graphs.TraditionalGraph method*), 48  
`to_pickle()` (*graphtools.graphs.TraditionalLandmarkGraph method*), 52  
`to_pickle()` (*graphtools.graphs.TraditionalLandmarkPyGSPGraph method*), 65  
`to_pickle()` (*graphtools.graphs.TraditionalPyGSPGraph method*), 77

[to\\_pygsp\(\)](#) ([graphtools.graphs.kNNPyGSPGraph](#) method), 111  
[to\\_pygsp\(\)](#) ([graphtools.graphs.LandmarkGraph](#) method), 11  
[to\\_pygsp\(\)](#) ([graphtools.graphs.MNNGraph](#) method), 15  
[to\\_pygsp\(\)](#) ([graphtools.graphs.MNNLandmarkGraph](#) method), 19  
[to\\_pygsp\(\)](#) ([graphtools.graphs.MNNLandmarkPyGSPGraph](#) method), 31  
[to\\_pygsp\(\)](#) ([graphtools.graphs.MNNPyGSPGraph](#) method), 43  
[to\\_pygsp\(\)](#) ([graphtools.graphs.TraditionalGraph](#) method), 48  
[to\\_pygsp\(\)](#) ([graphtools.graphs.TraditionalLandmarkGraph](#) method), 52  
[to\\_pygsp\(\)](#) ([graphtools.graphs.TraditionalLandmarkPyGSPGraph](#) method), 65  
[to\\_pygsp\(\)](#) ([graphtools.graphs.TraditionalPyGSPGraph](#) method), 77  
[TraditionalGraph](#) (class in [graphtools.graphs](#)), 44  
[TraditionalLandmarkGraph](#) (class in [graphtools.graphs](#)), 48  
[TraditionalLandmarkPyGSPGraph](#) (class in [graphtools.graphs](#)), 53  
[TraditionalPyGSPGraph](#) (class in [graphtools.graphs](#)), 65  
[transform\(\)](#) ([graphtools.base.Data](#) method), 116  
[transform\(\)](#) ([graphtools.base.DataGraph](#) method), 119  
[transform\(\)](#) ([graphtools.graphs.kNNGraph](#) method), 82  
[transform\(\)](#) ([graphtools.graphs.kNNLandmarkGraph](#) method), 86  
[transform\(\)](#) ([graphtools.graphs.kNNLandmarkPyGSPGraph](#) method), 99  
[transform\(\)](#) ([graphtools.graphs.kNNPyGSPGraph](#) method), 111  
[transform\(\)](#) ([graphtools.graphs.LandmarkGraph](#) method), 11  
[transform\(\)](#) ([graphtools.graphs.MNNGraph](#) method), 15  
[transform\(\)](#) ([graphtools.graphs.MNNLandmarkGraph](#) method), 19  
[transform\(\)](#) ([graphtools.graphs.MNNLandmarkPyGSPGraph](#) method), 31  
[transform\(\)](#) ([graphtools.graphs.MNNPyGSPGraph](#) method), 44  
[transform\(\)](#) ([graphtools.graphs.TraditionalGraph](#) method), 48  
[transform\(\)](#) ([graphtools.graphs.TraditionalLandmarkGraph](#) method), 52  
[transform\(\)](#) ([graphtools.graphs.TraditionalLandmarkPyGSPGraph](#) method), 65  
[transform\(\)](#) ([graphtools.graphs.TraditionalPyGSPGraph](#) method), 77  
[transitions](#) ([graphtools.graphs.kNNLandmarkGraph](#) attribute), 86  
[transitions](#) ([graphtools.graphs.kNNLandmarkPyGSPGraph](#) attribute), 99  
[transitions](#) ([graphtools.graphs.LandmarkGraph](#) attribute), 7, 11  
[transitions](#) ([graphtools.graphs.MNNLandmarkGraph](#) attribute), 19  
[transitions](#) ([graphtools.graphs.MNNLandmarkPyGSPGraph](#) attribute), 32  
[transitions](#) ([graphtools.graphs.TraditionalLandmarkGraph](#) attribute), 52  
[transitions](#) ([graphtools.graphs.TraditionalLandmarkPyGSPGraph](#) attribute), 65  
[translate\(\)](#) ([graphtools.base.PyGSPGraph](#) method), 128  
[translate\(\)](#) ([graphtools.graphs.kNNLandmarkPyGSPGraph](#) method), 99  
[translate\(\)](#) ([graphtools.graphs.kNNPyGSPGraph](#) method), 111  
[translate\(\)](#) ([graphtools.graphs.MNNLandmarkPyGSPGraph](#) method), 32  
[translate\(\)](#) ([graphtools.graphs.MNNPyGSPGraph](#) method), 44  
[translate\(\)](#) ([graphtools.graphs.TraditionalLandmarkPyGSPGraph](#) method), 65  
[translate\(\)](#) ([graphtools.graphs.TraditionalPyGSPGraph](#) method), 78  
**U**  
[U](#) ([graphtools.base.PyGSPGraph](#) attribute), 120  
[U](#) ([graphtools.graphs.kNNLandmarkPyGSPGraph](#) attribute), 87  
[U](#) ([graphtools.graphs.kNNPyGSPGraph](#) attribute), 100

- U (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 20
- U (*graphtools.graphs.MNNPyGSPGraph* attribute), 32
- U (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 53
- U (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 66

## W

- weighted (*graphtools.base.BaseGraph* attribute), 114
- weighted (*graphtools.base.DataGraph* attribute), 120
- weighted (*graphtools.graphs.kNNGraph* attribute), 82
- weighted (*graphtools.graphs.kNNLandmarkGraph* attribute), 86
- weighted (*graphtools.graphs.kNNLandmarkPyGSPGraph* attribute), 99
- weighted (*graphtools.graphs.kNNPyGSPGraph* attribute), 112
- weighted (*graphtools.graphs.LandmarkGraph* attribute), 12
- weighted (*graphtools.graphs.MNNGraph* attribute), 15
- weighted (*graphtools.graphs.MNNLandmarkGraph* attribute), 19
- weighted (*graphtools.graphs.MNNLandmarkPyGSPGraph* attribute), 32
- weighted (*graphtools.graphs.MNNPyGSPGraph* attribute), 44
- weighted (*graphtools.graphs.TraditionalGraph* attribute), 48
- weighted (*graphtools.graphs.TraditionalLandmarkGraph* attribute), 53
- weighted (*graphtools.graphs.TraditionalLandmarkPyGSPGraph* attribute), 65
- weighted (*graphtools.graphs.TraditionalPyGSPGraph* attribute), 78