Image Processing using Graph Laplacian Operator

David Wobrock
david.wobrock@gmail.com

March 2, 2018

Under the responsability of:

Supervisor: Frédéric Nataf
ALPINES Team - INRIA Paris

INSA Supervisor: Christine Solnon
Département Informatique
INSA de Lyon

Abstract

The latest image processing methods are based on global data-dependent filters. These methods involve huge affinity matrices and the graph Laplacian operator of the input image. As those matrices cannot fit in memory, they are approximated through sampling and using the Nyström extension based on the matrix's spectral decomposition. The inferred eigenvalue problem concerns the largest eigenvalues of the filter. However, they correspond to the smallest eigenvalues of the Laplacian operator. To solve this, we use the inverse iteration method, which has a better convergence rate, but involves solving linear systems. In this master thesis, we explore the behaviour of solving large and dense systems of linear equations using domain decomposition methods as preconditioner. The experiments show that...

Keywords image processing, graph Laplacian operator, eigenvalue problem, high performance computing

Résumé

Résumé français ici.

Mots-clés traitement d'images, Laplacien graphe, problème aux valeurs propres, calcul haute performance

1 Introduction

1.1 Background

The talk [1] and articles [2] [3] by Milanfar, working at Google Research, about using the graph Laplacian operator for nonconformist image processing purposes awakes curiosity.

Indeed, Milanfar reports that these techniques to build image filters are used on smartphones, which implies a reasonable execution time with limited computational resources. Over 2 billion photos are shared daily on social media [1], with very high resolutions and most of the time some processing or filter is applied to them. The algorithm must be efficient to be deployed at such scale.

1.2 Objective

The aim of this degree project is not to explore and improve the state of image processing. Instead, the spectral methods used in the algorithm will be our focus point. Those will inevitably expose eigenvalue problems, which may involve solving systems of linear equations.

Concerning the challenges about solving linear systems, on one hand, the size of the systems can be large considering high-resolution images with millions of pixels, or even considering 3D images. We process huge matrices of size N^2, with N the number of pixels of the input image. On the other hand, those huge affinity matrices are dense, thus the linear systems are dense. Often, linear systems result from discretising partial differential equations (PDEs) yielding sparse matrices, and therefore most linear solvers are specialised in sparse systems.

We want to explore the performance of linear solvers on dense problems, their stability and convergence. This will include preconditioning the linear systems, especially using domain decomposition methods, and analyse their behaviour on dense systems.

2 Image processing using graph Laplacian operator

An image filter consists of a function which outputs one pixel, taking all pixels as input and applying weights to them. Let z_i being the output pixel, W_{ij} the weight, N the number of pixels in the image and y_j all input pixels. We compute an output pixel with:

$$z_i = \sum_{j=1}^{N} W_{ij} y_j,$$

This means that a vector of weights exists for each pixel.

As a practical notation, we can say that, with W the matrix of weights and y and z respectively the input and output images as vectors,

$$z = Wy.$$

The filter matrix W considered here is data-dependent and built upon the input image y. A more mathematical notation would consider W as a nonlinear function of the input image such as $z = W(y)y$.

Image as graph Let's think of an image as a graph. Each pixel is a node and has edges to other nodes. The simplest way to connect pixels to other pixels is their direct neighbours, in which case each node has four edges. To avoid losing any information, we will instead consider the case of a complete graph; each node connects to all other nodes.

To preserve the image information in the graph, the graph edges will be assigned a weight, measuring the similarity[1] between the two nodes, thus between two pixels.

There are multiple ways the similarity can be defined. The most intuitive definition considers spatial proximity. This means that similar pixels are spatially close, which, translated to a basic filter, is the same as a Gaussian filter which computes a weighted average of the pixel's neighbourhood and produces what is known as Gaussian blur. Another similarity definition is to consider the pixel's colour. A good compromise is to consider an average of both, spatial and colour closeness, with a certain weighting.

Once the similarity is defined, we can compute the adjacency matrix of the graph including the edge weights. We will call this matrix the affinity matrix[2] K which represents the similarity of each pixel to every other pixel in the image. Consequently, this matrix is symmetric and of size $N \times N$ with N the number of pixels in the image. Also, most similarity functions define bounds on the values of K such as $0 \leq K_{ij} \leq 1$.

Using this affinity matrix, we obtain the graph Laplacian $\mathcal{L}$, used to build the filter matrix W.

Building the filter Multiple graph Laplacian definitions, more or less equivalent, exist and can have slightly different properties. In the case of image smoothing, the filter W is roughly defined such as $\mathcal{L} = I - W$ [1] and so $W = I - \mathcal{L}$. To get various filters using one Laplacian, we can apply some function f to $\mathcal{L}$ and obtain $W = I - f(\mathcal{L})$ which gives us more possibilities on the filter computation. Below the global filter algorithm if we compute the entire matrices:

[1] Also called affinity.

[2] Or similarity matrix, or kernel matrix

Algorithm 1 Image processing using entire graph Laplacian operator

Input: y an image of size N, f the function applied to $\mathcal{L}$
Output: z the output image
  Compute K (size $N \times N$)
  Compute Laplacian matrix $\mathcal{L}$
  $z \leftarrow (I - f(\mathcal{L}))y$

However, all these matrices K, $\mathcal{L}$ and W represent huge computational costs. Only storing one of these matrices is already a challenge since they have a size of N^2.

For example, a tiny test image of size 256×256 has 65 536 pixels, so one of these matrices has approximately 4.29×10^9 elements. Considering storing those with a 64 bits type, one matrix takes more than 34 GB of memory. Scaling to a modern smartphone picture, taken with a 10 MPixel camera, a matrix contains 10^{14} elements, meaning 800 TB of memory for each matrix.

Approximation by sampling and Nyström extension To avoid storing any of those huge matrices, approximation will be necessary. Following [4], we define the Nyström extension. It starts by sampling the image and only select a subset of p pixels, with $p \ll N$. Numerically, p should represent around 1% or less of the image pixels. The rows and columns of a matrix K are reorganised such as K_A the upper left affinity matrix of K of size $p \times p$, measuring the affinities between the sampled pixels. The submatrix K_B is holding the similarities between the sampled pixels and the remaining pixels and is of size $p \times (N - p)$. And the lower right submatrix K_C contains the affinities between the remaining pixels. We have:

$$K = \begin{bmatrix} K_A & K_B \\ K_B^T & K_C \end{bmatrix}.$$

Knowing that K_C is of size $(N - p) \times (N - p)$ and that $p \ll N$, this submatrix is still huge and must be avoided.

To have a numerical approximation of a symmetric (semi) positive definite matrix K, we use the eigendecomposition with Φ the orthonormal eigenvectors of K stored as a matrix and Π the eigenvalues of K:

$$K = \Phi \Pi \Phi^T.$$

The article [4] suggests the Nyström extension to approximate K by $\tilde{K} = \tilde{\Phi}\tilde{\Pi}\tilde{\Phi^T}$, using the eigendecomposition of the submatrix $K_A = \Phi_A \Pi_A \Phi_A^T$, with $\tilde{\Pi} = \Pi_A$ and the approximated leading eigenvectors $\tilde{\Phi}$:

$$\begin{aligned} \tilde{\Phi} &= \begin{bmatrix} \Phi_A \\ K_B^T K_A^{-1} \Phi_A \end{bmatrix} \\ &= \begin{bmatrix} \Phi_A \\ K_B^T \Phi_A \Pi_A^{-1} \end{bmatrix} \end{aligned} \tag{1}$$

We can calculate

$$\begin{aligned}\tilde{K} &= \tilde{\Phi}\tilde{\Pi}\tilde{\Phi^T} \\ &= \begin{bmatrix} \Phi_A \\ K_B^T \Phi_A \Pi_A^{-1} \end{bmatrix} \Pi_A \begin{bmatrix} \Phi_A^T & \Pi_A^{-1} \Phi_A^T K_B \end{bmatrix} \\ &= \begin{bmatrix} \Phi_A \Pi_A \\ K_B^T \Phi_A \end{bmatrix} \begin{bmatrix} \Phi_A^T & \Pi_A^{-1} \Phi_A^T K_B \end{bmatrix} \\ &= \begin{bmatrix} K_A & K_B \\ K_B^T & K_B^T K_A^{-1} K_B \end{bmatrix}\end{aligned} \tag{2}$$

We can clearly see that the huge submatrix K_C is now approximated by $K_B^T K_A^{-1} K_B$. The quality of the approximation is measurable by the norm of the difference of the two above terms. We recognise the norm of the Schur complement $\|K_C - K_B^T K_A^{-1} K_B\|$.

Eigendecomposition We need to compute the largest eigenvalues of the filter W. The reason can be found by formulating the filter by its diagonalisation $W = \sum_i^N \lambda_i \phi_i \phi_i^T$. When the ith eigenvalue λ_i is small, the eigenvector product will be negligible, and therefore the largest eigenvalues of the filter W are the most relevant.

For the approximation, we will actually need to compute the largest eigenvalues of the sampled pixels submatrix W_A to use the Nyström extension. Using a sample to compute the largest eigenvalues works to approximate the complete filter because these eigenvalues are decaying rapidly as shows the figure below:

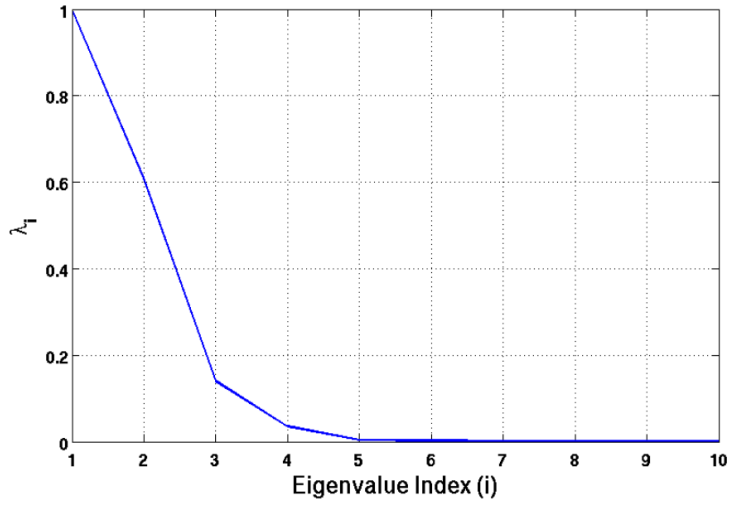


Figure 1 – Largest eigenvalues of an image filter, taken from [1].

We know that the eigenvalues of submatrix W_A are between the extremum eigenvalues of the filter W, which is known as the interlacing property of principal submatrices. We also know that computing the largest eigenvalues of the submatrix filter is equivalent to computing the smallest eigenvalues of the corresponding Laplacian operator.

The goal of this observation is the way of computing the eigenvalues. For the largest eigenvalues, the most famous algorithm is the power method. For the smallest eigenvalues, the inverse power method is a usual choice. Both methods converge faster when two successive eigenvalues are far from each other. In our case, the largest eigenvalues are close to each other; hence the inverse of these eigenvalues will be far from each other. We will therefore prefer the inverse power method.

The algorithm will, in an iterative manner, compute the associated eigenvector of an eigenvalue. This requires either to invert a matrix such as $x_{k+1} = A^{-1}x_k$, or to solve the linear system $Ax_{k+1} = x_k$. We will solve systems of linear equations to compute the first eigenvalues of the Laplacian in order to observe the behaviour of solvers on these dense matrices.

The main drawback of the Nyström method for us, is that it approximates the leading eigenvalues but we compute the trailing ones of the Laplacian $\mathcal{L}$ [5]. It is possible to obtain the eigendecomposition of the filter W, even when W is indefinite, through a method proposed by [4]. It consists of computing the inverse square root matrix $W_A^{-1/2}$, which could be done either by the complete eigendecomposition or by using Cauchy's integral formula. After this step, two more diagonalisation of matrices are required, demanding an important computation time.

Nevertheless, as stated in the objectives of the project, our main goal is not the image processing aspect, but the behaviour of linear solvers of these dense matrices using domain decomposition methods. We will therefore stick to computing the smallest eigenvalues of the Laplacian operator $\mathcal{L}$ and avoid spending too much time on the end of the algorithm implementation.

As the size of the image grows, computing the first p eigenvalues can easily represent computing more than thousand eigenvalues since we consider 1% of all pixels for the sampling step. Instead, we define m, with $m \leq p$, the number of computed eigenvalues of $\mathcal{L}_A$. It is not required to compute all eigenvalues of the sampled pixels matrix, only the first ones are the most relevant, since they correspond to the largest eigenvalues of the filter.

Output image Even if we cannot approximate the trailing eigenvectors of $\mathcal{L}$ through the eigenvectors of $\mathcal{L}_A$, we still define how the Laplacian is used to compute the output image. The summary [6] implicitly defines the filter as $W = I - f(\mathcal{L})$ with the function f that helps achieving various filters. To apply the function efficiently to the Laplacian operator, we apply it to the diagonal eigenvalue matrix such as $f(\mathcal{L}) = \Phi f(\Pi)\Phi^T$. The output image using the filter

approximation $\tilde{W}$ can be expressed as:

$$\begin{aligned} \tilde{z} &= \tilde{W}y \\ &= (I - f(\tilde{\mathcal{L}}))y \\ &= (I - \tilde{\Phi} f(\tilde{\Pi})\tilde{\Phi^T})y \\ &= y - \tilde{\Phi} f(\tilde{\Pi})\tilde{\Phi^T} y \end{aligned} \tag{3}$$

Algorithm 2 Image processing using approximated graph Laplacian operator

Input: y an image of size N, f the function applied to $\mathcal{L}$
Output: $\tilde{z}$ the output image by the approximated filter
{Sampling}
Sample p pixels, $p \ll N$
{Kernel matrix approximation}
Compute K_A (size $p \times p$) and K_B (size $p \times (N-p)$)
Compute the Laplacian submatrices $\mathcal{L}_A$ and $\mathcal{L}_B$
{Eigendecomposition}
Compute the m smallest eigenvalues Π_A and the associated eigenvectors Φ_A of $\mathcal{L}_A$
{Nyström extension and compute the filter}
See methods of solution proposed by [4]
$\tilde{z} \leftarrow \tilde{W}y$

3 Implementation

3.1 Parallel implementation

To scale our algorithm to use usual camera pictures, but also much larger inputs, we implemented it in a parallel manner using the C language and the Portable, Extensible Toolkit for Scientific Computation (PETSc) [7]. This library is built upon MPI and contains distributed data structures and parallel scientific computation routines. The most useful are the matrix and vector data structures and the parallel matrix-matrix and matrix-vector products. Additionally, PETSc provides Krylov subspace methods and preconditioners for solving linear systems, also implemented in a scalable and parallel manner. In a nutshell, PETSc provides an impressive parallel linear algebra toolkit which is very useful to shorten the development time. As we are basically using MPI, the main parallelism technique that we apply is SPMD. It is possible to activate some SIMD parallelism with PETSc but we do not consider it in our case. We want to point out to the reader that the distributed PETSc matrix data structure splits the data without overlap in a row-wise distribution manner.

In order to verify the correctness of our implementation, we used the Scalable Library for Eigenvalue Problem Computation (SLEPc) [8], which is based on

PETSc and provides parallel eigenvalue problem solvers. Furthermore, we need the library Elemental [9] in order to achieve dense matrix operations in PETSc.

We present how we included parallelism in our algorithm step-by-step, starting with reading the image and sampling. Then follows the computation of the affinities of the sampled pixels. And we finish with the computation of the smallest eigenvalues using the inverse subspace iteration. The implementation associated to this project is open source and can be found on GitHub[3].

3.2 Inverse subspace method

The used algorithm to compute the smallest eigenvalues is the inverse subspace iteration inspired by [10]. With m the number of eigenvalues we will compute, p the sample size and $m \leq p$, we start the algorithm by selecting m random orthonormal independent vectors X_0 of size p.

The inverse iteration algorithm consists of outer and inner iterations, with k the index of the current outer iteration. The inner iteration consists of solving m linear systems, one for each vector of X_k that we approximate, such that $\forall i \in [1, m]$ and $X_k^{(i)}$ the ith vector of the subspace X_k:

$$AX_{k+1}^{(i)} = X_k^{(i)}.$$

The outer iteration consists of repeating this process and orthonormalising the new vectors X_{k+1} until convergence, meaning having a small enough residual norm. We define the residual R_k of X_k, at a certain iteration k, as

$$\begin{aligned} R_k &= AX_k - X_k X_k^T AX_k \\ &= (I - X_k X_k^T)AX_k. \end{aligned} \tag{4}$$

We implemented a parallel Gram-Schmidt orthonormalising routine, based on the classical sequential one. A summary of the inverse subspace iteration algorithm:

Algorithm 3 Inverse subspace iteration

Input: A the matrix of size $p \times p$, m the number of required eigenvalues, ε ¿ 0
Output: X_k the desired invariant subspace
  Initialise m random orthonormal vectors X_0 of size p
  For k=0, 1, 2, ...
  while $\|R_k\| > \varepsilon$ **do**
    for i=1 **to** m **do**
      Solve $AX_{k+1}^{(i)} = X_k^{(i)}$
    end for
    Orthonormalise X_{k+1}
  end while

[3] https://github.com/David-Wobrock/image-processing-graph-laplacian/

Solving the systems of linear equations is done using the Krylov type solvers and the preconditioners included in PETSc. As a standard approach, we use the Restricted Additive Schwarz (RAS) method as preconditioning method, without overlap and 2 domains per process. Each subdomain is solved using the GMRES method.

On each outer iteration, we must compute the residuals to see if we converged. This requires multiple matrix-matrix products and computing a norm, so communication cannot be avoided here.

3.3 Entire matrix computation

We start by showing the result of the computation using the full matrices. We limited ourselves to grayscale images for the beginning and for computing the entire matrices, we can only process small images. The image below contains 135 000 pixels, so each matrix need around 145 GB.

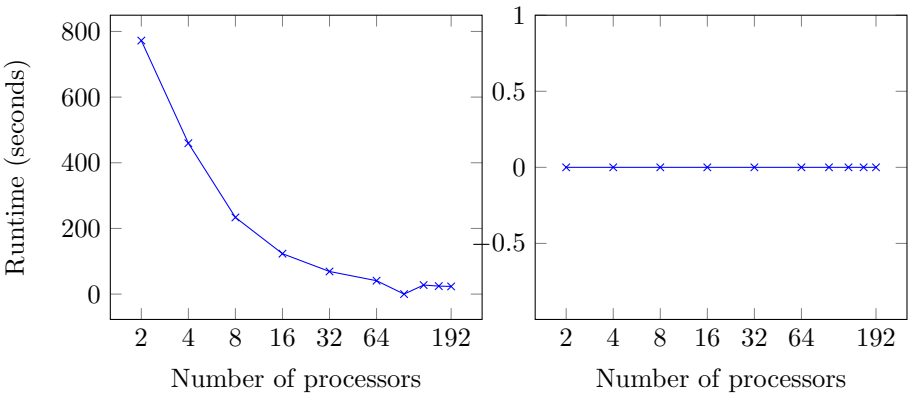
Figure 2 – Left: input image. Right: sharpened image.

The cat's fur on the left-hand side, the person's hand and the cushion on the right-hand side appear to be more detailed. We observe that the already sharp part of the image on the cat's head stays nice. We obtained this filter by defining $f(\mathcal{L}) = -3\,\mathcal{L}$ in the output image $z = (I - f(\mathcal{L}))y$.

This corresponds to the adaptive sharpening operator defined in [1] as $(I + \beta\,\mathcal{L})$ with $\beta > 0$. This approach remains a simple application of a scalar and doesn't require any eigenvalue computation. A more complete approach is called multiscale decomposition [3] and consists of applying a polynomial function to the Laplacian $\mathcal{L}$. We apply different coefficients to different eigenvalues of $\mathcal{L}$ because each eigenpair captures different features of the image.

3.4 Approximate matrix computation

The performances of the inverse subspace iteration for 50 and 500 eigenvalues:



(a) 50 eigenvalues.

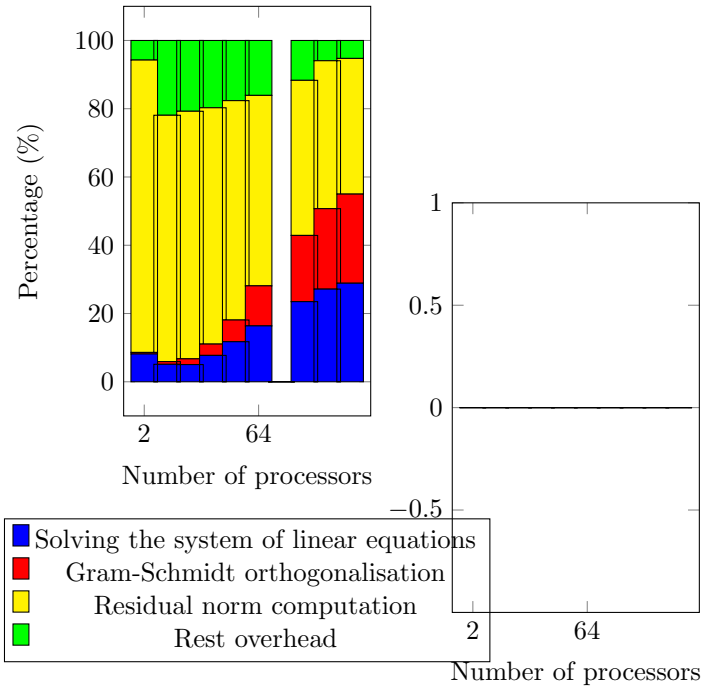
(b) 500 eigenvalues.

Figure 3 – Runtime of the inverse subspace iteration part of the algorithm.

When increasing a low number of processors, we see a sharp improvement of the performances in both cases. But the runtime stagnates quickly when the number of processors grows and we even observe a raise of the runtime. The algorithm reaches its parallelisation limit and the communication overhead takes over.

We know from the runtime performances that the inverse iteration part of the algorithm is not scaling correctly. For any amount of computed eigenvalues, when we increase the number of processes, the proportion of time spent computing the eigenvalues increases. For 500 eigenvalues and 128 processors, we spend more than 99% of the time computing the eigenvalues. This confirms that the algorithm does not quite scale yet.

We now have a look at the internal steps of the inverse power method to see where lies the problem. The algorithm consists of iteratively solving m linear systems, orthonormalising the vectors and computing the residual norm. Here is the proportion of each step of the inverse subspace interation for the computation of 50 and 500 eigenvalues:

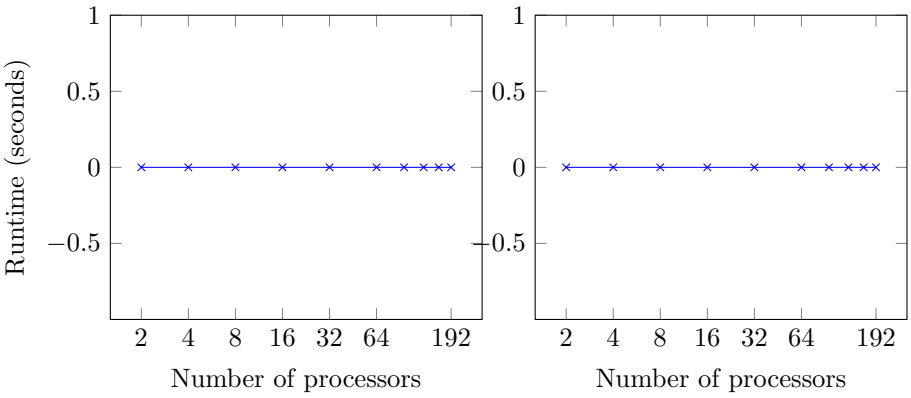


(a) 50 eigenvalues. (b) 500 eigenvalues.

Figure 4 – Proportion of each step in the inverse subspace iteration.

We observe that the Gram-Schmidt orthogonalisation is the limiting factor and is the most time-consuming step of the inverse iteration as the number of processors grows. It is a well-known problem that the simple Gram-Schmidt process is actually difficult to parallelise efficiently. Small optimisations for a parallel Gram-Schmidt orthogonalisation exist [11] but they do not properly solve the problem.

This issue will be difficult to overcome completely. But fundamentally, the orthogonalisation is used to stabilise the algorithm. To accelerate our algorithm further, we try to orthogonalise the vectors X_k every second iteration instead of every iteration. We present below the resulting performances:



(a) 50 eigenvalues.

(b) 500 eigenvalues.

Figure 5 – Runtime of the inverse subspace iteration part of the algorithm.

We observe ...

4 Conclusion

4.1 Discussions

Linear solvers and domain decomposition methods on dense matrices TODO

Gram-Schmidt process We saw that the orthogonalisation process is difficult to parallelise efficiently. Skipping it, to stabilise the algorithm less often, gave an improvement, but we cannot totally avoid the cost of it.

4.2 Perspectives

Image processing An improvement of our algorithm would obviously to finish the part computing the output image. This requires to compute the inverse of the square root of the matrix. This can be done either by computing the entire eigendecomposition to get the inverse square root matrix. Another way to accomplish this, could be to consider the Cauchy integral formula such as:

$$A^{-\frac{1}{2}} = \frac{1}{2\pi i} \oint_C z^{-\frac{1}{2}} (zI - A)^{-1} \mathrm{d}z.$$

However, the time for this project was not enough to explore this possibility completely.

An easy improvement would be to also consider color images. This can be done by decomposing the RGB image to a YCC image, with a grayscale and

two chroma components. The algorithm is applied to all three components, and then they are converted back to RGB.

A way to improve the filtering is multiscale decomposition. As explained in [3], instead of applying a linear function to all eigenvalues such as $f(W) = \phi f(\Pi)\phi^T$, we can actually use a polynomial function f. This is interesting because each eigenpair captured various features of the image and one can apply different coefficients on different aspects of the image.

For the state-of-the-art, the article [12] proposes an enhancement of global filtering. It argues that the eigendecomposition remains computationally too expense and shows results of an improvement. The presented results and performances are astonishing; however, the method is hardly described and replicating it would be difficult. This is understandable since this algorithm seems to be in the latest Pixel 2 smartphone by Google and they want to preserve their market advantage in the field of image processing.

Linear solver A real improvement would be to formulate a method for extending the trailing eigenvectors of the sampled Laplacian $\mathcal{L}_A$.

References

[1] Peyman Milanfar. *Non-conformist Image Processing with Graph Laplacian Operator*. 2016. URL: https://www.pathlms.com/siam/courses/2426/sections/3234.

[2] Hossein Talebi and Peyman Milanfar. "Global Image Denoising". In: *IEEE Transactions on Image Processing* 23.2 (Feb. 2014), pp. 755–768. ISSN: 1057-7149, 1941-0042. DOI: 10.1109/TIP.2013.2293425. URL: http://ieeexplore.ieee.org/document/6678291/.

[3] H. Talebi and P. Milanfar. "Nonlocal Image Editing". In: *IEEE Transactions on Image Processing* 23.10 (2014), pp. 4460–4473. ISSN: 1057-7149. DOI: 10.1109/TIP.2014.2348870.

[4] Charless Fowlkes et al. "Spectral grouping using the Nystrom method". In: *IEEE transactions on pattern analysis and machine intelligence* 26.2 (2004), pp. 214–225. URL: http://ieeexplore.ieee.org/abstract/document/1262185/.

[5] Serge Belongie et al. "Spectral partitioning with indefinite kernels using the Nyström extension". In: *European conference on computer vision*. Springer, 2002, pp. 531–542.

[6] Peyman Milanfar. "A Tour of Modern Image Filtering: New Insights and Methods, Both Practical and Theoretical". In: *IEEE Signal Processing Magazine* 30.1 (Jan. 2013), pp. 106–128. ISSN: 1053-5888. DOI: 10.1109/MSP.2011.2179329. URL: http://ieeexplore.ieee.org/document/6375938/.

[7] Satish Balay et al. *PETSc Web page*. http://www.mcs.anl.gov/petsc. 2017. URL: http://www.mcs.anl.gov/petsc.

[8] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. "SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems". In: *ACM Trans. Math. Software* 31.3 (2005), pp. 351–362.

[9] Jack Poulson et al. "Elemental: A New Framework for Distributed Memory Dense Matrix Computations". In: *ACM Trans. Math. Softw.* 39.2 (Feb. 2013), 13:1–13:24. ISSN: 0098-3500. DOI: 10.1145/2427023.2427030. URL: http://doi.acm.org/10.1145/2427023.2427030.

[10] G. El Khoury, Yu. M. Nechepurenko, and M. Sadkane. "Acceleration of inverse subspace iteration with Newton's method". In: *Journal of Computational and Applied Mathematics*. Proceedings of the Sixteenth International Congress on Computational and Applied Mathematics (ICCAM-2012), Ghent, Belgium, 9-13 July, 2012 259 (Mar. 2014), pp. 205–215. ISSN: 0377-0427. DOI: 10.1016/j.cam.2013.06.046. URL: http://www.sciencedirect.com/science/article/pii/S0377042713003440.

[11] Takahiro Katagiri. "Performance Evaluation of Parallel Gram-Schmidt Re-orthogonalization Methods". In: *High Performance Computing for Computational Science — VECPAR 2002*. Ed. by José M. L. M. Palma et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 302–314. ISBN: 978-3-540-36569-3.

[12] Hossein Talebi and Peyman Milanfar. "Fast Multilayer Laplacian Enhancement". In: *IEEE Transactions on Computational Imaging* 2.4 (Dec. 2016), pp. 496–509. ISSN: 2333-9403, 2334-0118. DOI: 10.1109/TCI.2016.2607142. URL: http://ieeexplore.ieee.org/document/7563313/.