GSPBOX: A toolbox for signal processing on graphs

Nathanael Perraudin, Johan Paratte, David Shuman, Lionel Martin
Vassilis Kalofolias, Pierre Vandergheynst and David K. Hammond

December 27, 2018

Abstract

This document introduces the Graph Signal Processing Toolbox (GSPBox) a framework that can be used to tackle graph related problems with a signal processing approach. It explains the structure and the organization of this software. It also contains a general description of the important modules.

1 Toolbox organization

In this document, we briefly describe the different modules available in the toolbox. For each of them, the main functions are briefly described. This chapter should help making the connection between the theoretical concepts introduced in [7, 9, 6] and the technical documentation provided with the toolbox. We highly recommend to read this document and the tutorial before using the toolbox. The documentation, the tutorials and other resources are available on-line[1].

The toolbox has first been implemented in MATLAB but a port to Python, called the PyGSP, has been made recently. As of the time of writing of this document, not all the functionalities have been ported to Python, but the main modules are already available. In the following, functions prefixed by **[M]:** refer to the MATLAB implementation and the ones prefixed with **[P]:** refer to the Python implementation.

1.1 General structure of the toolbox (MATLAB)

The general design of the GSPBox focuses around the graph object [7], a MATLAB structure containing the necessary informations to use most of the algorithms. By default, only a few attributes are available (see section 2), allowing only the use of a subset of functions. In order to enable the use of more algorithms, additional fields can be added to the graph structure. For example, the following line will compute the graph Fourier basis enabling exact filtering operations.

```
G = gsp_compute_fourier_basis(G);
```

Ideally, this operation should be done on the fly when exact filtering is required. Unfortunately, the lack of well defined class paradigm in MATLAB makes it too complicated to be implemented. Luckily, the above formulation prevents any unnecessary data copy of the data contained in the structure `G`. In order to avoid name conflicts, all functions in the GSPBox start with **[M]:** **`gsp_`**. A second important convention is that all functions applying a graph algorithm on a graph signal takes the graph as first argument. For example, the graph Fourier transform of the vector `f` is computed by

```
fhat = gsp_gft(G,f);
```

[1] See `https://lts2.epfl.ch/gsp/doc/` for MATLAB and `https://lts2.epfl.ch/pygsp` for Python. The full documentation is also available in a single document: `https://lts2.epfl.ch/gsp/gspbox.pdf`

The graph operators are described in section 4. Filtering a signal on a graph is also a linear operation. However, since the design of special filters (kernels) is important, they are regrouped in a dedicated module (see section 5).

The toolbox contains two additional important modules. The optimization module contains proximal operators, projections and solvers compatible with the UNLocBoX [5] (see section 6). These functions facilitate the definition of convex optimization problems using graphs. Finally, section **??** is composed of well known graph machine learning algorithms.

1.2 General structure of the toolbox (Python)

The structure of the Python toolbox follows closely the MATLAB one. The major difference comes from the fact that the Python implementation is object-oriented and thus allows for a natural use of instances of the graph object.

For example the equivalent of the MATLAB call:

```
G = gsp_estimate_lmax(G);
```

can be achieved using a simple method call on the graph object:

```
G.estimate_lmax()
```

Moreover, the use of class for the "graph object" allows to compute additional graph attributes on the fly, making the code clearer as its MATLAB equivalent. Note though that functionalities are grouped into different modules (one per section below) and that several functions that work on graphs have to be called directly from the modules. For example, one should write:

```
layers = pygsp.operators.kron_pyramid(G, levels)
```

This is the case as soon as the graph is the structure on which the action has to be performed and not our principal focus.

In a similar way to the MATLAB implementation using the UNLocBoX for the convex optimization routines, the Python implementation uses the PyUNLocBoX, which is the Python port of the UNLocBoX.

2 Graphs

The GSPBox is constructed around one main object: the graph. It is implemented as a structure in Matlab and as a class in Python. It stores the nodes, the edges and other attributes related to the graph. In the implementation, a graph is fully defined by the weight matrix $\mathbf{W}$, which is the main and only required attribute. Since most graph structures are far from fully connected, $\mathbf{W}$ is implemented as a sparse matrix. From the weight matrix a Laplacian matrix $\mathcal{L}$ is computed and stored as an attribute of the graph object. Different other attributes are available such as plotting attributes, vertex coordinates, the degree matrix, the number of vertices and edges. The list of all attributes is given in table 1.

Attribute	Format	Data type	Description
Mandatory fields			
`W`	NxN sparse matrix	double	Weight matrix $\mathbf{W}$
`L`	NxN sparse matrix	double	Laplacian matrix $\mathcal{L}$
`d`	Nx1 vector	double	The diagonal of the degree matrix
`N`	scalar	integer	Number of vertices
`Ne`	scalar	integer	Number of edges
`plotting`	**[M]: `structure` [P]: `dict`**	none	Plotting parameters
`type`	text	string	Name, type or short description
`directed`	scalar	**[M]: `logical` [P]: `boolean`**	State if the graph is directed or not
`lap_type`	text	string	Laplacian type
Optional fields			
`A`	NxN sparse matrix	**[M]: `logical` [P]: `boolean`**	Adjacency matrix
`coords`	Nx2 or Nx3 matrix	double	Vectors of coordinates in 2D or 3D.
`lmax`	scalar	double	Exact or estimated maximum eigenvalue
`U`	NxN matrix	double	Matrix of eigenvectors
`e`	Nx1 vector	double	Vector of eigenvalues
`mu`	scalar	double	Graph coherence

Table 1: Attributes of the graph object

The easiest way to create a graph is the **[M]: `gsp_graph` [P]: `pygsp.graphs.Graph`** function which takes the weight matrix as input. This function initializes a graph structure by creating the graph Laplacian and other useful attributes. Note that by default the toolbox uses the combinatorial definition of the Laplacian operator. Other Laplacians can be computed using the **[M]: `gsp_create_laplacian` [P]: `pygsp.gutils.create_laplacian`** function. Please note that almost all functions are dependent of the Laplacian definition. As a result, it is important to select the correct definition at first.

Many particular graphs are also available using helper functions such as : ring, path, comet, swiss roll, airfoil or two moons. In addition, functions are provided for usual non-deterministic graphs such as : Erdos-Renyi, community, Stochastic Block Model or sensor networks graphs.

Nearest Neighbors (NN) graphs form a class which is used in many applications and can be constructed from a set of points (or point cloud) using the **[M]: `gsp_nn_graph` [P]: `pygsp.graphs.NNGraph`** function. The function is highly tunable and can handle very large sets of points using FLANN [3].

Two particular cases of NN graphs have their dedicated helper functions : 3D point clouds and image patch-graphs. An example of the former can be seen in the function **[M]: `gsp_bunny` [P]: `pygsp.graphs.Bunny`**. As for the second, a graph can be created from an image by connecting similar patches of pixels together. The function **[M]: `gsp_patch_graph`** creates this graph. Parameters allow the resulting graph to vary between local and non-local and to use different distance functions [12, 4].

A few examples of the graphs are displayed in Figure1.

3 Plotting

As in many other domains, visualization is very important in graph signal processing. The most basic operation is to visualize graphs. This can be achieved using a call to the function **[M]: `gsp_plot_graph` [P]: `pygsp.plotting.plot_graph`**. In order to be displayable, a graph needs to have 2D (or 3D) coordinates (which is a field of the graph object). Some graphs do not possess default coordinates (e.g. Erdos-Renyi).

The toolbox also contains routines to plot signals living on graphs. The function dedicated to this task is **[M]: `gsp_plot_signal` [P]: `pygsp.plotting.plot_signal`**. For now, only 1D signals are supported. By default, the value of the signal is displayed using a color coding, but bars can be displayed by passing parameters.

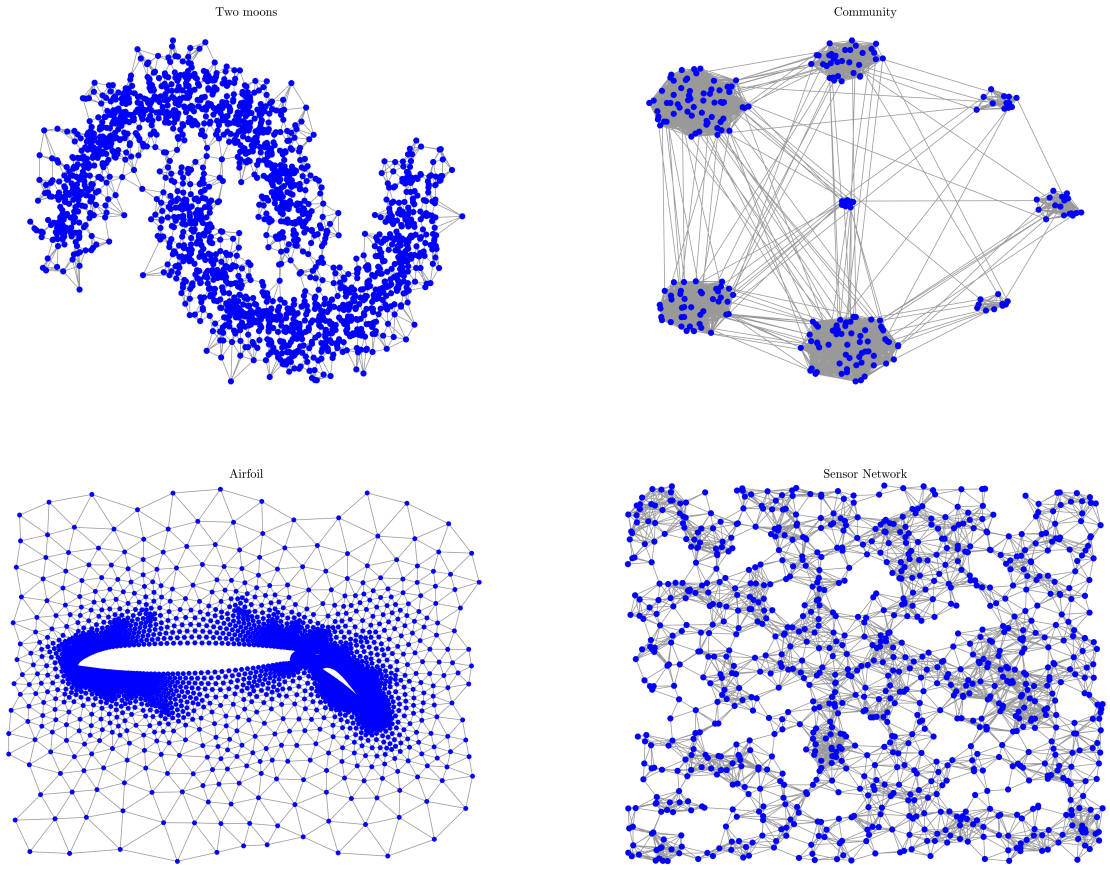

Figure 1: Examples of classical graphs : two moons (top left), community (top right), airfoil (bottom left) and sensor network (bottom right).

The third visualization helper is a function to plot filters (in the spectral domain) which is called **[M]: `gsp_plot_filter` [P]: `pygsp.plotting.plot_filter`**. It also supports filter-banks and allows to automatically inspect the related frames.

The results obtained using these three plotting functions are visible in Fig. 2.

4 Operators

The module operators contains basics spectral graph functions such as Fourier transform, localization, gradient, divergence or pyramid decomposition. Since all operator are based on the Laplacian definition, the necessary underlying objects (attributes) are all stored into a single object: the graph.

As a first example, the graph Fourier transform **[M]: `gsp_gft` [P]: `pygsp.operators.gft`** requires the Fourier basis. This attribute can be computed with the function **[M]: `gsp_compute_fourier_basis` [P]: `pygsp.graphs.compute_fourier_basis`** [9] that adds the fields `U`, `e` and `lmax` to the graph structure. As a second example, since the gradient and divergence operate on the edges of the graph, a search on the edge matrix is needed to enable the use of these operators. It can be done with the routines **[M]: `gsp_adj2vec` [P]: `pygsp.operators.adj2vec`**. These operations take time and should

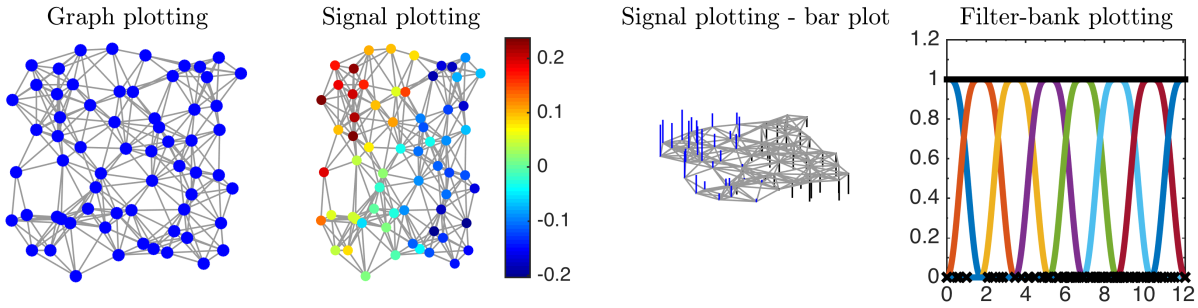

Figure 2: Visualization of graph and signals using plotting functions.

Name	Edge derivative $\frac{\partial f}{\partial_e}(i,j)$	Laplacian matrix (operator)	Available
Undirected graph			
Combinatorial Laplacian	$\sqrt{\mathbf{W}(i,j)}\,(f(j)-f(i))$	$\mathbf{D}-\mathbf{W}$	**V**
Normalized Laplacian	$\sqrt{\mathbf{W}(i,j)}\left(\frac{f(j)}{\sqrt{d(j)}}-\frac{f(i)}{\sqrt{d(i)}}\right)$	$\mathbf{D}^{-\frac{1}{2}}(\mathbf{D}-\mathbf{W})\mathbf{D}^{-\frac{1}{2}}$	**V**
Directed graph			
Combinatorial Laplacian	$\sqrt{\mathbf{W}(i,j)}\,(f(j)-f(i))$	$\frac{1}{2}\left(\mathbf{D}_+ + \mathbf{D}_- - \mathbf{W} - \mathbf{W}^*\right)$	**V**
Degree normalized Laplacian	$\sqrt{\mathbf{W}(i,j)}\left(\frac{f(j)}{\sqrt{d_-(j)}}-\frac{f(i)}{\sqrt{d_+(i)}}\right)$	$\mathbf{I}-\frac{1}{2}\left(\mathbf{D}_+^{-\frac{1}{2}}[\mathbf{W}+\mathbf{W}^*]\mathbf{D}_-^{-\frac{1}{2}}\right)$	**V**
Distribution normalized Laplacian	$\sqrt{\pi(i)}\left(\sqrt{\frac{p(i,j)}{\pi(j)}}f(j)-\sqrt{\frac{p(i,j)}{\pi(i)}}f(i)\right)$	$\frac{1}{2}\left(\mathbf{\Pi}^{\frac{1}{2}}\mathbf{P}\mathbf{\Pi}^{-\frac{1}{2}}+\mathbf{\Pi}^{-\frac{1}{2}}\mathbf{P}^*\mathbf{\Pi}^{\frac{1}{2}}\right)$	**V**

Table 2: Different definitions for graph Laplacian operator and their associated edge derivative. (For directed graph, $d_+, \mathbf{D}_+$ and $d_-, \mathbf{D}_-$ define the out degree and in-degree of a node. $\pi, \mathbf{\Pi}$ is the stationary distribution of the graph and $\mathbf{P}$ is a normalized weight matrix $\mathbf{W}$. For sake of clarity, exact definition of those quantities are not given here, but can be found in [14].)

be performed only once. In MATLAB, these functions are called explicitly by the user beforehand. However, in Python they are automatically called when needed and the result stored as an attribute.

The module operator also includes a Multi-scale Pyramid Transform for graph signals [6]. Again, it works in two steps. First the pyramid is precomputed with **[M]: `gsp_graph_multiresolution` [P]: `pygsp.operators.graph_multiresolution`**. Second the decomposition of a signal is performed with **[M]: `gsp_pyramid_analysis` [P]: `pygsp.operators.pyramid_analysis`**. The reconstruction uses **[M]: `gsp_pyramid_synthesis` [P]: `pygsp.operators.pyramid_synthesis`**.

The Laplacian is a special operator stored as a sparse matrix in the field `L` of the graph. Table 2 summarizes the available definitions. We are planning to implement additional ones.

5 Filters

Filters are a special kind of linear operators that are so prominent in the toolbox that they deserve their own module [9, 7, 2, 8, 2]. A filter is simply an anonymous function (in MATLAB) or a lambda function (in Python) acting element-by-element on the input. In MATLAB, a filter-bank is created simply by gathering these functions together into a cell array. For example, you would write:

```
% g(x) = x^2 + sin(x)
g = @(x) x.^2 + sin(x);
% h(x) = exp(-x)
h = @(x) exp(-x);
% Filterbank composed of g and h
fb = {g,h};
```

The toolbox contains many predefined design of filter. They all start with **[M]: `gsp_design_`** in MATLAB and are in the module **[P]: `pygsp.filters`** in Python. Once a filter (or a filter-bank) is created, it can be applied to a signal with **[M]: `gsp_filter_analysis`** in MATLAB and a call to the method **[P]: `analysis`** of the filter object in Python. Note that the toolbox uses accelerated algorithms to scale almost linearly with the number of sample [11].

The available type of filter design of the GSPBox can be classified as:

- Wavelets (Filters are scaled version of a mother window)
- Gabor (Filters are shifted version of a mother window)
- Low pass filter (Filters to de-noise a signal)
- High pass / Low pass separation filterbank (tight frame of 2 filters to separate the high frequencies from the low ones. No energy is lost in the process)

Additionally, to adapt the filter to the graph eigen-distribution, the warping function **[M]: `gsp_design_warped_translates`** **[P]: `pygsp.filters.WarpedTranslates`** can be used [10].

6 UNLocBoX Binding

This module contains special wrappers for the UNLocBoX[5]. It allows to solve convex problems containing graph terms very easily [13, 15, 14, 1]. For example, the proximal operator of the graph TV norm is given by **[M]: `gsp_prox_tv`**. The optimization module contains also some predefined problems such as graph basis pursuit in **[M]: `gsp_solve_l1`** or wavelet de-noising in **[M]: `gsp_wavelet_dn`**. There is still active work on this module so it is expected to grow rapidly in the future releases of the toolbox.

7 Toolbox conventions

7.1 General conventions

- As much as possible, all small letters are used for vectors (or vector stacked into a matrix) and capital are reserved for matrices. A notable exception is the creation of nearest neighbors graphs.
- A variable should never have the same name as an already existing function in MATLAB or Python respectively. This makes the code easier to read and less prone to errors. This is a best coding practice in general, but since both languages allow the override of built-in functions, a special care is needed.
- All function names should be lowercase. This avoids a lot of confusion because some computer architectures respect upper/lower casing and others do not.
- As much as possible, functions are named after the action they perform, rather than the algorithm they use, or the person who invented it.
- No global variables. Global variables makes it harder to debug and the code is harder to parallelize.

7.2 MATLAB

- All function start by `gsp_`.
- The graph structure is always the first argument in the function call. Filters are always second. Finally, optional parameter are last.
- In the toolbox, we do use any argument helper functions. As a result, optional argument are generally stacked into a graph structure named `param`.
- If a transform works on a matrix, it will per default work along the columns. This is a standard in Matlab (fft does this, among many other functions).
- Function names are traditionally written in uppercase in MATLAB documentation.

7.3 Python

- All functions should be part of a module, there should be no call directly from pygsp (**[P]: `pygsp.my_function`**).
- Inside a given module, functionalities can be further split in different files regrouping those that are used in the same context.
- MATLAB's matrix operations are sometimes ported in a different way that preserves the efficiency of the code. When matrix operations are necessary, they are all performed through the `numpy` and `scipy` libraries.
- Since Python does not come with a plotting library, we support both `matplotlib` and `pyqtgraph`. One should install the required libraries on his own. If both are correctly installed, then `pyqtgraph` is favoured unless specifically specified.

Acknowledgements

We would like to thanks all coding authors of the GSPBOX. The toolbox was ported in Python by Basile Châtillon, Alexandre Lafaye and Nicolas Rod. The toolbox was also improved by Nauman Shahid and Yann Schönenberger.

References

[1] M. Belkin, P. Niyogi, and V. Sindhwani. Manifold regularization: A geometric framework for learning from labeled and unlabeled examples. *The Journal of Machine Learning Research*, 7:2399–2434, 2006.

[2] D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

[3] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

[4] S. K. Narang, Y. H. Chao, and A. Ortega. Graph-wavelet filterbanks for edge-aware image processing. In *Statistical Signal Processing Workshop (SSP), 2012 IEEE*, pages 141–144. IEEE, 2012.

[5] N. Perraudin, D. Shuman, G. Puy, and P. Vandergheynst. UNLocBoX A matlab convex optimization toolbox using proximal splitting methods. *ArXiv e-prints*, Feb. 2014.

[6] D. I. Shuman, M. J. Faraji, and P. Vandergheynst. A multiscale pyramid transform for graph signals. *arXiv preprint arXiv:1308.4942*, 2013.

[7] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *Signal Processing Magazine, IEEE*, 30(3):83–98, 2013.

[8] D. I. Shuman, B. Ricaud, and P. Vandergheynst. A windowed graph Fourier transform. *Statistical Signal Processing Workshop (SSP), 2012 IEEE*, pages 133–136, 2012.

[9] D. I. Shuman, B. Ricaud, and P. Vandergheynst. Vertex-frequency analysis on graphs. *arXiv preprint arXiv:1307.5708*, 2013.

[10] D. I. Shuman, C. Wiesmeyr, N. Holighaus, and P. Vandergheynst. Spectrum-adapted tight graph wavelet and vertex-frequency frames. *arXiv preprint arXiv:1311.0897*, 2013.

[11] A. Susnjara, N. Perraudin, D. Kressner, and P. Vandergheynst. Accelerated filtering on graphs using lanczos method. *arXiv preprint arXiv:1509.04537*, 2015.

[12] F. Zhang and E. R. Hancock. Graph spectral image smoothing using the heat kernel. *Pattern Recognition*, 41(11):3328–3342, 2008.

[13] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf. Learning with local and global consistency. *Advances in neural information processing systems*, 16(16):321–328, 2004.

[14] D. Zhou, J. Huang, and B. Schölkopf. Learning from labeled and unlabeled data on a directed graph. In *the 22nd international conference*, pages 1036–1043, New York, New York, USA, 2005. ACM Press.

[15] D. Zhou and B. Schölkopf. A regularization framework for learning from graph data. 2004.