pygsp Documentation

Release 0.3.0

EPFL LTS2

Aug 11, 2017

Contents

CHAPTER 1

About

PyGSP is a Graph Signal Processing Toolbox implemented in Python. It is a port of the Matlab GSP toolbox.

- Development : https://github.com/epfl-lts2/pygsp
- GSP matlab toolbox : https://github.com/epfl-lts2/gspbox

Features

This toolbox facilitate graph constructions and give tools to perform signal processing on them.

A whole list of preconstructed graphs can be used as well as core functions to create any other graph among which:

```
- Neighest Neighbor Graphs
  - Bunny
  - Cube
  - Sphere
  - TwoMoons
- Airfoil
- Comet
- Community
- DavidSensorNet
- ErdosRenyi
- FullConnected
- Grid2d
- Logo GSP
- LowStretchTree
- Minnesota
- Path
- RandomRegular
- RandomRing
- Ring
- Sensor
- StochasticBlockModel
```

```
- Swiss roll
- Torus
```

On these graphs, filters can be applied to do signal processing. To this end, there is also a list of predefined filters on this toolbox:

```
- Abspline
- Expwin
- Gabor
- HalfCosine
- Heat
- Held
- Itersine
- MexicanHat
- Meyer
- Papadakis
- Regular
- Simoncelli
- SimpleTf
```

Installation

Ubuntu

The PyGSP module is available on PyPI, the Python Package Index. If you don't have pip, install it.:

```
$ sudo apt-get install python-pip
```

Ideally, you should be able to install the PyGSP on your computer by simply entering the following command:

```
$ pip install pygsp
```

This installation requires numpy and scipy. If you don't have them installed already, pip installing pygsp will try to install them for you. Note that these two mathematical libraries requires additional system packages.

For a classic UNIX system, you will need python-dev(el) (or equivalent) installed as a system package as well as the fortran extension for your favorite compiler (gfortran for gcc). You will also need the blas/lapack implementation for your system. If you can't install numpy or scipy, try installing the following and then install numpy and scipy:

```
$ sudo apt-get install python-dev liblapack-dev libatlas-dev gcc gfortran
```

Then, try again to install the pygsp:

```
$ pip install pygsp
```

Plotting

If you want to use the plotting functionalities of the PyGSP, you have to install matplotlib or pygtgraph. For matplotlib, just do:

```
$ sudo apt-get python-matplotlib
```

Another way is to manually download from PyPI, unpack the package and install with:

```
$ python setup.py install
```

Instructions and requirements to install pyqtgraph can be found at http://www.pyqtgraph.org/.

Testing

Execute the project test suite once to make sure you have a working install:

```
$ python setup.py test
```

Authors

- Basile Châtillon <basile.chatillon@epfl.ch>,
- Alexandre Lafaye <alexandre.lafaye@epfl.ch>,
- Lionel Martin <lionel.martin@epfl.ch>,
- Nicolas Rod <nicolas.rod@epfl.ch>

Acknowledgment

This project has been partly funded by the Swiss National Science Foundation under grant 200021_154350 “Towards Signal Processing on Graphs”.

CHAPTER 2

Tutorials

The following are some tutorials which show and explain how to use the toolbox to solve some real problems.

Simple problem

This example demonstrates how to create a graph, a filter and analyse a signal on the graph.

```
>>> import pygsp
>>> G = pygsp.graphs.Logo()
>>> f = pygsp.filters.Heat(G)
>>> Sl = f.analysis(G.L.todense(), method='cheby')
```

GSP Demo

This tutorial shows basic operations of the toolbox. To start open a python shell (IPython is recommended here) and import the pygsp. You would probably also import numpy as you will need it to create matrices and arrays.

```
>>> import pygsp
>>> import numpy as np
```

The first step is to create a graph, there's a general class that can be used to generate graph from it's weight matrix.

```
>>> np.random.seed(42) # We will use a seed to make reproducible results
>>> W = np.random.rand(400, 400)
>>> G = pygsp.graphs.Graph(W)
```

You have now a graph structure ready to be used everywhere in the box! If you want to know more about the Graph class and it's subclasses you can check the online doc at : https://lts2.epfl.ch/pygsp/ You can also check the included methods for all graphs with the usual help function.

For the next steps of the demo, we will be using the logo graph bundled with the toolbox :

```
>>> G = pygsp.graphs.Logo()
```

You can now plot the graph:

```
>>> G.plot(default_qtg=False, savefig=True, plot_name='doc/tutorials/img/logo')
```

Looks good isn't it? Now we can start to analyse the graph. The next step to compute Graph Fourier Transform or exact graph filtering is to precompute the Fourier basis of the graph. This operation can be very long as it needs to to fully diagonalize the Laplacian. Happily it is not needed to filter signal on graphs.

```
>>> G.compute_fourier_basis()
```

You can now access the eigenvalues of the fourier basis with G.e and the eigenvectors G.U, they look like sinuses on the graph. Let's plot the second and third eigenvector, as the one is only constant.

```
>>> pygsp.plotting.plt_plot_signal(G, G.U[:, 2], savefig=True, vertex_size=50, plot_
↪name='doc/tutorials/img/logo_second_eigenvector')
>>> pygsp.plotting.plt_plot_signal(G, G.U[:, 3], savefig=True, vertex_size=50, plot_
↪name='doc/tutorials/img/logo_third_eigenvector')
```

Fig. 2.1: Second eigenvector

Fig. 2.2: Third eigenvector

Let's discover basic filters operations, filters are usually defined in the spectral domain.

First let's define a filter object:

```
>>> F = pygsp.filters.Filter(G)
```

And we can assign this function

$$g(x) = \frac{1}{1 + \tau x}$$

to it:

```
>>> tau = 1
>>> g = lambda x: 1./(1. + tau * x)
>>> F.g = [g]
```

You can also put multiple functions in a list to define a filterbank!

```
>>> F.plot(plot_eigenvalues=True, savefig=True, plot_name='doc/tutorials/img/low_pass_
↪filter')
```

Here's our low pass filter.

To accompain our new filter, let's create a nice signal on the logo by setting each letter to a certain value and then adding some random noise.

```
>>> f = np.zeros((G.N,))
>>> f[G.info['idx_g']-1] = - 1
>>> f[G.info['idx_s']-1] = 1
>>> f[G.info['idx_p']-1] = -0.5
>>> f += np.random.rand(G.N,)
```

The filter is plotted all along the spectrum of the graph, the cross at the bottom are the laplacian's eigenvalues. Those are the point where the continuous filter will be evaluated to create a discrete filter. To apply it to a given signal, you only need to run:

```
>>> f2 = F.analysis(f)
```

Finally here's the noisy signal and the denoised version right under.

```
>>> pygsp.plotting.plt_plot_signal(G, f, savefig=True, vertex_size=50, plot_name='doc/
↪tutorials/img/noisy_logo')
>>> pygsp.plotting.plt_plot_signal(G, f2, savefig=True, vertex_size=50, plot_name=
↪'doc/tutorials/img/denoised_logo')
```

So here are the basics for the PyGSP toolbox, if you want more informations you can check the doc in *the reference guide section*.

Enjoy the toolbox!

GSP Wavelet Demo

- Introduction to spectral graph wavelet with the PyGSP

Description

The wavelets are a special type of filterbank, in this demo we will show you how you can very easily construct a wavelet frame and apply it to a signal.

In this demo we will show you how to compute the wavelet coefficients of a graph and visualize them. First let's import the toolbox, numpy and load a graph.

```
>>> import pygsp
>>> import numpy as np
>>> G = pygsp.graphs.Bunny()
```

This graph is a nearest-neighbor graph of a pointcloud of the Stanford bunny. It will allow us to get interesting visual results using wavelets.

At this stage we could compute the full Fourier basis using

```
>>> G.compute_fourier_basis()
```

but this would take a lot of time, and can be avoided by using Chebychev polynomials approximations.

Simple filtering

Before tackling wavelets, we can see the effect of one filter localized on the graph. So we can first design a few heat kernel filters

```
>>> taus = [1, 10, 25, 50]
>>> Hk = pygsp.filters.Heat(G, taus, normalize=False)
```

Let's now create a signal as a Kronecker located on one vertex (e.g. the vertex 83)

```
>>> S = np.zeros(G.N)
>>> vertex_delta = 83
>>> S[vertex_delta] = 1
>>> Sf_vec = Hk.analysis(S)
>>> Sf = Sf_vec.reshape((Sf_vec.size/len(taus), len(taus)), order='F')
```

Let's plot the signal:

```
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,0], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/heat_tau_1')
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,1], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/heat_tau_10')
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,2], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/heat_tau_25')
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,3], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/heat_tau_50')
```

Fig. 2.3: Heat tau = 1

Fig. 2.4: Heat tau = 10

Fig. 2.5: Heat tau = 25

Visualizing wavelets atoms

Let's now replace the Heat filter by a filter bank of wavelets. We can create a filter bank using one of the predefined filters such as pygsp.filters.MexicanHat.

```
>>> Nf = 6
>>> Wk = pygsp.filters.MexicanHat(G, Nf)
```

We can now plot the filter bank spectrum :

```
>>> Wk.plot(savefig=True, plot_name='doc/tutorials/img/mexican_hat')
```

As we can see, the wavelets atoms are stacked on the low frequency part of the spectrum. If we want to get a better coverage of the graph spectrum, we could have used the WarpedTranslates filter bank.

```
>>> S_vec = Wk.analysis(S)
>>> S = S_vec.reshape((S_vec.size/Nf, Nf), order='F')
>>> pygsp.plotting.plt_plot_signal(G, S[:, 0], savefig=True, plot_name='doc/tutorials/
↪img/wavelet_filtering')
```

Fig. 2.6: Heat tau = 50

Fig. 2.7: Mexican Hat Wavelet filter

We can visualize the filtering by one atom the same way the did for the Heat kernel, by placing a Kronecker delta at one specific vertex.

```
>>> S = np.zeros((G.N * Nf, Nf))
>>> S[vertex_delta] = 1
>>> for i in range(Nf):
...     S[vertex_delta + i * G.N, i] = 1
>>> Sf = Wk.synthesis(S)
```

```
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,0], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/wavelet_1')
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,1], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/wavelet_2')
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,2], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/wavelet_3')
>>> pygsp.plotting.plt_plot_signal(G, Sf[:,3], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/wavelet_4')
```

```
>>> G = pygsp.graphs.Bunny()
>>> Wk = pygsp.filters.MexicanHat(G, Nf)
>>> s_map = G.coords
```

```
>>> s_map_out = Wk.analysis(s_map)
>>> s_map_out = np.reshape(s_map_out, (G.N, Nf, 3))
```

```
>>> d = s_map_out[:, :, 0]**2 + s_map_out[:, :, 1]**2 + s_map_out[:, :, 2]**2
>>> d = np.sqrt(d)
```

```
>>> pygsp.plotting.plt_plot_signal(G, d[:, 1], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/curv_scale_1')
>>> pygsp.plotting.plt_plot_signal(G, d[:, 2], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/curv_scale_2')
>>> pygsp.plotting.plt_plot_signal(G, d[:, 3], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/curv_scale_3')
>>> pygsp.plotting.plt_plot_signal(G, d[:, 4], vertex_size=20, savefig=True, plot_
↪name='doc/tutorials/img/curv_scale_4')
```

GSP Graph TV Demo - Reconstruction of missing sample on a graph using TV

Description

Reconstruction of missing sample on a graph using TV

In this demo, we try to reconstruct missing sample of a piece-wise smooth signal on a graph. To do so, we will minimize the well-known TV norm defined on the graph.

For this example, you need the pyunlocbox. You can download it from https://github.com/epfl-lts2/pyunlocbox and installing it.

We express the recovery problem as a convex optimization problem of the following form:

$$arg\min_x \|\nabla(x)\|_1 \text{ s. t. } \|Mx - b\|_2 \leq \epsilon$$

Where b represents the known measurements, M is an operator representing the mask and ϵ is the radius of the l2 ball.

We set:

- $f_1(x) = ||\nabla x||_1$

We define the prox of f_1 as:

$$prox_{f1,\gamma}(z) = arg\min_x \frac{1}{2}\|x - z\|_2^2 + \gamma\|\nabla z\|_1$$

- f_2 is the indicator function of the set S define by :math:||Mx-b||_2 < epsilon

We define the prox of f_2 as

$$prox_{f2,\gamma}(z) = arg\min_x \frac{1}{2}\|x - z\|_2^2 + i_S(x)$$

with $i_S(x)$ is zero if x is in the set S and infinity otherwise. This previous problem has an identical solution as:

$$arg\min_z \|x - z\|_2^2 \qquad such \quad that \qquad \|Mz - b\|_2 \leq \epsilon$$

It is simply a projection on the B2-ball.

Results and code

```
>>> from pygsp import graphs, plotting
>>> import numpy as np
>>>
>>> # Create a random sensor graph
>>> G = graphs.Sensor(N=256, distribute=True)
>>> G.compute_fourier_basis()
>>>
>>> # Create signal
>>> graph_value = np.copysign(np.ones(np.shape(G.U[:, 3])[0]), G.U[:, 3])
>>>
>>> plotting.plt_plot_signal(G, graph_value, savefig=True, plot_name='doc/tutorials/
↪img/original_signal')
```

This figure shows the original signal on graph.

```
>>> # Create the mask
>>> M = np.random.rand(G.U.shape[0], 1)
>>> M = M > 0.6  # Probability of having no label on a vertex.
>>>
```

```
>>> # Applying the mask to the data
>>> sigma = 0.0
>>> depleted_graph_value = M * (graph_value.reshape(graph_value.size, 1) + sigma * np.
↪random.standard_normal((G.N, 1)))
>>>
>>> plotting.plt_plot_signal(G, depleted_graph_value, show_edges=True, savefig=True,␣
↪plot_name='doc/tutorials/img/depleted_signal')
```

This figure shows the signal on graph after the application of the mask and addition of noise. More than half of the vertices are set to 0.

This figure shows the reconstructed signal thanks to the algorithm.

Comparison with Tikhonov regularization

We can also use the Tikhonov regularizer that will promote smoothness. In this case, we solve:

$$arg\min_x \tau \|\nabla(x)\|_2^2 \text{ s. t. } \|Mx - b\|_2 \leq \epsilon$$

The result is presented as following:

This figure shows the reconstructed signal thanks to the algorithm.

CHAPTER 3

Reference guide

Toolbox overview

This toolbox is splitted in different modules taking care of the different aspects of Graph Signal Processing.

Those modules are : Graphs, Filters, Operators and PointClouds.

You can find detailed documentation on the use of the functions in the subsequent pages.

Graphs

Graphs objects

Main Graph Class

class `pygsp.graphs.`**`Graph`**(*W*, *gtype='unknown'*, *lap_type='combinatorial'*, *coords=None*, *plotting={}*, ***kwargs*)

Bases: `object`

The main graph object.

It is used to initialize by default every missing field of the subclass graphs. It can also be used alone to initialize customs graphs.

Parameters **W** : sparse matrix or ndarray (data is float)

Weight matrix. Mandatory.

gtype : string

Graph type (default is “unknown”)

lap_type : string

Laplacian type (default is ‘combinatorial’)

coords : ndarray

Coordinates of the vertices (default is None)

plotting : dict

Dictionnary containing the plotting parameters

Examples

```
>>> from pygsp import graphs
>>> import numpy as np
>>> W = np.arange(4).reshape(2, 2)
>>> G = graphs.Graph(W)
```

compute_fourier_basis(*smallest_first=True*, *force_recompute=False*, ***kwargs*)

Compute the fourier basis of the graph.

Parameters **smallest_first: bool**

Define the order of the eigenvalues. Default is smallest first (True).

force_recompute: bool

Force to recompute the Fourier basis if already existing.

References

cite ´chung1997spectral´

copy_graph_attributes(*Gn*, *ctype=True*)

Copy some parameters of the graph into a given one.

Parameters **G** : Graph structure

ctype : bool

Flag to select what to copy (Default is True)

Gn : Graph structure

The graph where the parameters will be copied

Returns **Gn** : Partial graph structure

Examples

```
>>> from pygsp import graphs
>>> Torus = graphs.Torus()
>>> G = graphs.TwoMoons()
>>> G.copy_graph_attributes(ctype=False, Gn=Torus);
```

create_laplacian(*lap_type='combinatorial'*)

Create a new graph laplacian.

Parameters **lap_type** : string

The laplacian type to use. Default is “combinatorial”.

estimate_lmax(*force_recompute=False*)
Estimate the maximal eigenvalue.

Parameters **force_recompute** : boolean

Force to recompute the maximal eigenvalue. Default is false.

Examples

Just define a graph and apply the estimation on it.

```
>>> from pygsp import graphs
>>> import numpy as np
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> G.estimate_lmax()
```

is_connected()
Function to check the strong connectivity of the input graph.

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.

Returns **connected** : bool

A bool value telling if the graph is connected

Examples

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

is_directed()
Define if the graph has directed edges.

Notes

Can also be used to check if a matrix is symmetrical

Examples

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

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

See plotting doc.

set_coords (*kind='ring2D'*, *coords=None*)
Set coordinates for the vertices.

Parameters **kind** : string

The kind of display. Default is 'ring2D'. Accepting ['ring2D', 'community2D', 'manual', 'random2D', 'random3D'].

coords : np.ndarray

An array of coordinates in 2D or 3D. Used only if kind is manual. Set the coordinates to this array as is.

Examples

```
>>> from pygsp import graphs
>>> G = graphs.ErdosRenyi()
>>> G.set_coords()
>>> G.plot()
```

subgraph (*ind*)
Create a subgraph from G.

Parameters **G** : Graph

Original graph

ind : list

Nodes to keep

Returns **subG** : Graph

Subgraph

Examples

```
>>> from pygsp import graphs
>>> import numpy as np
>>> W = np.arange(16).reshape(4, 4)
>>> G = graphs.Graph(W)
>>> ind = [3]
>>> subG = graphs.Graph.subgraph(G, ind)
```

This function create a subgraph from G taking only the nodes in ind.

update_graph_attr (**args*, ***kwargs*)
Recompute some attribute of the graph.

Parameters **args: list of string**

the arguments that will be not changed and not re-compute.

kwargs: Dictionnary

The arguments with their new value.

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Ring(N=10)
>>> newW = G.W
>>> newW[1] = 1
>>> G.update_graph_attr('N', 'd', W=newW)
```

Updates all attributes of G excepted 'N' and 'd'

Grid2d

class pygsp.graphs.**Grid2d**(*Nv=16*, *Mv=None*, ***kwargs*)

Bases: pygsp.graphs.graph.Graph

Create a 2 dimensional grid graph.

Parameters **Nv** : int

Number of vertices along the first dimension (default is 16)

Mv : int

Number of vertices along the second dimension (default is Nv)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Grid2d(Nv=32)
```

Torus

class pygsp.graphs.**Torus**(*Nv=16*, *Mv=None*, ***kwargs*)

Bases: pygsp.graphs.graph.Graph

Create a Torus graph.

Parameters **Nv** : int

Number of vertices along the first dimension (default is 16)

Mv : int

Number of vertices along the second dimension (default is Nv)

References

See *[Str99]* for more informations.

Examples

```
>>> from pygsp import graphs
>>> Nv = 32
>>> G = graphs.Torus(Nv=Nv)
```

Comet

class pygsp.graphs.**Comet**(*Nv=32*, *k=12*, ***kwargs*)
Bases: pygsp.graphs.graph.Graph

Create a Comet graph.

Parameters **Nv** : int

Number of vertices along the first dimension (default is 16)

Mv : int

Number of vertices along the second dimension (default is Nv)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Comet() # (== graphs.Comet(Nv=32, k=12))
```

LowStretchTree

class pygsp.graphs.**LowStretchTree**(*k=6*, ***kwargs*)
Bases: pygsp.graphs.graph.Graph

Create a low stretch tree graph.

Parameters **k** : int

2^k points on each side of the grid of vertices (default 6)

Examples

```
>>> from pygsp import graphs, plotting
>>> G = graphs.LowStretchTree(k=3)
>>> G.plot()
```

RandomRegular

class pygsp.graphs.**RandomRegular**(*N=64*, *k=6*, ***kwargs*)
Bases: pygsp.graphs.graph.Graph

Create a random regular graphs

The random regular graph has the property that every nodes is connected to 'k' other nodes.

Parameters **N** : int

Number of nodes (default is 64)

k : int

Number of connections of each nodes (default is 6)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.RandomRegular()
```

createRandRegGraph (*vertNum*, *deg*, *maxIter=10*)
Create a simple d-regular undirected graph.

simple = without loops or double edges d-reglar = each vertex is adjecent to d edges

Parameters **vertNum** : int

Number of vertices

deg : int

The degree of each vertex

maxIter : int

The maximum number of iterations

Returns **A** : sparse

Representation of the graph

isRegularGraph (*A*)
Troubleshoot a given regular graph.

Parameters **A** : sparse matrix

Ring

class pygsp.graphs.**Ring** (*N=64*, *k=1*, ***kwargs*)
Bases: pygsp.graphs.graph.Graph

Create a ring graph.

Parameters **N** : int

Number of vertices (default is 64)

k : int

Number of neighbors in each directions (default is 1)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Ring()
```

Community

class pygsp.graphs.**Community** (*N=256*, ***kwargs*)
Bases: pygsp.graphs.graph.Graph

Create a community graph.

Parameters **N** : int

Number of nodes (default = 256)

kwargs : Dict

Optional parameters for the construction of the Community graph

Nc [int] Number of communities (default = $round(\sqrt{N}/2)$)

min_comm [int] Minimum size of the communities (default = round(N/Nc/3))

min_deg [int] Minimum degree of each node (default = 0, NOT IMPLEMENTED YET)

comm_sizes [int] Size of the communities (default = random)

size_ratio [float] Ratio between the radius of world and the radius of communities (default = 1)

world_density [float] Probability of a random edge between two different communities (default = 1/N)

comm_density [float] Probability of a random edge inside any community (default = None, not used if None)

k_neigh [int] Number of intra-community connections (default = None, not used if None or comm_density is defined)

epsilon [float] Max distance at which two nodes sharing a community are connected (default = $sqrt(2\sqrt{N})/2$, not used if k_neigh or comm_density is defined)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Community()
```

Minnesota

class pygsp.graphs.**Minnesota**(*connect=True*)

Bases: pygsp.graphs.graph.Graph

Create a community graph.

Parameters **connect** : bool

Change the graph to be connected. (default = True)

References

See *[Gle]*

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Minnesota()
```

Sensor

class pygsp.graphs.**Sensor**(*N=64*, *Nc=2*, *regular=False*, *n_try=50*, *distribute=False*, *connected=True*, ***kwargs*)

Bases: pygsp.graphs.graph.Graph

Create a random sensor graph.

Parameters **N** : int

Number of nodes (default = 64)

Nc : int

Minimum number of connections (default = 1)

regular : bool

Flag to fix the number of connections to nc (default = False)

n_try : int

Number of attempt to create the graph (default = 50)

distribute : bool

To distribute the points more evenly (default = False)

connected : bool

To force the graph to be connected (default = True)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Sensor(N=300)
```

create_weight_matrix(*N*, *param_distribute*, *param_regular*, *param_Nc*)

get_nc_connection(*W*, *param_nc*)

Airfoil

class pygsp.graphs.**Airfoil**(***kwargs*)

Bases: pygsp.graphs.graph.Graph

Create the airfoil graph.

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Airfoil()
```

DavidSensorNet

class pygsp.graphs.**DavidSensorNet**(*N=64*)
Bases: pygsp.graphs.graph.Graph

Create a sensor network.

Parameters **N** : int

Number of vertices (default = 64)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.DavidSensorNet(N=500)
```

FullConnected

class pygsp.graphs.**FullConnected**(*N=10*)
Bases: pygsp.graphs.graph.Graph

Create a fully connected graph.

Parameters **N** : int

Number of vertices (default = 10)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.FullConnected(N=5)
```

Logo

class pygsp.graphs.**Logo**(***kwargs*)
Bases: pygsp.graphs.graph.Graph

Create a graph with the GSP Logo.

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Logo()
```

Path

class pygsp.graphs.**Path**(*N=16*)
Bases: pygsp.graphs.graph.Graph

Create a path graph.

Parameters **N** : int

Number of vertices (default = 32)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Path(N=16)
```

RandomRing

class pygsp.graphs.**RandomRing**(*N=64*)

Bases: pygsp.graphs.graph.Graph

Create a ring graph.

Parameters **N** : int

Number of vertices (default = 64)

Examples

```
>>> from pygsp import graphs
>>> G = graphs.RandomRing(N=16)
```

SwissRoll

class pygsp.graphs.**SwissRoll**(*N=400*, *a=1*, *b=4*, *dim=3*, *thresh=1e-06*, *s=None*, *noise=False*, *srtype='uniform'*)

Bases: pygsp.graphs.graph.Graph

Create a swiss roll graph.

Parameters **N** : int

Number of vertices (default = 400)

a : int

(default = 1)

b : int

(default = 4)

dim : int

(default = 3)

thresh : float

(default = 1e-6)

s : float

sigma (default = sqrt(2./N))

noise : bool

Wether to add noise or not (default = False)

srtype : str

Swiss roll Type, possible arguments are 'uniform' or 'classic' (default = 'uniform')

Examples

```
>>> from pygsp import graphs
>>> G = graphs.SwissRoll()
```

rescale_center(*x*)

Rescaling the dataset.

Rescaling the dataset, previously and mainly used in the SwissRoll graph.

Parameters **x** : ndarray

Dataset to be rescaled.

Returns **r** : ndarray

Rescaled dataset.

Examples

```
>>> from pygsp import utils
>>> utils.dummy(0, [1, 2, 3], True)
array([1, 2, 3])
```

Check Connectivity

Check Weights

pygsp.graphs.**check_weights**(*W*)

Check the characteristics of the weights matrix.

Parameters **W** : weights matrix

The weights matrix to check

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 zero value else false

Examples

```
>>> from scipy import sparse
>>> from pygsp.graphs import gutils
>>> np.random.seed(42)
>>> W = sparse.rand(10,10,0.2)
>>> weights_chara = gutils.check_weights(W)
```

Compute Fourier Basis

Create Laplacian

Estimate Lmax

Is directed

Symetrize

NNGraphs objects

NNGraphs Class

class pygsp.graphs.**NNGraph**(*Xin*, *NNtype='knn'*, *use_flann=False*, *center=True*, *rescale=True*, *k=10*, *sigma=0.1*, *epsilon=0.01*, *gtype=None*, *plotting={}*, *symmetrize_type='average'*, ***kwargs*)

Bases: pygsp.graphs.graph.Graph

Creates a graph from a pointcloud.

Parameters **Xin** : ndarray

Input points

use_flann : bool

Whether flann method should be used (knn is otherwise used). (default is False) (this option is not implemented yet)

center : bool

Center the data (default is True)

rescale : bool

Rescale the data (in a 1-ball) (default is True)

k : int

Number of neighbors for knn (default is 10)

sigma : float

Variance of the distance kernel (default is 0.1)

epsilon : float

RRdius for the range search (default is 0.01)

gtype : string

The type of graph (default is "knn")

Examples

```
>>> from pygsp import graphs
>>> import numpy as np
>>> Xin = np.arange(90).reshape(30, 3)
>>> G = graphs.NNGraph(Xin)
```

Bunny

class pygsp.graphs.**Bunny**(***kwargs*)

Bases: pygsp.graphs.nngraphs.nngraph.NNGraph

Create a graph of the stanford bunny.

References

[TL94]

Examples

```
>>> from pygsp import graphs
>>> G = graphs.Bunny()
```

Cube

class pygsp.graphs.**Cube**(*radius=1*, *nb_pts=300*, *nb_dim=3*, *sampling='random'*, ***kwargs*)

Bases: pygsp.graphs.nngraphs.nngraph.NNGraph

Creates the graph of an hyper-cube.

Parameters **radius** : float

Edge lenght (default = 1)

nb_pts : int

Number of vertices (default = 300)

nb_dim : int

Dimension (default = 3)

sampling : string

Variance of the distance kernel (default = 'random') (Can now only be 'random')

Examples

```
>>> from pygsp import graphs
>>> radius = 5
>>> G = graphs.Cube(radius=radius)
```

Sphere

class pygsp.graphs.**Sphere**(*radius=1*, *nb_pts=300*, *nb_dim=3*, *sampling='random'*, ***kwargs*)
Bases: pygsp.graphs.nngraphs.nngraph.NNGraph

Creates a spherical-shaped graph.

Parameters **radius** : flaot

Radius of the sphere (default = 1)

nb_pts : int

Number of vertices (default = 300)

nb_dim : int

Dimension (default = 3)

sampling : sting

Variance of the distance kernel (default = 'random') (Can now only be 'random')

Examples

```
>>> from pygsp import graphs
>>> radius = 5
>>> G = graphs.Sphere(radius=radius)
```

TwoMoons

class pygsp.graphs.**TwoMoons**(*moontype='standard'*, *sigmag=0.05*, *N=400*, *sigmad=0.07*, *d=0.5*)
Bases: pygsp.graphs.nngraphs.nngraph.NNGraph

Create a 2 dimensional graph of the Two Moons.

Parameters **moontype** : string

You have the freedom to chose if you want to create a standard two_moons graph or a synthetised one (default is 'standard'). 'standard' : Create a two_moons graph from a based graph. 'synthetised' : Create a synthetised two_moon

sigmag : float

Variance of the distance kernel (default = 0.05)

N : int

Number of vertices (default = 2000)

sigmad : float

Variance of the data (do not set it too high or you won't see anything) (default = 0.05)

d : float

Distance of the two moons (default = 0.5)

Examples

```
>>> from pygsp import graphs
>>> G1 = graphs.TwoMoons(moontype='standard')
>>> G2 = graphs.TwoMoons(moontype='synthetised', N=1000, sigmad=0.1, d=1)
```

create_arc_moon(*N*, *sigmad*, *d*, *number*)

This module implements graphs and contains predefined graphs for the most famous ones.

A graph is constructed either from its adjacency matrix, its weight matrix or any other parameter which depends on the particular graph you are trying to build. For specific information, *see details here*.

Filters

Filters Objects

Main Filters Class

class pygsp.filters.**Filter**(*G*, *filters=None*, ***kwargs*)

Bases: object

Parent class for all Filters or Filterbanks, contains the shared methods for those classes.

analysis(*s*, *method=None*, *cheb_order=30*, *lanczos_order=30*, ***kwargs*)

Operator to analyse a filterbank

Parameters **s** : ndarray

graph signals to analyse

method : string

wether using an exact method, cheby approx or lanczos

cheb_order : int

Order for chebyshev

Returns **c** : ndarray

Transform coefficients

Examples

```
>>> import numpy as np
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> MH = filters.MexicanHat(G)
>>> x = np.arange(G.N**2).reshape(G.N, G.N)
>>> co = MH.analysis(x)
```

[HVG11]

approx(*m*, *N*, ***kwargs*)

can_dual()

Creates a dual graph form a given graph

evaluate(*f*, **args*, ***kwargs*)

filterbank_bounds(*N=999*, *bounds=None*)
Compute approximate frame bounds for a filterbank.

Parameters **bounds** : interval to compute the bound.

Given in an ndarray: np.array([xmin, xnmax]). By default, bounds is None and filtering is bounded by the eigenvalues of G.

N : Number of point for the line search

Default is 999

Returns **lower** : Filterbank lower bound

upper : Filterbank upper bound

filterbank_matrix()
Create the matrix of the filterbank frame.

This function creates the matrix associated to the filterbank g. The size of the matrix is MN x N, where M is the number of filters.

Returns **F** : Frame

inverse(*c*, ***kwargs*)

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

See plotting doc.

synthesis(*c*, *order=30*, *method=None*, ***kwargs*)
Synthesis operator of a filterbank

Parameters **G** : Graph structure.

c : Transform coefficients

method : Select the method ot be used for the computation.

- 'exact' : Exact method using the graph Fourier matrix
- 'cheby' : Chebyshev polynomial approximation
- 'lanczos' : Lanczos approximation

Default : if the Fourier matrix is present: 'exact' otherwise 'cheby'

order : Degree of the Chebyshev approximation

Default is 30

Returns **signal** : sythesis signal

tighten()

wlog_scales(*lmin*, *lmax*, *Nscales*, *t1=1*, *t2=2*)
Compute logarithm scales for wavelets

Parameters **lmin** : int

Minimum non-zero eigenvalue

lmax : int

Maximum eigenvalue

Nscales : int

Number of scales

Returns **s** : ndarray

Scale

Abspline

class pygsp.filters.**Abspline**(*G*, *Nf=6*, *lpfactor=20*, *t=None*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Abspline Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

Nf : int

Number of filters from 0 to lmax (default = 6)

lpfactor : int

Low-pass factor lmin=lmax/lpfactor will be used to determine scales, the scaling function will be created to fill the lowpass gap. (default = 20)

t : ndarray

Vector of scale to be used (Initialized by default at the value of the log scale)

Returns **out** : Abspline

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Abspline(G)
```

Expwin

class pygsp.filters.**Expwin**(*G*, *bmax=0.2*, *a=1.0*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Expwin Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

bmax : float

Maximum relative band (default = 0.2)

a : int

Slope parameter (default = 1)

Returns **out** : Expwin

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Expwin(G)
```

HalfCosine

class pygsp.filters.**HalfCosine**(*G*, *Nf=6*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

HalfCosine Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

Nf : int

Number of filters from 0 to lmax (default = 6)

Returns

——-

out : HalfCosine

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.HalfCosine(G)
```

Itersine

class pygsp.filters.**Itersine**(*G*, *Nf=6*, *overlap=2.0*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Create a itersine filterbanks

This function create a itersine half overlap filterbank of Nf filters Going from 0 to lambda_max

Parameters **G** : Graph

Nf : int

Number of filters from 0 to lmax. (default = 6)

overlap : int

(default = 2)

Returns **out** : Itersine

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Itersine(G)
```

MexicanHat

class pygsp.filters.**MexicanHat**(*G*, *Nf=6*, *lpfactor=20*, *t=None*, *normalize=False*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Mexican hat Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

Nf : int

Number of filters from 0 to lmax (default = 6)

lpfactor : int

Low-pass factor lmin=lmax/lpfactor will be used to determine scales, the scaling function will be created to fill the lowpass gap. (default = 20)

t : ndarray

Vector of scale to be used (Initialized by default at the value of the log scale)

normalize : bool

Wether to normalize the wavelet by the factor/sqrt(t). (default = False)

Returns **out** : MexicanHat

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.MexicanHat(G)
```

Meyer

class pygsp.filters.**Meyer**(*G*, *Nf=6*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Meyer Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

Nf : int

Number of filters from 0 to lmax (default = 6)

Returns **out** : Meyer

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Meyer(G)
```

SimpleTf

class pygsp.filters.**SimpleTf**(*G*, *Nf=6*, *t=None*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

SimpleTf Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

Nf : int

Number of filters from 0 to lmax (default = 6)

t : ndarray

Vector of scale to be used (Initialized by default at the value of the log scale)

Returns **out** : SimpleTf

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.SimpleTf(G)
```

WarpedTranslates

class pygsp.filters.**WarpedTranslates**(*G*, *Nf=6*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Creates a vertex frequency filterbank

Parameters **G** : Graph

Nf : int

Number of filters

Returns **out** : WarpedTranslates

Examples

Not Implemented for now # >>> from pygsp import graphs, filters # >>> G = graphs.Logo() # >>> F = filters.WarpedTranslates(G)

See *[SWHV13]*

Papadakis

class pygsp.filters.**Papadakis**(*G*, *a=0.75*, ***kwargs*)
Bases: pygsp.filters.filter.Filter

Papadakis Filterbank

Inherits its methods from Filters

This function create a parseval filterbank of 2. The low-pass filter is defined by a function $f_l(x)$

$$f_l = \begin{cases} 1 & \text{if } x \leq a \\ \sqrt{1 - \frac{\sin\left(\frac{3\pi}{2a}x\right)}{2}} & \text{if } a < x \leq \frac{5a}{3} \\ 0 & \text{if } x > \frac{5a}{3} \end{cases}$$

The high pass filter is adaptated to obtain a tight frame.

Parameters **G** : Graph

a : float

See equation above for this parameter The spectrum is scaled between 0 and 2 (default = 3/4)

Returns **out** : Papadakis

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Papadakis(G)
```

Regular

class pygsp.filters.**Regular**(*G*, *d=3*, ***kwargs*)
Bases: pygsp.filters.filter.Filter

Regular Filterbank

Inherits its methods from Filters

This function creates a parseval filterbank 2 filters. The low-pass filter is defined by a function $f_l(x)$ between 0 and 2. For $d = 0$.

$$f_l = \sin\left(\frac{\pi}{4}x\right)$$

For $d = 1$

$$f_l = \sin\left(\frac{\pi}{4}\left(1 + \sin\left(\frac{\pi}{2}(x-1)\right)\right)\right)$$

For $d = 2$

$$f_l = \sin\left(\frac{\pi}{4}\left(1 + \sin\left(\frac{\pi}{2}\sin\left(\frac{\pi}{2}(x-1)\right)\right)\right)\right)$$

And so for other degrees d

The high pass filter is adaptated to obtain a tight frame.

Parameters **G** : Graph

d : float

See equations above for this parameter Degree (default = 3)

Returns **out** : Regular

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Regular(G)
```

Simoncelli

class pygsp.filters.**Simoncelli**(*G*, *a=0.6666666666666666*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Simoncelli Filterbank

Inherits its methods from Filters

This function create a parseval filterbank of 2. The low-pass filter is defined by a function $f_l(x)$.

$$f_l = \begin{cases} 1 & \text{if } x \leq a \\ \cos\left(\frac{\pi}{2}\frac{\log\left(\frac{x}{2}\right)}{\log(2)}\right) & \text{if } a < x \leq 2a \\ 0 & \text{if } x > 2a \end{cases}$$

The high pass filter is is adaptated to obtain a tight frame.

Parameters **G** : Graph

a : float

See equation above for this parameter The spectrum is scaled between 0 and 2 (default = 2/3)

Returns **out** : Simoncelli

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Simoncelli(G)
```

Held

class pygsp.filters.**Held**(*G*, *a=0.6666666666666666*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Held Filterbank

Inherits its methods from Filters

This function create a parseval filterbank of 2 filters. The low-pass filter is defined by a function $f_l(x)$

$$f_l = \begin{cases} 1 & \text{if } x \leq a \\ \sin\left(2\pi\mu\left(\frac{x}{8a}\right)\right) & \text{if } a < x \leq 2a \\ 0 & \text{if } x > 2a \end{cases}$$

with

$$\mu(x) = -1 + 24x - 144 * x^2 + 256 * x^3$$

The high pass filter is adaptated to obtain a tight frame.

Parameters **G** : Graph

a : float

See equation above for this parameter The spectrum is scaled between 0 and 2 (default = 2/3)

Returns **out** : Held

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Held(G)
```

Heat

class pygsp.filters.**Heat**(*G*, *tau=10*, *normalize=False*, ***kwargs*)

Bases: pygsp.filters.filter.Filter

Heat Filterbank

Inherits its methods from Filters

Parameters **G** : Graph

tau : int or list of ints

Scaling parameter. (default = 10)

normalize : bool

Normalize the kernel (works only if the eigenvalues are present in the graph). (default = 0)

Returns **out** : Heat

Examples

```
>>> from pygsp import graphs, filters
>>> G = graphs.Logo()
>>> F = filters.Heat(G)
```

This module implements filters and contains predefined filters that can be directly applied to graphs.

A filter is associated to a graph and is defined with one or several function(s). We define by Filterbank a list of filters applied to a single graph. Tools for the analysis, the synthesis and the evaluation are provided to work with the filters on the graphs. For specific information, *see details here*.

Operators

This module implements the main operators for the PyGSP box.

Operators functions

Adj2vec

pygsp.operators.**adj2vec**(*G*)

Prepare the graph for the gradient computation.

Parameters **G** : Graph structure

Divergence

pygsp.operators.**div**(*G*, *s*)

Compute Graph divergence of a signal.

Parameters **G** : Graph structure

s : ndarray

Signal living on the nodes

Returns **di** : float

The graph divergence

Gradient

pygsp.operators.**grad**(*G*, *s*)

Compute the Graph gradient.

Parameters **G** : Graph structure

s : ndarray

Signal living on the nodes

Returns **gr** : ndarray

Gradient living on the edges

Gradient Matriciel

pygsp.operators.**grad_mat**(*G*)

Gradient sparse matrix of the graph G.

Parameters **G** : Graph structure

Returns **D** : ndarray

Gradient sparse matrix

Gwft

pygsp.operators.**generalized_wft**(*G*, *g*, *f*, *lowmemory=True*)

Graph windowed Fourier transform

Parameters **G** : Graph

g : ndarray or Filter

Window (graph signal or kernel)

f : ndarray

Graph signal

lowmemory : bool

use less memory (default=True)

Returns **C** : ndarray

Coefficients

Gwft2

pygsp.operators.**gabor_wft**(*G*, *f*, *k*)

Graph windowed Fourier transform

Parameters **G** : Graph

f : ndarray

Graph signal

k : #TODO

kernel

Returns **C** : Coefficient.

Gwft Frame Matrix

pygsp.operators.**gwft_frame_matrix**(*G*, *g*)

Create the matrix of the GWFT frame

Parameters **G** : Graph

g : window

Returns **F** : ndarray

Frame

Igft

pygsp.operators.**igft**(*G*, *f_hat*)
Compute inverse graph Fourier transform.

Parameters **G** : Graph or Fourier basis

f_hat : ndarray

Signal

Returns **f** : ndarray

Inverse graph Fourier transform of *f_hat*

Ngwft

pygsp.operators.**ngwft**(*G*, *f*, *g*, *lowmemory=True*)
Normalized graph windowed Fourier transform

Parameters **G** : Graph

f : ndarray

Graph signal

g : ndarray

Window

lowmemory : bool

Use less memory. (default = True)

Returns **C** : ndarray

Coefficients

Ngwft Frame Matrix

pygsp.operators.**ngwft_frame_matrix**(*G*, *g*)
Create the matrix of the GWFT frame

Parameters **G** : Graph

g : ndarray

Window

Output parameters:

F : ndarray

Frame

Compute Fourier Basis

Compute Chebyshev Coefficient

pygsp.operators.**compute_cheby_coeff**(*f*, **args*, ***kwargs*)

Chebyshev Operator

pygsp.operators.**cheby_op**(*G*, *c*, *signal*, ***kwargs*)

Chebyshev polynomial of graph Laplacian applied to vector.

Parameters **G** : Graph

c : ndarray

Chebyshev coefficients

signal : ndarray

Signal to filter

Returns **r** : ndarray

Result of the filtering

Localize

pygsp.operators.**localize**(*g*, *i*)

Localize a kernel g to the node i.

Parameters **g** : Filter

kernel (or filterbank)

i : int

Index of vertex

Returns **gt** : ndarray

Translated signal

Modulate

pygsp.operators.**modulate**(*G*, *f*, *k*)

Tranlate the signal f to the node i.

Parameters **G** : Graph

f : ndarray

Signal (column)

k : int

Index of frequencies

Returns **fm** : ndarray

Modulated signal

Translate

pygsp.operators.**translate**(*G*, *f*, *i*)

Tranlate the signal f to the node i

Parameters **G** : Graph

f : ndarray

Signal

i : int

Indices of vertex

Returns **ft** : translate signal

PointsCloud

This module implements some PointsClouds.

PointsCloud

PointsCloud Class

class pygsp.pointsclouds.**PointsCloud**(*pointcloudname*, *max_dim=2*)

Bases: object

Load the parameters of models and the points.

Parameters **name** : string

The name of the point cloud to load. Possible arguments : 'airfoil', 'bunny', 'david64', 'david500', 'logo', 'minnesota', two_moons'.

max_dim : int

The maximum dimensionality of the points (only valid for two_moons) (default is 2)

Returns The differents informations of the loaded PointsCloud.

References

See *[TL94]* for more informations.

Examples

```
>>> from pygsp import pointsclouds
>>> bunny = pointsclouds.PointsCloud('bunny')
>>> Xin = bunny.Xin
```

plot(***kwargs*)

Plot the pointcloud.

See plotting doc.

Plotting

This module implements plotting functions for the PyGSP main objects.

Plotting functions

Plot

pygsp.plotting.**plot**(*O*, *default_qtg=True*, ***kwargs*)

Main plotting function.

This function should be able to determine the appropriate plot for the object. Additionnal kwargs may be given in case of filter plotting.

Parameters **O** : object

Should be either a Graph, Filter or PointCloud

default_qtg: boolean

Define the library to use if both are installed. Default is pyqtgraph (field=True).

Examples

```
>>> from pygsp import graphs, plotting
>>> G = graphs.Logo()
>>> try:
...     plotting.plot(G, default_qtg=False)
... except Exception as e:
...     print(e)
```

Plot Graph

pygsp.plotting.**plot_graph**(*G*, *default_qtg=True*, ***kwargs*)

Plot a graph or an array of graphs with installed libraries.

This function should be able to determine the appropriate plot for the graph. Additionnal kwargs may be given in case of filter plotting.

Parameters **G** : Graph

Graph object to plot

show_edges : boolean

Set to False to only draw the vertices (default G.Ne < 10000).

default_qtg: boolean

Define the library to use if both are installed. Default is pyqtgraph (field=True).

Examples

```
>>> from pygsp import graphs, plotting
>>> G = graphs.Logo()
>>> try:
...     plotting.plot_graph(G, default_qtg=False)
... except Exception as e:
...     print(e)
```

Plot Pointcloud

pygsp.plotting.**plot_pointcloud**(*P*)

Plot the coordinates of a pointcloud.

Parameters **P** : PointsClouds object

Examples

```
>>> from pygsp import plotting, pointsclouds
>>> logo = pointsclouds.PointsCloud('logo')
>>> try:
...     plotting.plot_pointcloud(logo)
... except:
...     pass
```

Plot Filter

pygsp.plotting.**plot_filter**(*filters*, *npoints=1000*, *line_width=4*, *x_width=3*, *x_size=10*, *plot_eigenvalues=None*, *show_sum=None*, *savefig=False*, *plot_name=None*)

Plot a system of graph spectral filters.

Parameters **filters** : filter object

npoints : int

Number of point where the filters are evaluated.

line_width : int

Width of the filters plots.

x_width : int

Width of the X marks representing the eigenvalues.

x_size : int

Size of the X marks representing the eigenvalues.

plot_eigenvalues : boolean

To plot black X marks at all eigenvalues of the graph (You need to compute the Fourier basis to use this option). By default the eigenvalues are plot if they are contained in the Graph.

show_sum : boolean

To plot an extra line showing the sum of the squared magnitudesof the filters (default True if there is multiple filters).

savefig : boolean

Determine wether the plot is saved as a PNG file in yourcurrent directory (True) or shown in a window (False) (default False).

plot_name : str

To give custom names to plots

Examples

```
>>> from pygsp import filters, plotting, graphs
>>> G = graphs.Logo()
>>> mh = filters.MexicanHat(G)
>>> try:
...     plotting.plot_filter(mh)
... except:
...     pass
```

Plot Signal

pygsp.plotting.**plot_signal**(*G*, *signal*, *default_qtg=True*, ***kwargs*)

Plot a graph signal in 2D or 3D with installed libraries.

Parameters **G** : Graph object

If not specified it will take the one used to create the filter.

signal : array of int

Signal applied to the graph.

show_edges : boolean

Set to False to only draw the vertices (default G.Ne < 10000).

cp : List of int

Camera position for a 3D graph.

vertex_size : int

Size of circle representing each signal component.

vertex_highlight : boolean

Vector of indices of vertices to be highlighted.

climits : array of int

Limits of the colorbar.

colorbar : boolean

To plot an extra line showing the sum of the squared magnitudes of the filters (default True if there is multiple filters).

bar : boolean

NOT IMPLEMENTED: False display color, True display bar for the graph (default False).

bar_width : int

Width of the bar (default 1).

default_qtg: boolean

Define the library to use if both are installed. Default is pyqtgraph (field=True).

Examples

```
>>> import numpy as np
>>> from pygsp import graphs, filters, plotting
>>> G = graphs.Ring(15)
>>> signal = np.sin((np.arange(1, 16)*2*np.pi/15))
>>> try:
...     plotting.plot_signal(G, signal, default_qtg=False)
... except:
...     pass
```

Reduction

Reduction functions

Graph Sparsify

Interpolate

Kron Pyramid

Kron Reduction

Pyramid Analysis

Pyramid Synthesis

Tree Depths

Tree Multiresolution

Optimization

This module provides optimization tools to accelarate graph signal processing as a whole.

Optimization

Prox TV

pygsp.optimization.**prox_tv**(*x*, *gamma*, *G*, *A=None*, *At=None*, *nu=1*, *tol=0.001*, *maxit=200*, *use_matrix=True*)

Total Variation proximal operator for graphs.

This function computes the TV proximal operator for graphs. The TV norm is the one norm of the gradient. The gradient is defined in the function `grad()`. This function require the PyUNLocBoX to be executed.

pygsp.optimization.prox_tv(y, gamma, param) solves:

$sol = \min_z \frac{1}{2}\|x - z\|_2^2 + \gamma\|x\|_{TV}$

Parameters **x: int**

Input signal

gamma: ndarray

Regularization parameter

G: graph object

Graphs structure

A: lambda function

Forward operator, this parameter allows to solve the following problem: $sol = \min_z \frac{1}{2}\|x - z\|_2^2 + \gamma\|Ax\|_{TV}$ (default = Id)

At: lambda function

Adjoint operator. (default = Id)

nu: float

Bound on the norm of the operator (default = 1)

tol: float

Stops criterion for the loop. The algorithm will stop if : $\frac{n(t)-n(t-1)}{n(t)} < tol$ where :math: *n(t) = f(x) + 0.5 |x-y|_2^2* is the objective function at iteration t (default = $10e-4$)

maxit: int

Maximum iteration. (default = 200)

use_matrix: bool

If a matrix should be used. (default = True)

Returns sol: solution

Examples

```
>>> from pygsp import optimization, graphs
```

Data Handling

Data Handling

Adj2vec

pygsp.data_handling.**adj2vec**(*G*)

Prepare the graph for the gradient computation.

Parameters **G** : Graph structure

Mat2vec

pygsp.data_handling.**mat2vec**(*d*)

Not implemented yet

Pyramid Cell2coeff

Repmatline

pygsp.data_handling.**repmatline**(*A*, *ncol=1*, *nrow=1*)

Repeat the matrix A in a specific manner.

Parameters **A** : ndarray

ncol : Integer

default is 1

nrow : Integer

default is 1

Returns **Ar** : Matrix

Examples

For nrow=2 and ncol=3, the matrix

```
x   =   [1 2 ]
        [3 4 ]
```

becomes

```
        [1 1 1 2 2 2 ]
M   =   [1 1 1 2 2 2 ]
        [3 3 3 4 4 4 ]
        [3 3 3 4 4 4 ]
```

with:: M = np.repeat(np.repeat(x, nrow, axis=1), ncol, axis=0)

Vec2mat

pygsp.data_handling.**vec2mat**(*d*, *Nf*)

Vector to matrix transformation.

Parameters **d** : Ndarray

Data

Nf : int

Number of filter

Returns **d** : list of ndarray

Data

Utils

This module implements some utilitary functions used throughout the PyGSP box.

Utils

Build logger

pygsp.utils.**build_logger**(*name*, ***kwargs*)

Graph array handler

pygsp.utils.**graph_array_handler**(*func*)

Filterbank handler

pygsp.utils.**filterbank_handler**(*func*)

Sparsifier

pygsp.utils.**sparsifier**(*func*)

Distanz

pygsp.utils.**distanz**(*x*, *y=None*)

Calculate the distanz between two colon vectors

Parameters **x** : ndarray

First colon vector

y : ndarray

Second colon vector

Returns **d** : ndarray

Distance between x and y

Examples

```
>>> import numpy as np
>>> from pygsp import utils
>>> x = np.random.rand(16)
>>> y = np.random.rand(16)
>>> distanz = utils.distanz(x, y)
```

Full eigen

Resistance distance

pygsp.utils.**resistance_distance**(*M*)

Compute the resistance distances of a graph.

Parameters **M** : Graph or sparse matrix

Graph structure or Laplacian matrix (L)

Returns **rd** : sparse matrix

distance matrix

Examples

```
>>>
>>>
>>>
```

CHAPTER 4

Contributing

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

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/epfl-lts2/pygsp/issues.

If you are reporting a bug, please include:

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

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to fix it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/epfl-lts2/pygsp/issues.

If you are proposing a feature:

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

Get Started!

Ready to contribute? Here's how to set up *pygsp* for local development.

1. Fork the *pygsp* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/pygsp.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv pygsp
$ cd pygsp/
$ python setup.py develop
```

Note: alternatively, the third step could be replaced by:

```
$ pip install -e .
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 pygsp tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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

```
$ git add *
$ git commit -m "Your detailed description of your changes."
$ git push --set-upstream origin name-of-your-branch
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.2, and 3.4, and for PyPy. Check https://travis-ci.org/epfl-lts2/pygsp/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python -m unittest tests.test_pygsp
```

CHAPTER 5

History

0.3.0 (2015-11-24)

Refactoring graphs using object programming and fail safe checks.

Refactoring filters to use only the Graph object used at the construction of the filter for all operations.

Refactoring Graph pyramid to match MATLAB implementation.

Removal of default coordinates (all vertices on the origin) for graphs that do not possess spatial meaning.

Correction of minor issues on Python3+ imports.

Various fixes.

Finalizing demos for the documentation.

0.2.1 (2015-10-14)

Fix bug on pip installation.

Update full documentation.

0.2.0 (2015-10-05)

Adding functionalities to match the content of the Matlab GSP Box.

First release of the PyGSP.

0.1.0 (2015-06-02)

Main features of the box are present most of the graphs and filters can be used.

The utils and operators modules also have most of their features implemented.

0.0.2 (2015-04-19)

Beginning of user tests.

0.0.1 (2014-10-06)

Toolbox template release.

CHAPTER 6

References

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Bibliography

[Gle] D. Gleich. The MatlabBGL Matlab library. http://www.cs.purdue.edu/homes/dgleich/packages/matlab_bgl/index.html.

[HVG11] D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *Appl. Comput. Harmon. Anal.*, 30(2):129–150, Mar. 2011.

[SWHV13] David I Shuman, Christoph Wiesmeyr, Nicki Holighaus, and Pierre Vandergheynst. Spectrum-adapted tight graph wavelet and vertex-frequency frames. *arXiv preprint arXiv:1311.0897*, 2013.

[Str99] Gilbert Strang. The discrete cosine transform. *SIAM review*, 41(1):135–147, 1999.

[TL94] Greg Turk and Marc Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 311–318. ACM, 1994.

Python Module Index

p

Index

M

N

P

R

S

T

U

V

W