AIMS CDT Signal Processing Lab Session 2
Graph Signal Processing

Xiaowen Dong (xdong@robots.ox.ac.uk)

In this lab session[1], we will be looking into graph signal processing, a fast-growing field that deals with signals that are defined on the vertex set of weighted graphs. We will be using **PyGSP**, a Python package for graph signal processing operations. You can install **PyGSP** with **pip** or **conda**. Alternatively, if you prefer MATLAB, you can make use of **GSPBOX**, a counterpart of **PyGSP** developed by the same team that has the same functionality. The sample code provided in this lab notes would need to be adapted accordingly.

- **PyGSP**: `https://github.com/epfl-lts2/pygsp`
- **GSPBOX**: `https://github.com/epfl-lts2/gspbox`

Now let's get started. First we import the necessary Python modules.

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
from pygsp import graphs, filters, plotting
```

Graph. A graph consists of a set of nodes $\mathcal{V}$, and a set of edges $\mathcal{E}$ with associated edge weights. It can be encoded in the adjacency matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$ where $|\mathcal{V}| = N$. Graphs can be created using the *graphs* module in **PyGSP** (`https://pygsp.readthedocs.io/en/stable/reference/graphs.html`).

We can define a graph as a real world network, such as the road network in Minnesota, USA.

```
G = graphs.Minnesota()
G.coords.shape % coordinates are already set for nodes for visualisation
fig, axes = plt.subplots(1, 2)
_ = axes[0].spy(G.W, markersize=0.5) % visualise the adjacency matrix in a spy plot
G.plot(ax=axes[1]) % visualise the graph in 2D coordinates
```

Alternatively, we can define a graph as an instance of a random graph model, e.g., an Erdős-Rényi graph.

```
G = graphs.ErdosRenyi(N=30, p=0.2)
G.set_coordinates(kind='spring') % set coordinates for nodes for visualisation
fig, axes = plt.subplots(1, 2)
_ = axes[0].spy(G.W, markersize=2)
G.plot(ax=axes[1])
```

[1]This lab session was largely inspired by the exercises built by Michaél Defferrard, Effrosyni Simou, and Hermina Petric Maretić for the course *A Network Tour of Data Science* at EPFL and that by Michaél Defferrard and Nicolas Tremblay for the GraphSiP Summer School in 2018.

Graph Laplacian. The unnormalised (combinatorial) graph Laplacian matrix is defined as:

$$\mathbf{L} = \mathbf{D} - \mathbf{W},$$

where $\mathbf{D}$ is the matrix that contains the degree of the nodes along diagonal. Alternatively, the normalised graph Laplacian matrix is defined as:

$$\mathbf{L}_{\text{norm}} = \mathbf{D}^{-\frac{1}{2}}(\mathbf{D} - \mathbf{W})\mathbf{D}^{-\frac{1}{2}}.$$

Let's compute and visualise the graph Laplacian matrix $\mathbf{L}$.

```
G.compute_laplacian('combinatorial')
fig, axes = plt.subplots(1, 2)
axes[0].spy(G.L, markersize=5)
axes[1].hist(G.L.data, bins=50, log=True);
```

A spy plot is a visualisation of a matrix in which there is a dot for each non-zero entry in the matrix.

- Can you observe the difference between $\mathbf{W}$ and $\mathbf{L}$ from their spy plots?
- Can you understand the histogram for the entries of $\mathbf{L}$?

Now compute the normalised graph Laplacian $\mathbf{L}_{\text{norm}}$.

- Can you understand the histogram for the entries of $\mathbf{L}$?

Graph signal. A graph signal $\mathbf{f} : \mathcal{V} \to \mathbb{R}^N$ assigns a scalar value to each node of the graph. Let's plot a random graph signal on a ring graph.

```
G = graphs.Ring(N=60)
f = np.random.normal(size=G.N)
G.plot_signal(f, vertex_size=50)
```

The smoothness of a signal $\mathbf{f}$ on the graph can be evaluated by the Laplacian quadratic form:

$$\mathbf{f}^T\mathbf{L}\mathbf{f} = \frac{1}{2}\sum_{i,j=1}^{N} \mathbf{W}_{ij}(\mathbf{f}(i) - \mathbf{f}(j))^2,$$

i.e., the smaller the quantity of $\mathbf{f}^T\mathbf{L}\mathbf{f}$, the smoother the signal on the graph (less variation of signal values across edges).

- Compute the Laplacian quadratic form for $\mathbf{f}$. Now sort the values in $\mathbf{f}$ in increasing order and compute the Laplacian quadratic form again. How do the two quantities compare, and why? (Note: You can plot both the original and sorted signals on the graph and compare.)
- This simple 'sorting trick' does not always make the signal smoother on an arbitrary graph. Can you see why?

Fourier basis. The eigenvectors of the graph Laplacian matrix provides a Fourier-like basis for signals on graphs. Let's first look at the Fourier basis defined by the graph Laplacian of the ring graph.

```
G.compute_fourier_basis()
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
G.plot_signal(G.U[:, 2], vertex_size=50, ax=axes[0]) % the 3rd eigenvector as a signal on the graph
G.set_coordinates('line1D')
G.plot_signal(G.U[:, 0:3], ax=axes[1]) % the first three eigenvectors on the real line
fig.tight_layout()
```

- Can you observe the equivalence between the classical Fourier basis and the Fourier basis defined on the ring graph?

Now plot the first 5 eigenvectors of the 2D grid graph and the Minnesota road network. (Note: The Minnesota road network consists of many vertices, so it is better to set vertex_size $= 5$.)

```
G = graphs.Grid2d(10, 10)
G = graphs.Minnesota()
```

- What do you observe about these eigenvectors?
- Compute the Laplacian quadratic form for the first 5 eigenvectors of the 2D grid graph and the Minnesota road network. How do the quantities vary, and why?

Graph Fourier transform (GFT). Similar to the classical Fourier transform, the GFT transforms the graph signal from the vertex domain into the graph spectral domain. Let's examine the Fourier coefficients of a random signal $\mathbf{f}$ on a community graph, i.e., a graph in which nodes form a number of communities.

```
communities = [50, 50, 50]
% 'comm_density' and 'world_density' determines intra- and inter-community edge probability
G = graphs.Community(N=150, Nc=3, comm_sizes=communities, comm_density=0.2, world_density=0.01)
f = np.random.normal(size=G.N)
G.compute_fourier_basis()
f_hat = G.gft(f)

limits = [np.min(f)-0.1, np.max(f)+0.1]
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
G.plot_signal(f, vertex_size=20, ax=axes[0], limits=limits)
axes[1].plot(G.e, np.abs(f_hat), '.-')
```

- Can you comment on the smoothness of $\mathbf{f}$ on the graph, from its Fourier coefficients?

Now let's consider another signal, i.e., the partition function:

$$\mathbf{f}_{\text{part}}(i) = \begin{cases} -1, & \text{if } i \in C_1, \\ 0, & \text{if } i \in C_2, \\ 1, & \text{if } i \in C_3, \end{cases}$$

where C_1, C_2, and C_3 denote the three communities.

```
fp = np.zeros(G.N)
fp[:communities[0]] = -1 * np.ones(communities[0])
fp[-communities[-1]:] = 1 * np.ones(communities[-1])
```

- Plot both $\mathbf{f}$ and $\mathbf{f}_{\text{part}}$ on the graph. What do you observe?
- Compute the Fourier coefficients of $\mathbf{f}_{\text{part}}$ and compare with that of $\mathbf{f}$. What do you observe, and why?
- Compare the Laplacian quadratic form of $\mathbf{f}$ and $\mathbf{f}_{\text{part}}$. Is it consistent with what you observe above? (Note: For a fair comparison, you should divide the Laplacian quadratic form by the squared norm of the signal.)

[Checkpoint 1] You are now done with the first part of this lab session. Please ask a lab demonstrator to evaluate your work in this part.

Filtering. Just like in classical signal processing, we can also apply filtering to graph signals. The main idea is to transform the graph signal into the graph spectral domain, attenuate unwanted frequencies or amplify desired frequencies of the signal by modifying the corresponding Fourier coefficients, and convert the signal back to the vertex domain. This modification is done via a filter function $g(\lambda)$ in the graph spectral domain, where λ is the eigenvalue of the graph Laplacian. Let's start with the following low-pass filtering example, using the random signal $\mathbf{f}$ and the community graph defined before.

```
import copy
fb_hat = copy.deepcopy(f_hat)
fb_hat[20:] = 0 % this corresponds to an ideal brick-wall low-pass filtering
fb = G.igft(fb_hat)
```

- Plot both the original signal $\mathbf{f}$ and the filtered signal $\mathbf{f}_{\text{brick}}$ on the graph. What do you observe?

More sophisticated filtering operations can be done using the *filters* module in **PyGSP** (`https://pygsp.readthedocs.io/en/stable/reference/filters.html`). For example, another popular low-pass filter is the *heat kernel*:

$$g(\lambda) = e^{-\tau\lambda},$$

where τ is a parameter that controls the decay of the function.

```
tau = 10
delta = 60 % define a Kronecker delta function having value 1 on node 60 and 0 elsewhere
g = filters.Heat(G, tau) % the heat kernel defined via the graph spectral domain
s = g.localize(delta) % the heat kernel localised at node 60 in the graph
fig, axes = plt.subplots(1, 2, figsize=(10, 4))
G.plot_signal(s, vertex_size=20, ax=axes[0])
g.plot(ax=axes[1])
```

From the plot, you can see that in the graph spectral domain the filter penalises more the high frequencies (larger eigenvalues). This filter can be visualised in the vertex domain as well, by "localising" the heat kernel at a particular node. This corresponds to applying the filter to a graph signal that is a Kronecker delta function, i.e., having value 1 on the chosen node and 0 otherwise. Now apply the filter to our signal $\mathbf{f}$, and obtained the filtered signal $\mathbf{f}_{\text{heat}}$.

```
fh = g.filter(f)
```

- Verify that localising the heat kernel at a particular node is equivalent to applying filtering to the corresponding Kronecker delta function;
- Plot the original signal $\mathbf{f}$, the filtered signal $\mathbf{f}_{\text{brick}}$ by the brick-wall low-pass filter, and the filtered signal $\mathbf{f}_{\text{heat}}$ by the heat kernel on the graph. How do the two filtered signals compare?
- What if we repeat the heat kernel based filtering with a smaller τ or bigger τ?

We can define our own filters by designing the function $g(\lambda)$. For example, the following filter

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

can be defined as follows.

```
g = filters.Filter(G, lambda x: 1. / (1. + x))
```

Denoising. Denoising is a classical signal processing problem where we wish to remove noise from the observed signal. Denoising for a graph signal can be formulated as the following optimisation problem:

$$\min_{\mathbf{f}\in\mathbb{R}^N} ||\mathbf{y}-\mathbf{f}||_2^2 + \alpha\, \mathbf{f}^T\mathbf{L}\mathbf{f}, \tag{1}$$

where $\mathbf{y}$ is the observed noisy graph signal. The optimisation problem in Eq. (1) tries to find an $\mathbf{f}$ that is close to $\mathbf{y}$, and at the same time being smooth on the graph. The regularisation parameter α controls the trade-off between the data fidelity term and the smoothness prior. The solution can therefore be considered as a denoised version of the observed noisy signal.

- Show that the solution to the optimisation problem in Eq. (1) can be interpreted as applying a low-pass filter to the observed noisy graph signal. You may find *The Matrix Cookbook* (`http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=3274`) useful in terms of differentiation with respect to a matrix.
- Implement the solution to Eq. (1) via the following steps:
  1. Construct a sensor network using the class *pygsp.graphs.Sensor* in **PyGSP**, and compute the graph Laplacian;
  2. Construct a smooth signal on the graph, e.g., a linear combination of the first few eigenvectors of the Laplacian;
  3. Add Gaussian random noise (with appropriate variance) to the smooth signal, to obtain the noisy signal;
  4. Define the low-pass filter with a given α, and apply it to the noisy signal, to obtain the denoised signal; compare the clean, noisy, and denoised signals on the graph.
  5. Repeat with a different α. What is the impact of the value of α on the denoised signal? Can you find an optimal α that minimises the error between the clean and denoised signals?

Inpainting. Inpainting is another classical signal processing problem where we wish to fill in the missing values in a partially observed signal. This is typically done in the context of image processing for inferring missing pixel values in an image. Inpainting for a graph signal can be formulated as the following optimisation problem:

$$\min_{\mathbf{f}\in\mathbb{R}^N} ||\mathbf{y} - \mathbf{M}\mathbf{f}||_2^2 + \alpha\, \mathbf{f}^T\mathbf{L}\mathbf{f}, \tag{2}$$

where $\mathbf{y}$ is a partially observed graph signal (with missing values being 0), and $\mathbf{M}$ is a diagonal matrix that satisfies:

$$\mathbf{M}(i,i) = \begin{cases} 1, & \text{if } \mathbf{y}(i) \text{ is observed,} \\ 0, & \text{if } \mathbf{y}(i) \text{ is not observed.} \end{cases}$$

The optimisation problem in Eq. (2) tries to find an $\mathbf{f}$ that matches the observed values in $\mathbf{y}$, and at the same time being smooth on the graph. The regularisation parameter α controls the trade-off between the data fidelity term and the smoothness prior. The solution can therefore be considered as an inpainted version of the partially observed signal.

- Derive the solution to the optimisation problem in Eq. (2). Can it be interpreted as applying a filter to the partially observed graph signal?
- Implement the solution to Eq. (2) via the following steps:
  1. Construct a sensor network using the class *pygsp.graphs.Sensor* in **PyGSP**, and compute the graph Laplacian;
  2. Construct a smooth signal on the graph, e.g., a linear combination of the first few eigenvectors of the Laplacian;
  3. Scale the values of the signal to the range [0, 1] (so that missing values are easily seen);
  4. Randomly set $p = 50\%$ of the signal values to 0, to obtain the partially observed signal;
  5. Solve Eq. (2) using *np.linalg.solve()*, to obtain the inpainted signal; compare the fully observed, partially observed, and inpainted signals on the graph.
  6. Repeat with a different p and α. What is the impact of the value of p and α on the inpainted signal? How large can p be so that a reasonably recovery is still possible?
- Now, can you repeat this experiment on a standard test image, e.g., the Cameraman image (`https://testimages.juliaimages.org/stable/imagelist/`)? Hints:
  1. An image can be thought of as a regular grid graph that can be generated using the class *pygsp.graphs.Grid2d* in **PyGSP**;
  2. Try a grayscale version of the image with a relatively low resolution, e.g., 64 by 64 or 128 by 128. You may find the Python library *Imageio* (`https://imageio.github.io`) useful in terms of reading and writing image data.
  3. Obviously we shouldn't need to implement inpainting by treating an image as a graph. In addition, the formulation in Eq. (2) corresponds to inpainting by panelising high-frequency coefficients in the Fourier basis. More effectively, image inpainting can be done by sparsifying the signal representation in the wavelet basis. Try out such an exercise at `https://nbviewer.org/github/gpeyre/numerical-tours/blob/master/python/inverse_5_inpainting_sparsity.ipynb` and compare with the graph- and Fourier-based solution above.

[Checkpoint 2] You are now done with the second part of this lab session. Please ask a lab demonstrator to evaluate your work in this part.