Gasper: GrAph Signal ProcEssing in R

Basile de Loynes, Fabien Navarro, Baptiste Olivier

2024-02-28

Abstract

We present a short tutorial on to the use of the R **gasper** package. Gasper is a package dedicated to signal processing on graphs. It also provides an interface to the SuiteSparse Matrix Collection.

1 Introduction

The emerging field of Graph Signal Processing (GSP) aims to bridge the gap between signal processing and spectral graph theory. One of the objectives is to generalize fundamental analysis operations from regular grid signals to irregular structures in the form of graphs. There is an abundant literature on GSP, in particular we refer the reader to Shuman et al. (2013) and Ortega et al. (2018) for an introduction to this field and an overview of recent developments, challenges and applications. GSP has also given rise to numerous applications in machine/deep learning: convolutional neural networks (CNN) on graphs Bruna et al. (2014), Henaff et al. (2015), Defferrard et al. (2016), semi-supervised classification with graph CNN Kipf and Welling (2017), Hamilton et al. (2017), community detection Tremblay and Borgnat (2014), to name just a few.

Different software programs exist for processing signals on graphs, in different languages. The Graph Signal Processing toolbox (GSPbox) is an easy to use matlab toolbox that performs a wide variety of operations on graphs. This toolbox was port to Python as the PyGSP Perraudin et al. (2014). There is also another matlab toolbox the Spectral Graph Wavelet Transform (SGWT) toolbox dedicated to the implementation of the SGWT developed in Hammond et al. (2011). However, to our knowledge, there are not yet any tools dedicated to GSP in R. A development version of the **gasper** package is currently available online[1], while the latest stable release can be obtained from the Comprehensive R Archive Network[2]. In particular, it includes the methodology and codes[3] developed in de Loynes et al. (2021) and provides an interface to the SuiteSparse Matrix Collection Davis and Hu (2011).

This vignette is organized as follows. Section 2 introduces the interface to the SuiteSparse Matrix Collection and some visualization tools for GSP. Section 3 gives a short introduction to some underlying concepts of GSP, focusing on the Graph Fourier Transform. Section 4 gives an illustration for denoising signals on graphs using SGWT, thresholding techniques, and the minimization of Stein Unbiased Risk Estimator for an automatic selection of the threshold parameter.

[1]https://github.com/fabnavarro/gasper
[2]https://cran.r-project.org/web/packages/gasper/
[3]https://github.com/fabnavarro/SGWT-SURE

2 Graphs Collection and Visualization

A certain number of graphs are present in the package. They are stored as an Rdata file which contains a list consisting of the graph's weight matrix W (in the form of a sparse matrix denoted by `sA`) and the coordinates associated with the graph (if it has any).

An interface is also provided. It allows to retrieve the matrices related to many problems provided by the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) Davis and Hu (2011), Kolodziej et al. (2019). This collection is a large and actively growing set of sparse matrices that arise in real applications (as structural engineering, computational fluid dynamics, computer graphics/vision, optimization, economic and financial modeling, mathematics and statistics, to name just a few). For more details see https://sparse.tamu.edu/.

The package includes the `SuiteSparseData` dataset, which contains data from the SuiteSparse Matrix Collection. The structure of this dataframe mirrors the structure of the main table presented on the SuiteSparse Matrix Collection website, allowing users to query and explore the dataset directly within R.

Here is a sample of the `SuiteSparseData` dataset, showing the first 15 rows of the table:

```r
head(SuiteSparseData, 15)
```

Table 1: Overview of the first 15 matrices from the SuiteSparse Matrix Collection.

ID	Name	Group	Rows	Cols	Nonzeros	Kind	Date
1	1138_bus	HB	1138	1138	4054	Power Network Problem	1985
2	494_bus	HB	494	494	1666	Power Network Problem	1985
3	662_bus	HB	662	662	2474	Power Network Problem	1985
4	685_bus	HB	685	685	3249	Power Network Problem	1985
5	abb313	HB	313	176	1557	Least Squares Problem	1974
6	arc130	HB	130	130	1037	Materials Problem	1974
7	ash219	HB	219	85	438	Least Squares Problem	1974
8	ash292	HB	292	292	2208	Least Squares Problem	1974
9	ash331	HB	331	104	662	Least Squares Problem	1974
10	ash608	HB	608	188	1216	Least Squares Problem	1974
11	ash85	HB	85	85	523	Least Squares Problem	1974
12	ash958	HB	958	292	1916	Least Squares Problem	1974
13	bcspwr01	HB	39	39	131	Power Network Problem	1981
14	bcspwr02	HB	49	49	167	Power Network Problem	1981
15	bcspwr03	HB	118	118	476	Power Network Problem	1981

For example, to retrieve all undirected graphs with between 100 and 150 columns and rows:

```r
filtered_mat <- SuiteSparseData[SuiteSparseData$Kind == "Undirected Graph" &
                  SuiteSparseData$Rows >= 100 & SuiteSparseData$Rows <= 150 &
                  SuiteSparseData$Cols >= 100 & SuiteSparseData$Cols <= 150, ]
filtered_mat
```

Table 2: Subset of undirected matrices with 100 to 150 rows and columns.

	ID	Name	Group	Rows	Cols	Nonzeros	Kind	Date
1484	1484	GD06_theory	Pajek	101	101	380	Undirected Graph	2006
1497	1497	GD98_c	Pajek	112	112	336	Undirected Graph	1998
2389	2389	adjnoun	Newman	112	112	850	Undirected Graph	2006
2403	2403	polbooks	Newman	105	105	882	Undirected Graph	2001

The `download_graph` function allows to download a matrix from this collection, based on the name of the matrix and the name of the group that provides it. An example is given below

```
matrixname <- "grid1"
groupname <- "AG-Monien"
download_graph(matrixname, groupname)
```

```
attributes(grid1)
#> $names
#> [1] "sA"   "xy"   "dim"  "temp"
```

The output is stored (in a temporary folder) as a list composed of:

- `sA` the corresponding sparse matrix (in compressed sparse column format);

```
str(grid1$sA)
#> Formal class 'dsCMatrix' [package "Matrix"] with 7 slots
#>   ..@ i       : int [1:476] 173 174 176 70 71 74 74 75 77 77 ...
#>   ..@ p       : int [1:253] 0 3 6 9 12 15 18 21 24 27 ...
#>   ..@ Dim     : int [1:2] 252 252
#>   ..@ Dimnames:List of 2
#>   .. ..$ : NULL
#>   .. ..$ : NULL
#>   ..@ x       : num [1:476] 1 1 1 1 1 1 1 1 1 1 ...
#>   ..@ uplo    : chr "L"
#>   ..@ factors : list()
```

- possibly coordinates `xy` (stored in a `data.frame`);

```
head(grid1$xy, 3)
#>            x       y
#> [1,] 0.00000 0.00000
#> [2,] 2.88763 3.85355
#> [3,] 3.14645 4.11237
```

- `dim"` the numbers of rows, columns and numerically nonzero elements;

```
grid1$dim
#>   NumRows NumCols NonZeros
#> 1     252     252      476
```

- `temp` the path to the temporary directory where the matrix and downloaded files (including singular values if requested) are stored.

```r
list.files(grid1$temp)
```

Metadata associated with the matrix can be display via

```r
file.show(paste(grid1$temp,"grid1",sep=""))
```

or in the console:

```r
cat(readLines(paste(grid1$temp,"grid1",sep=""), n=14), sep = "\n")
```

`download_graph` function has an optional `svd` argument; setting `svd = "TRUE"` downloads a ".mat" file containing the singular values of the matrix, if available.

For further insights, the `get_graph_info` function retrieve detailed information about the matrix from the SuiteSparse Matrix Collection website. `get_graph_info` fetches the three tables with "MatrixInformation", "MatrixProperties," and "SVDStatistics", providing a comprehensive overview of the matrix (`rvest` package needs to be installed).

```r
matrix_info <- get_graph_info(matrixname, groupname)
matrix_info
```

The `download_graph` function also has an optional argument `add_info` which, when set to `TRUE`, automatically calls `get_graph_info` and appends the retrieved information to the output of `download_graph`. This makes it easy to get both the graph data and its associated information in a single function call.

```r
downloaded_graph <- download_graph(matrixname, groupname, add_info = TRUE)
downloaded_graph$info
```

Table 3: Matrix Information (left) and Matrix Properties (right).

	MatrixInformation
Name	grid1
Group	AG-Monien
Matrix ID	2416
Num Rows	252
Num Cols	252
Nonzeros	952
Pattern Entries	952
Kind	2D/3D Problem
Symmetric	Yes
Date	1998
Author	R. Diekmann, R. Preis
Editor	R. Diekmann, R. Preis

	SVDStatistics
Structural Rank	252
Structural Rank Full	true
Num Dmperm Blocks	2
Strongly Connect Components	1
Num Explicit Zeros	0
Pattern Symmetry	100%
Numeric Symmetry	100%
Cholesky Candidate	no
Positive Definite	no
Type	binary

The package also allows to plot a (planar) graph using the function `plot_graph`. It also contains a function to plot signals defined on top of the graph `plot_signal`.

```r
f <- rnorm(nrow(grid1$sA))
plot_graph(grid1)
plot_signal(grid1, f, size = 2)
```

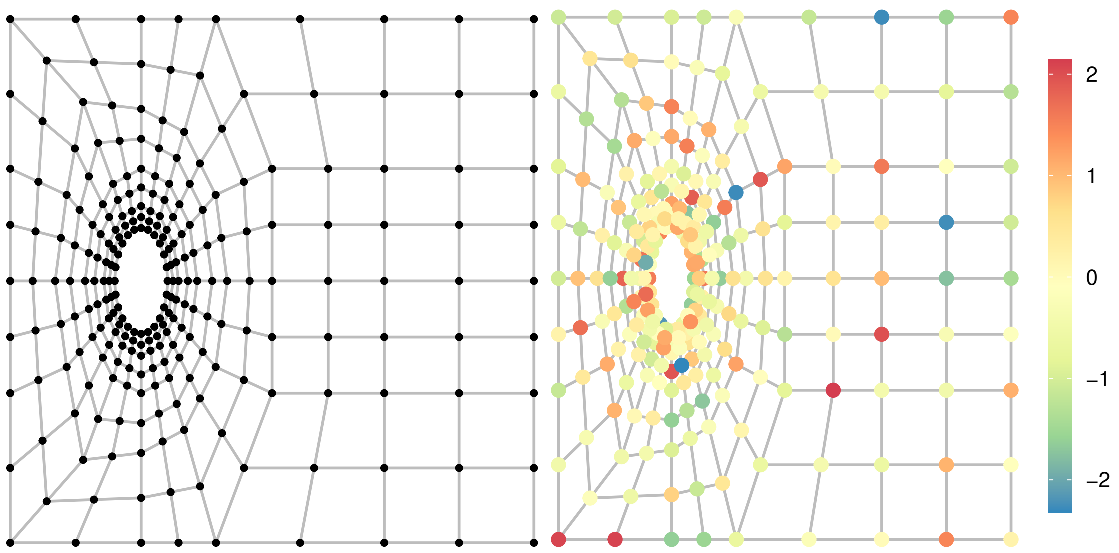

Figure 1: Graph (left) and graph signal (right).

In cases where these coordinates are not available, `plot_graph` employs simple spectral graph drawing to calculate some node coordinates. This is done using the function `spectral_coords`, which computes the spectral coordinates based on the eigenvectors associated with the two smallest non-zero eigenvalues of the graph's Laplacian Hall (1970).

3 A Short Introduction to Graph Signal Processing

Graph theory provides a robust mathematical framework for representing complex systems. In this context, entities are modeled as vertices (or nodes) and their interconnections as edges, encapsulating a broad spectrum of real-world phenomena from social and communication networks to molecular structures and brain connectivity patterns.

Among the diverse types of graphs, such as undirected, directed, weighted, bipartite, and multigraphs, each offers distinct analytical advantages tailored to specific contexts. This vignette is devoted to undirected, connected, graphs where edges link two vertices symmetrically, often with weighted values to express connection strength or intensity. For instance, in a road network graph, the weights might correspond to the length of each road segment.

Graphs are defined by $G = (V, E)$, where V denotes the set of vertices or nodes and E represents the set of edges. Each edge $(i, j) \in E$ connects nodes i and j, potentially with an associated weight w_{ij}. The connectivity and interaction structure of G is encoded in the adjacency matrix W, where $w_{ij} = w_{ji}$ for $i, j \in V$. The size of the graph is the number of nodes $n = |V|$. The degree matrix D is a diagonal matrix with $D_{ii} = \sum_{j \in V} w_{ij}$. These matrices, W and D, serve as the foundation for analyzing signal behavior on graph structures in GSP.

The spectral properties of the Laplacian matrices offer deep insights into the structure of graphs.

The unnormalized Laplacian, $\mathcal{L} = D - W$, has non-negative eigenvalues, with the smallest being zero, indicating the number of connected components in the graph. This number can be retrieved from the "MatrixProperty" dataframe using the `get_graph_info` function, if the graph has been downloaded with `download_graph`, or computed using Depth-First Search algorithm, using **igraph** R package for instance Csárdi et al. (2023).

```
grid1$info$MatrixProp["Strongly Connect Components",]
#> [1] "1"
```

On the other hand, the normalized Laplacian, $\mathcal{L}_{\text{norm}} = I - D^{-1/2}WD^{-1/2}$, has a spectrum that typically lies between 0 and 2. The zero eigenvalue corresponds to the number of connected components, and the first non-zero smallest eigenvalue, often referred to as the spectral gap, plays a crucial role in determining the graph's propensity for clustering. The larger this gap, the more pronounced the cluster structures within the graph. By scaling the eigenvalues according to node degrees, the normalized Laplacian accentuates the separation between clusters, making the spectral gap a significant measure in spectral graph clustering algorithms. However, a large spectral gap might present challenges for fast spectral filtering, especially depending on the approximation methods used, where it could lead to issues in convergence or computational efficiency.

Lastly, the random walk Laplacian, $\mathcal{L}_{\text{rw}} = I - D^{-1}W$, is generally better suited for directed graphs and scenarios involving random walk dynamics. Its spectrum is also non-negative. The choice of which Laplacian to use is dictated by the particular graph properties one aims to emphasize or analyze in a given application.

The `laplacian_mat` function (which supports both standard and sparse matrix representations) allows to compute those three forms of Laplacian matrices. For example, let $G = (V, E)$ a simple undirected graph with the vertex set $V = \{1, 2, 3\}$ and the edge set $E = \{\{1, 2\}, \{2, 3\}\}$. The corresponding adjacency matrix W, as well as its unnormalized, normalized, and random walk Laplacians, can be represented and calculated as follows:

```
W <- matrix(c(0, 1, 0,
              1, 0, 1,
              0, 1, 0), ncol=3)
laplacian_mat(W, "unnormalized")
#>      [,1] [,2] [,3]
#> [1,]    1   -1    0
#> [2,]   -1    2   -1
#> [3,]    0   -1    1
laplacian_mat(W, "normalized")
#>            [,1]       [,2]       [,3]
#> [1,]  1.0000000 -0.7071068  0.0000000
#> [2,] -0.7071068  1.0000000 -0.7071068
#> [3,]  0.0000000 -0.7071068  1.0000000
laplacian_mat(W, "randomwalk")
#>      [,1] [,2] [,3]
#> [1,]  1.0   -1  0.0
#> [2,] -0.5    1 -0.5
#> [3,]  0.0   -1  1.0
```

GSP extends classical signal processing concepts to signals defined on graphs. Let G be a graph on the vertex set $V = \{v_1, \dots, v_n\}$. A graph signal on G is a function $f : V \to \mathbb{R}$, that assigns a value to each node of the graph. It can be represented as a vector $(f(v_1), f(v_2), \dots, f(v_n))^\top \in \mathbb{R}^n$, where each entry f_i corresponds to the signal value at node i.

The Laplacian quadratic form $f^\top \mathcal{L} f$ gives a measure of a graph signal's smoothness:

$$f^\top \mathcal{L} f = \frac{1}{2} \sum_{(i,j)\in E} w_{ij}(f_i - f_j)^2,$$

where a lower value suggests that the signal varies little between connected nodes, and thus is smoother on the graph. It's a global measure of the graph's "frequency" content which is insightful for understanding the overall variation in graph signals. The `smoothmodulus` function calculates this form for a given graph signal, returning a scalar value that quantifies the signal's smoothness in relation to the graph's structure. Moreover, the `randsignal` function can be used to generate graph signals with varying smoothness properties.

To analyze graph signals, the concept of the Graph Fourier Transform (GFT) is fundamental. The GFT provides a means to represent graph signals in the frequency domain, analogous to the classical Fourier Transform for traditional signals. Given a graph G, a GFT can be defined as the representation of signals on an orthonormal basis for $\mathbb{R}^n$ consisting of eigenvectors of the graph shift operator. The choice of graph shift operator is essential, as it determines the basis for the GFT, it can be either the Laplacian matrix or the adjacency matrix. In this tutorial, we primarily focus on signal processing using the Laplacian matrix as the shift operator.

For undirected graphs, the Laplacian matrix $\mathcal{L}$ is symmetric and positive semi-definite, with non-negative real eigenvalues. Given the eigenvalue decomposition of the graph Laplacian $\mathcal{L} = U\Lambda U^T$, where U is the matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues, the GFT of a signal f is given by $\hat{f} = U^T f$. Here, $\hat{f}$ represents the graph signal in the frequency domain. The elements of $\hat{f}$ are the coefficients of the signal f with respect to the eigenvectors of $\mathcal{L}$, which can be interpreted as the frequency components of the signal on the graph.

The inverse GFT is given by $f = U\hat{f}$. This allows for the reconstruction of the graph signal in the vertex domain from its frequency representation. The GFT provides a powerful tool for analyzing and processing signals on graphs. It enables the identification of signal components that vary smoothly or abruptly over the graph, facilitating tasks such as filtering, denoising, and compression of graph signals. The function `forward_gft` allows to perform a GFT decomposition and to obtain the associated Fourier coefficients. The function `inverse_gft` allows to make the reconstruction.

4 Data-Driven Graph Signal Denoising

We give an example of an application in the case of the denoising of a noisy signal f defined on a graph G with set of vertices V. More precisely, the (unnormalized) graph Laplacian matrix $\mathcal{L} \in \mathbb{R}^{V\times V}$ associated with G is the symmetric matrix defined as $\mathcal{L} = D - W$, where W is the matrix of weights with coefficients $(w_{ij})_{i,j\in V}$, and D the diagonal matrix with diagonal coefficients $D_{ii} = \sum_{j\in V} w_{ij}$. A signal f on the graph G is a function $f : V \to \mathbb{R}$.

The degradation model can be written as

$$\tilde{f} = f + \xi,$$

where $\xi \sim \mathcal{N}(0, \sigma^2)$. The purpose of denoising is to build an estimator of f that depends only on $\tilde{f}$.

A simple way to construct an effective non-linear estimator is obtained by thresholding the SGWT coefficients of f on a frame (see Hammond et al. (2011) for details about the SGWT).

A general thresholding operator τ with threshold parameter $t \geq 0$ applied to some signal f is defined as

$$\tau(x,t) = x \max\{1 - t^{\beta}|x|^{-\beta}, 0\}, \tag{1}$$

with $\beta \geq 1$. The most popular choices are the soft thresholding ($\beta = 1$), the James-Stein thresholding ($\beta = 2$) and the hard thresholding ($\beta = \infty$).

Given the Laplacian and a given frame, denoising in this framework can be summarized as follows:

- Analysis: compute the SGWT transform $\mathcal{W}\tilde{f}$;
- Thresholding: apply a given thresholding operator to the coefficients $\mathcal{W}\tilde{f}$;
- Synthesis: apply the inverse SGWT transform to obtain an estimation of the original signal.

Each of these steps can be performed via one of the functions `analysis`, `synthesis`, `beta_thresh`. Laplacian is given by the function `laplacian_mat`. The `tight_frame` function allows the construction of a tight frame based on Göbel et al. (2018) and Coulhon et al. (2012). In order to select a threshold value, we consider the method developed in de Loynes et al. (2021) which consists in determining the threshold that minimizes the Stein unbiased risk estimator (SURE) in a graph setting (see de Loynes et al. (2021) for more details).

We give an illustrative example on the `grid1` graph from the previous section. We start by calculating, the Laplacian matrix (from the adjacency matrix), its eigendecomposition and the frame coefficients.

```r
A <- grid1$sA
L <- laplacian_mat(A)
val1 <- eigensort(L)
evalues <- val1$evalues
evectors <- val1$evectors
#- largest eigenvalue
lmax <- max(evalues)
#- parameter that controls the scale number
b <- 2
tf <- tight_frame(evalues, evectors, b=b)
```

Wavelet frames can be seen as special filter banks. The tight-frame considered here is a finite collection $(\psi_j)_{j=0,\ldots,J}$ forming a finite partition of unity on the compact $[0, \lambda_1]$, where λ_1 is the largest eigenvalue of the Laplacian spectrum $\mathrm{sp}(\mathcal{L})$. This partition is defined as follows: let $\omega : \mathbb{R}^+ \to [0,1]$ be some function with support in $[0,1]$, satisfying $\omega \equiv 1$ on $[0, b^{-1}]$, for some $b > 1$, and set

$$\psi_0(x) = \omega(x) \quad \text{and} \quad \psi_j(x) = \omega(b^{-j}x) - \omega(b^{-j+1}x) \quad \text{for} \quad j = 1, \ldots, J, \quad \text{where} \quad J = \left\lfloor \frac{\log \lambda_1}{\log b} \right\rfloor + 2.$$

Thanks to Parseval's identity, the following set of vectors is a tight frame:

$$\mathfrak{F} = \left\{ \sqrt{\psi_j}(\mathcal{L})\delta_i, j = 0, \ldots, J, i \in V \right\}.$$

The `plot_filter` function allows to represent the elements $\sqrt{\psi_j}$ (filters) of this partition, with

$$\omega(x) = \begin{cases} 1 & \text{if } x \in [0, b^{-1}] \\ b \cdot \frac{x}{1-b} + \frac{b}{b-1} & \text{if } x \in (b^{-1}, 1] \\ 0 & \text{if } x > 1 \end{cases}$$

which corresponds to the tigth-frame constructed from the `zetav` function.

```
plot_filter(lmax,b)
```

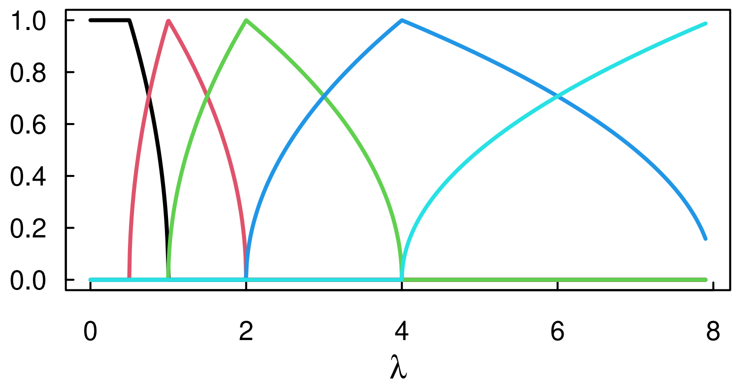

Figure 2: Plot of the spectral graph filters on the spectrum of grid1 graph.

The SGWT of a signal $f \in \mathbb{R}^V$ is given by

$$\mathcal{W}f = \left(\sqrt{\psi_0}(\mathcal{L})f^T, \ldots, \sqrt{\psi_J}(\mathcal{L})f^T\right)^T \in \mathbb{R}^{n(J+1)}.$$

The adjoint linear transformation $\mathcal{W}^*$ of $\mathcal{W}$ is:

$$\mathcal{W}^*\left(\eta_0^T, \eta_1^T, \ldots, \eta_J^T\right)^T = \sum_{j \geq 0} \sqrt{\psi_j}(\mathcal{L})\eta_j.$$

The tightness of the underlying frame implies that $\mathcal{W}^*\mathcal{W} = \mathrm{Id}_{\mathbb{R}^V}$ so that a signal $f \in \mathbb{R}^V$ can be recovered by applying $\mathcal{W}^*$ to its wavelet coefficients $((\mathcal{W}f)_i)_{i=1,\ldots,n(J+1)} \in \mathbb{R}^{n(J+1)}$.

Then, noisy observations $\tilde{f}$ are generated from a random signal f.

```
n <- nrow(L)
f <- randsignal(0.01, 3, A)
sigma <- 0.01
noise <- rnorm(n, sd = sigma)
tilde_f <- f + noise
```

Below is a graphical representation of the original signal and its noisy version.

```
plot_signal(grid1, f, size = 2)
plot_signal(grid1, tilde_f, size = 2)
```

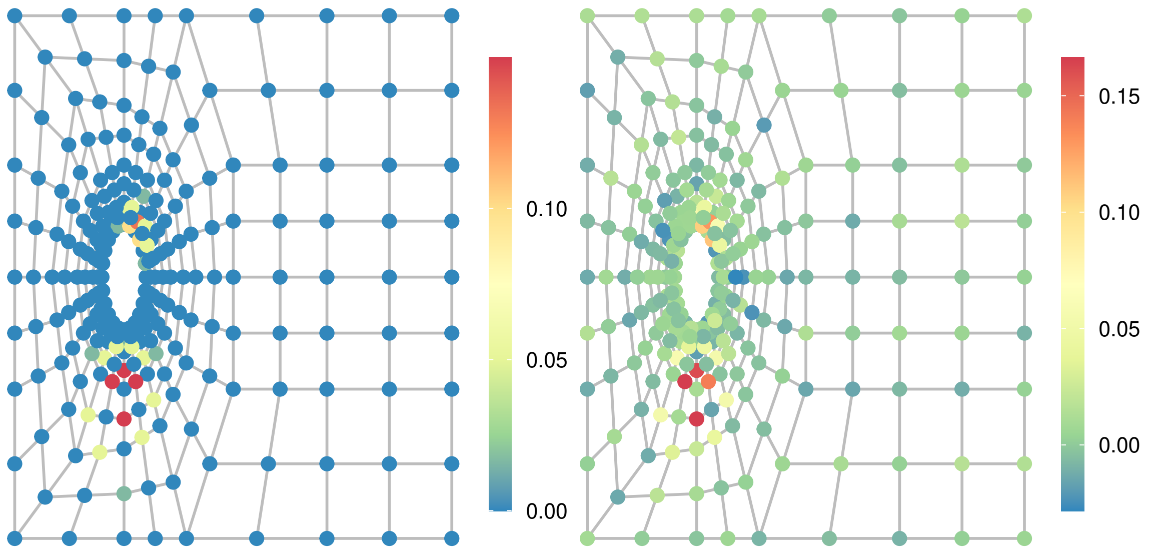

Figure 3: Original graph signal (left) and noisy observations (right).

We compute the SGWT transforms $\mathcal{W}\tilde{f}$ and $\mathcal{W}f$.

```
wcn <- analysis(tilde_f,tf)
wcf <- analysis(f,tf)
```

An alternative to avoid frame calculation is given by the `forward_sgwt` function which provides a forward SGWT. For example:

```
wcf <- forward_sgwt(f, evalues, evectors, b=b)
```

The optimal threshold is then determined by minimizing the SURE (using Donoho and Johnstone's trick Donoho and Johnstone (1995) which remains valid here, see de Loynes et al. (2021)). More precisely, the SURE for a general thresholding process h is given by the following identity

$$\mathbf{SURE}(h) = -n\sigma^2 + \|h(\widetilde{F}) - \widetilde{F}\|^2 + 2\sum_{i,j=1}^{n(J+1)} \gamma_{i,j}^2 \partial_j h_i(\widetilde{F}), \tag{2}$$

where $\gamma_{i,j}^2 = \sigma^2(\mathcal{W}\mathcal{W}^*)_{i,j}$ that can be computed from the frame (or estimated via Monte-Carlo simulation). The `SURE_thresh`/`SURE_MSEthresh` allow to evaluate the SURE (in a global fashion) considering the general thresholding operator τ (1). These functions provide two different ways of applying the threshold, "uniform" and "dependent" (*i.e.*, the same threshold for each coefficient vs a threshold normalized by the variance of each coefficient). The second approach generally provides better results (especially when the weights have been calculated via the frame). A comparative example of these two approaches is given below (with $\beta = 2$ James-Stein attenuation threshold).

```
diagWWt <- colSums(t(tf)^2)
thresh <- sort(abs(wcn))
opt_thresh_d <- SURE_MSEthresh(wcn,
                               wcf,
                               thresh,
                               diagWWt,
                               beta=2,
                               sigma,
                               NA,
                               policy = "dependent",
                               keepwc = TRUE)

opt_thresh_u <- SURE_MSEthresh(wcn,
                               wcf,
                               thresh,
                               diagWWt,
                               beta=2,
                               sigma,
                               NA,
                               policy = "uniform",
                               keepwc = TRUE)
```

We can plot MSE risks and their SUREs estimates as a function of the threshold parameter (assuming that σ is known).

```
plot(thresh, opt_thresh_u$res$MSE,
     type="l", xlab = "t", ylab = "risk", log="x")
lines(thresh, opt_thresh_u$res$SURE-n*sigma^2, col="red")
lines(thresh, opt_thresh_d$res$MSE, lty=2)
lines(thresh, opt_thresh_d$res$SURE-n*sigma^2, col="red", lty=2)
legend("topleft", legend=c("MSE_u", "SURE_u",
                           "MSE_d", "SURE_d"),
       col=rep(c("black", "red"), 2),
       lty=c(1,1,2,2), cex = 1)
```

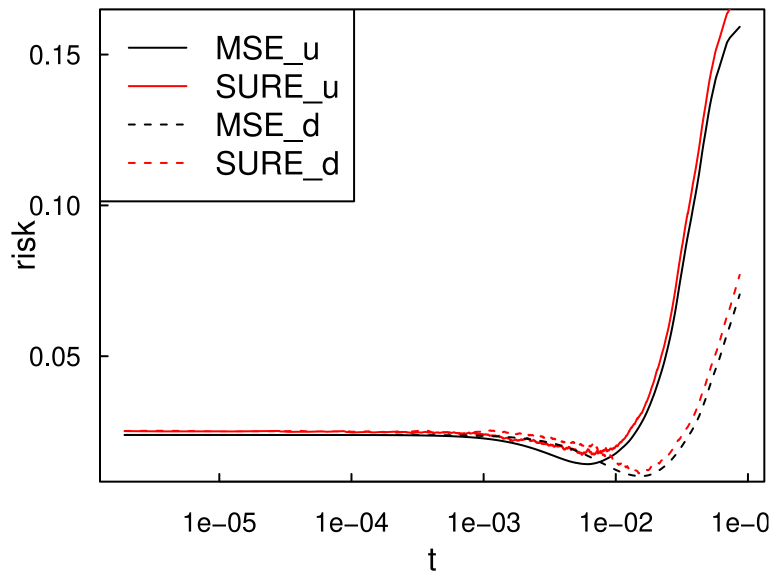

Figure 4: MSE risk and its SURE estimates as a function of the threshold parameter.

Finally, the synthesis allows us to determine the resulting estimators of f, *i.e.*, the ones that minimize the unknown MSE risks and the ones that minimizes the SUREs.

```
wc_oracle_u <- opt_thresh_u$wc[, opt_thresh_u$min["xminMSE"]]
wc_oracle_d <- opt_thresh_d$wc[, opt_thresh_d$min["xminMSE"]]
wc_SURE_u <- opt_thresh_u$wc[, opt_thresh_u$min["xminSURE"]]
wc_SURE_d <- opt_thresh_d$wc[, opt_thresh_d$min["xminSURE"]]

hatf_oracle_u <- synthesis(wc_oracle_u, tf)
hatf_oracle_d <- synthesis(wc_oracle_d, tf)
hatf_SURE_u  <- synthesis(wc_SURE_u, tf)
hatf_SURE_d  <- synthesis(wc_SURE_d, tf)

res <- data.frame("Input_SNR"=round(SNR(f,tilde_f),2),
                  "MSE_u"=round(SNR(f,hatf_oracle_u),2),
                  "SURE_u"=round(SNR(f,hatf_SURE_u),2),
                  "MSE_d"=round(SNR(f,hatf_oracle_d),2),
                  "SURE_d"=round(SNR(f,hatf_SURE_d),2))
```

Table 4: Comparison of SNR performance between uniform and dependent policies.

Input_SNR	MSE_u	SURE_u	MSE_d	SURE_d
8.24	12.55	12.15	14.38	14.38

It can be seen from Table 4 that in both cases, SURE provides a good estimator of the MSE and therefore the resulting estimators have performances close (in terms of SNR) to those obtained by minimizing the unknown risk.

Equivalently, estimators can be obtained by the inverse of the SGWT given by the function `inverse_sgwt`. For exemple:

```
hatf_oracle_u <- inverse_sgwt(wc_oracle_u,
                              evalues, evectors, b)
```

Or if the coefficients have not been stored for each threshold value (*i.e.*, with the argument "keepwc=FALSE" when calling `SUREthresh`) using the thresholding function `beta_thresh`, *e.g.*,

```
wc_oracle_u <- betathresh(wcn,
                          thresh[opt_thresh_u$min[[1]]], 2)
```

Notably, SURE can also be applied in a level-dependent manner using `SUREthresh` at each scale (the output of `SUREthresh` can be retrieve with the argument "keepSURE = TRUE" at the function call).

```
J <- floor(log(lmax)/log(b)) + 2
LD_opt_thresh_u <- LD_SUREthresh(J=J,
                                 wcn=wcn,
                                 diagWWt=diagWWt,
                                 beta=2,
                               sigma=sigma,
                               hatsigma=NA,
                               policy = "uniform",
                               keepSURE = FALSE)
hatf_LD_SURE_u <- synthesis(LD_opt_thresh_u$wcLDSURE, tf)
print(paste0("LD_SURE_u = ",round(SNR(f,hatf_LD_SURE_u),2),"dB"))
#> [1] "LD_SURE_u = 13.1dB"
```

Even though the SURE no longer depends on the original signal, it does depend on σ^2, two naive (biased) estimators are obtained via `GVN` or `HPVN` functions (see de Loynes et al. (2021) for more details). Another possible improvement would be to use a scale-dependent variance estimator (especially in the case of "policy ="dependent" ").

Furthermore, the major limitations are the need to diagonalize the graph's Laplacian, and the calculation of the weights involved in the SURE (which requires an explicit calculation of the frame). To address the first limitation, several strategies have been proposed in the literature, notably via approximation by Chebyshev polynomials (see Hammond et al. (2011) or Shuman et al. (2018)). Combined with these approximations, a Monte Carlo method to estimate the SURE weights has been proposed in Chedemail et al. (2022), extending the applicability of SURE to large graphs.

References

Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2014). Spectral networks and locally connected networks on graphs. In *ICLR*.

Chedemail, E., de Loynes, B., Navarro, F., and Olivier, B. (2022). Large graph signal denoising with application to differential privacy. *IEEE Transactions on Signal and Information Processing over Networks*, 8:788–798.

Coulhon, T., Kerkyacharian, G., and Petrushev, P. (2012). Heat kernel generated frames in the setting of dirichlet spaces. *Journal of Fourier Analysis and Applications*, 18(5):995–1066.

Csárdi, G., Nepusz, T., Traag, V., Horvát, S., Zanini, F., Noom, D., and Müller, K. (2023). *igraph: Network Analysis and Visualization in R*. R package version 1.5.1.

Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1.

de Loynes, B., Navarro, F., and Olivier, B. (2021). Data-driven thresholding in denoising with spectral graph wavelet transform. *Journal of Computational and Applied Mathematics*, 389:113319.

Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852.

Donoho, D. L. and Johnstone, I. M. (1995). Adapting to unknown smoothness via wavelet shrinkage. *J. Amer. Statist. Assoc.*, 90(432):1200–1224.

Göbel, F., Blanchard, G., and von Luxburg, U. (2018). Construction of tight frames on graphs and application to denoising. In *Handbook of big data analytics*, Springer Handb. Comput. Stat., pages 503–522. Springer, Cham.

Hall, K. M. (1970). An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229.

Hamilton, W., Ying, Z., and Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034.

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

Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep convolutional networks on graph-structured data. In *NIPS*.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *ICLR*.

Kolodziej, S. P., Aznaveh, M., Bullock, M., David, J., Davis, T. A., Henderson, M., Hu, Y., and Sandstrom, R. (2019). The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244.

Ortega, A., Frossard, P., Kovačević, J., Moura, J. M., and Vandergheynst, P. (2018). Graph signal processing: Overview, challenges, and applications. *Proceedings of the IEEE*, 106(5):808–828.

Perraudin, N., Paratte, J., Shuman, D., Martin, L., Kalofolias, V., Vandergheynst, P., and Hammond, D. K. (2014). GSPBOX: A toolbox for signal processing on graphs. *ArXiv e-prints*.

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

Shuman, D. I., Vandergheynst, P., Kressner, D., and Frossard, P. (2018). Distributed signal processing via chebyshev polynomial approximation. *IEEE Transactions on Signal and Information Processing over Networks*, 4(4):736–751.

Tremblay, N. and Borgnat, P. (2014). Graph wavelets for multiscale community mining. *IEEE Trans. Signal Process.*, 62(20):5227–5239.