ENS de Lyon, M2 Computer Science
2023/2024 **Pierre Borgnat (CNRS, LP ENSL)**

Practical Session : ML and/for Graphs: Introduction to Graph Signal Processing

This is the following of the tutorial, now about graph signals and methods of graph signal processing.

Reminder: the practical sessions will be done preferentially in python, using the pygsp toolbox: https://pygsp.readthedocs.io/en/stable/tutorials/intro.html

This practical session was prepared by Nicolas Tremblay (CNRS, now at GIPSA-lab, Grenoble) and Michael Defferrard (formerly at EPFL, now at Qualcomm Research), and I warmly thanks them for that. You can find this tutorial, and more, online: https://github.com/mdeff/pygsp_tutorial_graphsip

- Step 3: Graph signals: gradient, divergence, smoothness -

A graph signal is a function $\mathcal{V} \to \mathbb{R}$ that associates a value to each node $v \in \mathcal{V}$ of a graph. The signal values can be represented as a vector $f \in \mathbb{R}^N$ where $N = |\mathcal{V}|$ is the number of nodes in the graph.

1) Let's generate a graph and a random signal. Then plot the signal on the graph to visualize it.

```
graph = graphs.Sensor(N=100)
signal = np.random.normal(size=graph.N)
graph.plot_signal(signal)
```

2) **Gradient and divergence.** The gradient $\nabla_\mathcal{G}\, f$ of the signal f on the graph $\mathcal{G}$ is a signal on the edges defined as

$$(\nabla_\mathcal{G})_{(i,j)}\, f = \sqrt{W_{ij}}(f_i - f_j)$$

Similarly, we can compute the divergence of an edge signal, which is again a signal on the nodes.

$$(\operatorname{div}_\mathcal{G} x)_i = \sum_{j\sim i} \sqrt{W_{ij}} x_{(i,j)}$$

```
graph.compute_differential_operator()
gradient = graph.D @ signal
assert gradient.size == graph.Ne
divergence = graph.D.T @ gradient
assert divergence.size == graph.N
graph.plot_signal(divergence)
```

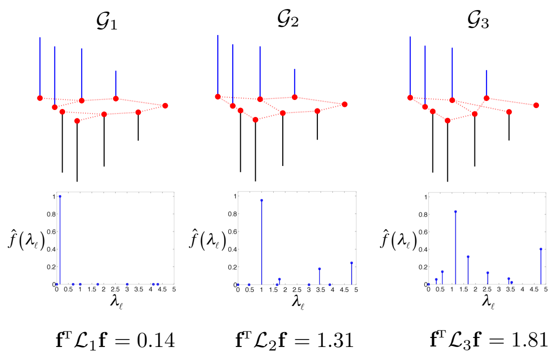
Compare the Laplacian operator ton the divergence of the gradient.

3) **Smoothness.** What is the smoothest graph signal, i.e. the signal f for which $f^\intercal L f = 0$? Verify computationally. What if L is the normalized Laplacian? Verify computationally.

Replicate the experiments of Figure 1.

- Step 4: Fourier: modes, transform -

As in classical signal processing, the Fourier transform plays a central role in graph signal processing. Getting the Fourier basis is however computationally intensive as it needs to fully diagonalize the Laplacian. While it can be used to filter signals on graphs, a better alternative is to use one of the fast approximations (see `pygsp.filters.Filter.filter`). Let's compute it nonetheless to visualize the eigenvectors of the Laplacian. Analogous to classical Fourier analysis, they look like sinuses on the graph. Let's plot the second and third eigenvectors (the first is constant). Those are graph signals, i.e. functions $s : \mathcal{V} \to \mathbb{R}^d$ which assign a set of values (a vector in $\mathbb{R}^d$ at every node $v \in \mathcal{V}$ of the graph.



1) **Graph Fourier Basis.**

The graph Fourier basis $U = [u_1, \dots, u_N]$, where u_i is the i^{th} Fourier mode, is given by the eigendecomposition of the graph Laplacian L such as

$$L = U\Lambda U^\intercal.$$

$\Lambda = \text{diag}([\lambda_1, \dots, \lambda_N])$ is the diagonal matrix of squared "graph frequencies".

Indeed, the following relation holds:

$$\lambda_i = u_i^\intercal L u_i$$

The parallel with classical signal processing is best seen on a ring graph, where the graph Fourier basis is equivalent to the classical Fourier basis. The following plot shows some eigenvectors drawn on a 1D and 2D embedding of the ring graph. While the signals are easier to interpret on a 1D plot (with `graph.set_coordinates('line1D')`, the 2D plot best represents the graph (with `graph.set_coordinates('ring')`).

Vizualize the eigenvectors on a ring graph (as a ring, or in 1D), then on the 2D Euclidean grid.

```
# Your code here
# Use the following functions:

G.compute_fourier_basis()

graph.plot_signal(graph.U[:, 4], ax=axes[0])
```

Finally, on more complicated domains, the intuition that lower frequencies should be smooth transfers.

```
graph = graphs.Logo()
```

Localization of some eigenvectors

A fundamental property of the classical Fourier modes is that they are delocalized. Put differently: a signal can never be localized in time/space and in the Fourier space. This Heisenberg principle does not transfer to graphs so easily. For instance, let us look at a very irregular graph: the comet graph.

```
from pygsp import filters

G = graphs.Comet(N=30, k=20)
graph.set_coordinates('spring')
G.compute_fourier_basis()
G.plot()

plt.figure(figsize=(10, 10))
plt.imshow(np.abs(G.U))
plt.colorbar()
print('The largest entry in the Fourier basis is ' + str(np.max(np.abs(G.U))))
```

2) **Graph Fourier transform.**

The spectral content of a signal indicates if the signal is low-pass, band-pass, or high-pass. Again, intuition transfers from classical Fourier analysis. Some examples:

```
G = graphs.Sensor(seed=42)
G.compute_fourier_basis()

taus = [0, 3, 10]
fig, axes = plt.subplots(len(taus), 2, figsize=(11, 6))

x0 = np.random.RandomState(1).normal(size=G.N)
for i, tau in enumerate(taus):
    g = filters.Heat(G, tau)
    x = g.filter(x0).squeeze()
    x_hat = G.gft(x).squeeze()

    G.plot_signal(x, ax=axes[i, 0])
    axes[i, 0].set_axis_off()
    axes[i, 0].text(0, -0.2, '$x^T L x = {:.2f}$'.format(x.T @ G.L @ x))

    axes[i, 1].plot(G.e, np.abs(x_hat), '.-')


axes[0, 0].set_title(r'$x$: signal in the vertex domain')
axes[0, 1].set_title(r'$\hat{x}$: signal in the spectral domain')
axes[-1, 1].set_xlabel("laplacian's eigenvalues / graph frequencies")
```

3) **Exercise.**

Verify $\lambda_i = u_i^\intercal L u_i$ computationally.

(a) Make a band-pass signal in the spectral domain.

(b) Compute the inverse Fourier transform.

(c) Visualize your signal in the vertex domain.

- Step 5: Filters – Heat Diffusion, Denoising,... -

To filter signals on graphs, we need to define filters. They are represented in the toolbox by the `pygsp.filters.Filter` class.
https://pygsp.readthedocs.io/en/stable/reference/filters.html
Filters are usually defined in the spectral domain, given a transfer function that weights the coefficients at the different eigenvalues.

Utilities:

```
import numpy as np
import matplotlib.pyplot as plt
from pygsp import graphs, filters
from additional_utils import get_approx_filter
import time
```

1) **Heat diffusion.**

The heat kernel $h(\lambda)$ is defined as:

$$h_\tau(\lambda) = \exp^{-\tau\lambda}.$$

The graph heat equation reads:

$$\frac{d\mathbf{x}}{dt} + \mathbf{L}\mathbf{x} = \mathbf{0}.$$

With $\hat{\mathbf{x}} = \mathbf{U}^\top \mathbf{x}$ is the Fourier transform of $\mathbf{x}$, this equation is easily solved:

$$\hat{\mathbf{x}} = \exp^{-t\boldsymbol{\Lambda}} \hat{\mathbf{x}}_0$$

where $\hat{x_0}$ is the initial signal at time $t = 0$. Such that:

$$\mathbf{x}(t) = \mathbf{U} \exp^{-t\boldsymbol{\Lambda}} \mathbf{U}^\top \mathbf{x}_0 = \mathbf{U} h_t(\boldsymbol{\Lambda}) \mathbf{U}^\top \mathbf{x}_0.$$

Apply this filter to a random sensor graph, or on a ring graph (you will see the gaussian kernel).

```
G1 = graphs.Sensor(seed=42)

G2 = graphs.Ring(N=100)
```

2) **Example of denoising.**
Let's define a low-pass filter

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

Given a noisy version of a smooth signal x_{noisy}, one can denoise it with the low-pass filter g:

$$x_{\text{denoised}} = \mathbf{U} g(\boldsymbol{\Lambda}) \mathbf{U}^\top x_{\text{noisy}}$$

Apply the filter to a noisy signal on a graph of your choice, e.g, the GSP one.

3) **Polynomial approximation.**
Let us approximate $g(x) = \dfrac{1}{1+x}$ on the interval $[0, \lambda_N]$ by a Chebychev polynomial of order m:

$$g(x) \simeq \sum_{k=0}^{m} \alpha_k x^k = p(x),$$

such that the exact filtering can be approximated by a polynomial in $\mathbf{L}$:

$$\begin{aligned} x_{\text{filtered}} &= \mathbf{U} g(\boldsymbol{\Lambda}) \mathbf{U}^\top x && (1)\\ &\simeq \mathbf{U} p(\boldsymbol{\Lambda}) \mathbf{U}^\top x && (2)\\ &= \mathbf{U} \sum_{k=0}^{m} \alpha_k \boldsymbol{\Lambda}^k \mathbf{U}^\top x && (3)\\ &= \sum_{k=0}^{m} \alpha_k \mathbf{L}^k x && (4) \end{aligned}$$

Note that computing $\sum_{k=0}^{m} \alpha_k \mathbf{L}^k x$ takes only m matrix-vector multiplication and costs thus $\mathcal{O}(m|E|)$ with $|E|$ the number of edges of the graph (compared to the $\mathcal{O}(N^3)$ necessary operations just to diagonalize $\mathbf{L}$ for the exact computation!)

Compare the computation times of the direct implementation and the approximation .
Compare also the precisions.