Gasper: GrAph Signal ProcEssing in R

Basile de Loynes, Fabien Navarro, Baptiste Olivier

2023-10-26

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 first version of the **gasper** package is available online[1]. In particular, it includes the methodology and codes[2] developed in de Loynes et al. [2021] and provides an interface to the SuiteSparse Matrix Collection Davis and Hu [2011].

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

[1] https://github.com/fabnavarro/gasper

[2] https://github.com/fabnavarro/SGWT-SURE

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 `download_graph` function allows to download a graph from this collection, based on the name of the graph 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 and

```
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.

```
list.files(grid1$temp)
#> [1] "grid1"           "grid1.mtx"        "grid1_coord.mtx"
```

Information about the matrix that can be display via `file.show(paste(grid1$temp,"grid1",sep=""))`

for example or in the console:

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

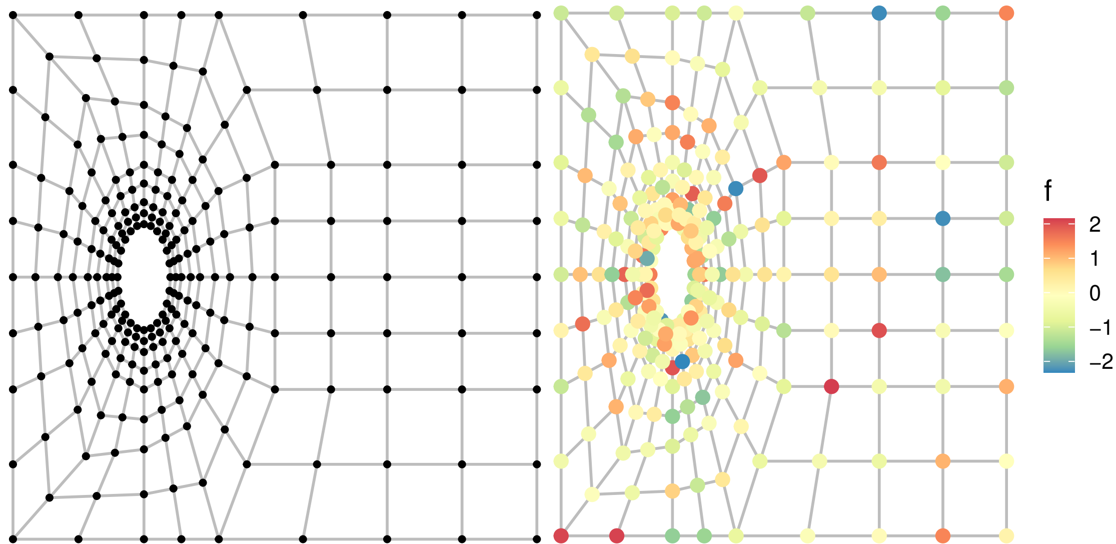
`download_graph` function has an optional svd argument; setting "svd = TRUE" downloads a ".mat" file containing the singular values of the matrix, if available.

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 "Matrix Information", "Matrix Properties," and "SVD Statistics", providing a comprehensive overview of the matrix (`rvest` package needs to be installed).

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

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)
```



3 Example of application to 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 $\hat{f}$ 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 $\hat{f}$ 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.

```
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 $(\psi_j)_{j=0,\dots,J}$ 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

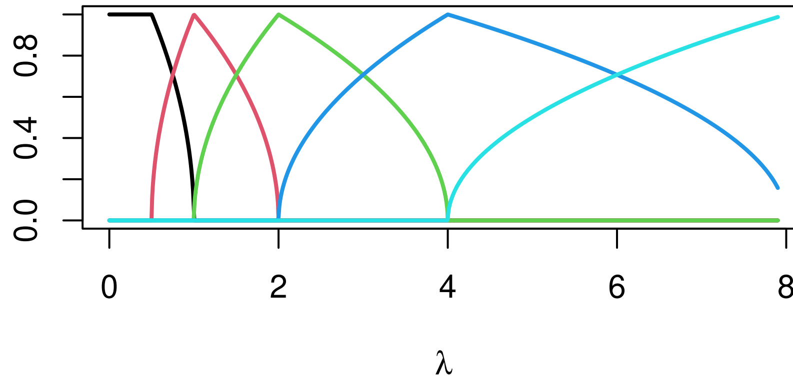
$$\psi_0(x) = \omega(x) \quad \text{and} \quad \psi_j(x) = \omega(b^{-j}x) - \omega(b^{-j+1}x) \quad \text{for} \quad j = 1, \dots, J, \quad \text{where} \quad J = \left\lfloor \frac{\log \lambda_1}{\log b} \right\rfloor + 2.$$

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

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

The `plot_filter` function allows to represent the elements (filters) of this partition.

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



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:

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

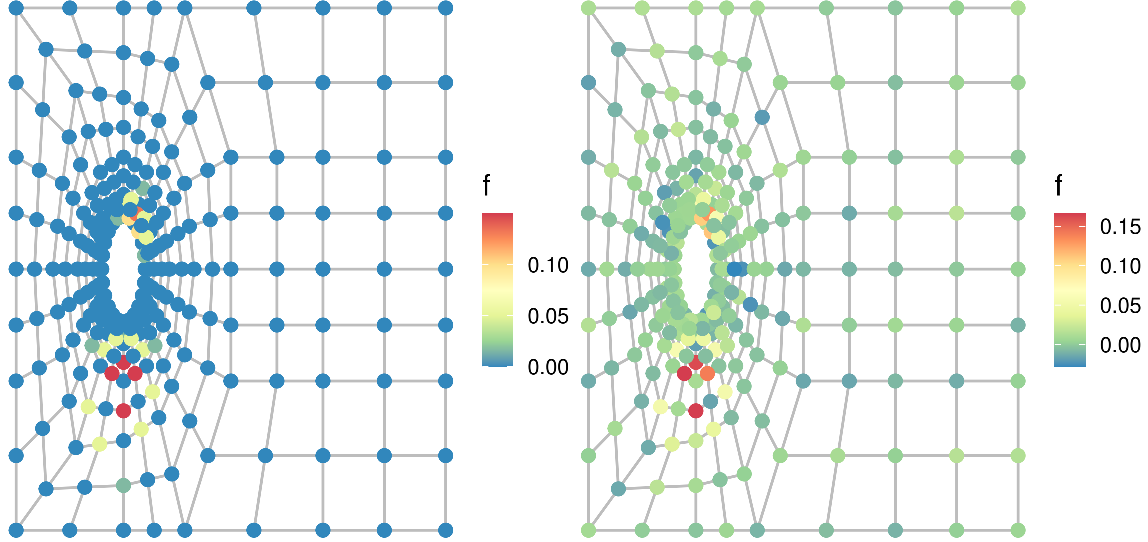
The tightness of the underlying frame implies that $\mathcal{W}^*\mathcal{W} = \mathrm{Id}_{\mathbb{R}^V}$ so that a signal $f \in \mathbb{R}^V$ can be recovered by applying $\mathcal{W}^*$ to its wavelet coefficients $((\mathcal{W}f)_i)_{i=1,\ldots,n(J+1)} \in \mathbb{R}^{n(J+1)}$.

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)
```



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 fast 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).

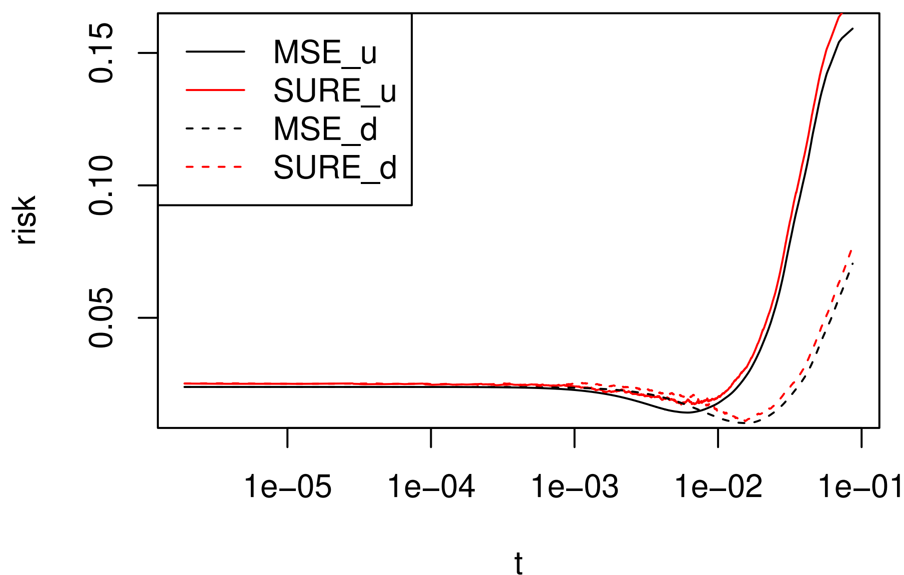
```
diagWWt <- colSums(t(tf)^2)
thresh <- sort(abs(wcn))
opt_thresh_d <- SURE_MSEthresh(wcn,
                           wcf,
                           thresh,
                           diagWWt,
                           beta=2,
                           sigma,
                           NA,
```

```
                              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)
```



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 1: Uniform vs Dependent

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 1 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_d <- LD_SUREthresh(J=J,
                                 wcn=wcn,
                                 diagWWt=diagWWt,
                                 beta=2,
                                 sigma=sigma,
                                 hatsigma=NA,
                                 policy = "uniform",
                                 keepSURE = FALSE)
hatf_LD_SURE_d <- synthesis(LD_opt_thresh_d$wcLDSURE, tf)
```

```r
print(paste0("LD_SURE_u = ",round(SNR(f,hatf_LD_SURE_d),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.

Bibliography

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014.

Elie Chedemail, Basile de Loynes, Fabien Navarro, and Baptiste Olivier. Large graph signal denoising with application to differential privacy. *IEEE Transactions on Signal and Information Processing over Networks*, 8:788–798, 2022.

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

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

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

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, pages 3844–3852, 2016.

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

Franziska Göbel, Gilles Blanchard, and Ulrike von Luxburg. 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, 2018.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

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

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. In *NIPS*, 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019. doi: 10.21105/joss.01244. URL https://doi.org/10.21105/joss.01244.

Antonio Ortega, Pascal Frossard, Jelena Kovačević, José MF Moura, and Pierre Vandergheynst. Graph signal processing: Overview, challenges, and applications. *Proceedings of the IEEE*, 106(5): 808–828, 2018.

Nathanaël Perraudin, Johan Paratte, David Shuman, Lionel Martin, Vassilis Kalofolias, Pierre Vandergheynst, and David K. Hammond. GSPBOX: A toolbox for signal processing on graphs. *ArXiv e-prints*, August 2014.

David Shuman, Sunil Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. 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, 2013.

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

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