Acknowledgments

- I would like to thank to Oracle MDC and Cinvestav for conceiving and sponsoring the Master Program. Specially to the people who walked with us during all these years: from Oracle MDC side Director Erik Peterson, Sandra Vargas and Hugo Coyote; and from Cinvestav side Dr. Andrés Méndez.

- To my former local senior manager, Carlos Ordonez, for believing in me and giving me the great opportunity of joining the Master Program.

- To my local managers Julio Méndez and Juan Pineda, and to the dotted ones Shashanka Malathesha, Ashutosh Gupta and Lik Wong; for being supportive and understanding about the time investment that the Master Program required.

- To all our professors, who enthusiastically and patiently shared with us part of their knowledge.

- To Dustin Harvey at Oracle, for giving me a great topic for my thesis.

- To my thesis directors, Dr. Andrés Méndez and Dr. Isaac Scherson, for all their great support and guidance during the last year.

- To my family, mainly my wife for her immense patience and support.

Resumen

En el contexto de una aplicación de la vida real que usa Spectral Clustering; el objetivo de esta tesis es emitir una recomendación de algoritmo, para el cálculo eficiente del Fiedler Vector en un solo procesador (requerimiento de la aplicación). Los algoritmos considerados fueron MRRR, Lanczos/IRLM y LOBPCG. El primero está diseñado para matrices densas, así que realmente no es un competidor; pero fue incluido para efectos de comparación (dado que la aplicación actualmente usa un procedimiento de ese tipo). Los algoritmos fueron puestos a competir a través de una serie de pruebas en una laptop estándar, usando matrices generadas del dominio de la aplicación. Aun con un solver lineal optimizado tipo Cholesky, la implementación de Lanczos/IRLM más conocida (ARPACK) no pudo competir con LOBPCG, quién se vuelve el claro ganador del experimento. Los resultados sugieren que para las matrices más grandes consideradas (size ≈ 4500), los tiempos de cálculo pueden moverse de la escala de minutos a sub-segundos; y a un par de segundos, si agregamos el tiempo del pre-procesamiento para eliminar clustered eigenvalues.

Abstract

On the context of a real-life application doing Spectral Clustering; the objective of this thesis is to emit an algorithm recommendation, for efficient computation of the the Fiedler Vector on a single processor (application requirement). The algorithms considered were MRRR, Lanczos/IRLM and LOBPCG. The first one is designed for dense matrices, hence not really a competitor here; but was included for the sake of comparison (as the application currently uses a procedure of such kind). The algorithms were put into a competition through a series of tests on a commodity laptop, using matrices generated from the application's domain. Even with an optimized Cholesky linear solver, the best known Lanczos/IRLM implementation (ARPACK) failed to compete with LOBPCG, which comes as the clear winner of the experiment. The results suggest that for the biggest matrices considered (size ≈ 4500), the execution times can move from the minute into the sub-second scale; and to a couple of seconds, if we include the time of the pre-processing to eliminate clustered eigenvalues.

Contents

List of Tables **9**

List of Figures **11**

Glossary **13**

1 Introduction **15**
1.1 Motivation 15
1.2 Objective 16
1.3 Scope 16
1.4 Outline 17

2 Background **19**
2.1 Spectral Clustering 19
2.1.1 Similarity Graphs 20
2.1.2 Computing partitions 21
2.1.3 Minimal Bi-Partitional RatioCut 21
2.1.4 The Laplacian and its Fiedler Vector 24
2.2 The Symmetric Eigenproblem 25
2.2.1 The Spectral Theorem 25
2.2.2 Numerical Considerations 26
2.2.2.1 Conditioning 26
2.2.2.2 Numerical Stability 27

3 The Algorithms **29**
3.1 Proposal 29
3.1.1 Multiple Relatively Robust Representations (MRRR) . 30

3.1.2 Lanczos: variant Implicitly Restarted Lanczos Method (IRLM) . . . 30
3.1.3 Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) . . . 30
3.2 Preliminaries . . . 31
3.2.1 Power Method (for Symmetric Matrices) . . . 31
3.2.2 Krylov Subspaces . . . 32
3.2.3 Rayleigh-Ritz Method . . . 33
3.3 Dense Matrix Algorithms . . . 34
3.3.1 Symmetric Tridiagonal QL Algorithm . . . 34
3.3.2 The MRRR Algorithm . . . 35
3.4 Sparse Matrix Algorithms . . . 35
3.4.1 Lanczos (IRLM) . . . 35
3.4.2 LOBPCG . . . 37

4 The Experiment **39**
4.1 Setup . . . 39
4.1.1 Hardware . . . 39
4.1.2 Software . . . 39
4.2 Data Preparation . . . 41
4.2.1 The actual data . . . 41
4.2.2 Sparse formats . . . 41
4.2.3 Avoiding Clustered Eigenvalues . . . 41
4.2.4 Shifting the spectra . . . 44
4.3 Results . . . 44
4.4 Why LOBPCG beats IRLM? . . . 47
4.4.1 No need for restarting . . . 47
4.4.2 Clever search strategy . . . 48
4.4.3 Search constrains / Requested spectra . . . 48
4.4.4 Cheaper iterations . . . 49
4.4.5 Use of preconditioner . . . 49
4.4.6 Clustered eigenvalues . . . 50

5 Conclusions **51**
5.1 Recommendations . . . 51
5.1.1 Prefer Sparse Matrix Algorithms . . . 51
5.1.2 If dense, go with MRRR . . . 51

5.1.3 If development time is a constraint, consider Lanczos/IRLM 52
5.1.4 If speed is a concern, go with LOBPCG 52
5.1.5 Use an specialized algorithm for computing SCC 53
5.2 Additional considerations 53
5.2.1 Need for sparse format 53
5.2.2 Port of the algorithm for computing the Strongly Connected Components (SCC) 54
5.3 Future Work 54
5.3.1 Numerical Linear Algebra is hard 54
5.3.2 There is a zoo of algorithms out there 55
5.3.3 Measure the error in the predictions vs real-life execution 56

Bibliography **57**

List of Tables

4.1 Hardware used on the experiment 39
4.2 Lanczos(IRLM) vs LOBPCG 48

List of Figures

2.1 Minimal Bi-Partitional RatioCut 23

4.1 Experiment Results in CSR format 45
4.2 Experiment Results in CSC format 47

Glossary

Cholmod Software package implementing an sparse linear solver for symmetric positive definite matrices. Used in this work along with IRLM from ARPACK. 32, 40, 42, 48

Clustered Eigenvalues An scenario where certain eigenvalues of a matrix are very close to each other. It usually implies slow convergence on Power Method based algorithms. 13, 23, 24, 28, 32, 34, 37, 39, 42, 46, 49

Fiedler Vector The eigenvector associated to the second smallest eigenvalue of the Laplacian Matrix. 12, 13, 15, 17, 18, 20, 21, 26, 33, 47, 48, 51, 52

Implicitly Restarted Lanczos Method The variant of the Lanczos family of algorithms implemented by ARPACK, widely used for solving the Symmetric Eigenproblem of sparse matrices. 13

IRLM Implicitly Restarted Lanczos Method. 13, 24–26, 31, 32, 34, 36, 37, 40, 42–46, 48, 49, 51, *Glossary:* Implicitly Restarted Lanczos Method

Laplacian Refers to the unnormalized version, which is matrix derived from the data graph as $D - W$, where D is the diagonal of degrees and W is the weights matrix. 11, 12, 15, 18, 20, 21, 26, 31–33, 37–40, 42, 47, 49, 51

LOBPCG Locally Optimal Block Preconditioned Conjugate Gradient. 13, 24–26, 33, 34, 36, 37, 40–46, 48, 49, 52, *Glossary:* Locally Optimal Block Preconditioned Conjugate Gradient

Locally Optimal Block Preconditioned Conjugate Gradient Sparse matrix algorithm invented by A. Knyazev for solving the Symmetric Eigenproblem, it is also the winner on the comparison made on this thesis. 13

MRRR Multiple Relatively Robust Representations. 13, 24–26, 31, 36, 42, 47, *Glossary:* Multiple Relatively Robust Representations

Multiple Relatively Robust Representations A symmetric dense-matrix algorithm for solving the Eigenproblem. 13

SCC Strongly Connected Components. 36, 38, 39, 49, 50, *Glossary:* Strongly Connected Components

Strongly Connected Components Refers to the Strongly Connected Components of the graph behind the Laplacian matrix. Having more than one will imply a disconnected graph. 36

SuperLU Software package implementing an sparse linear solver for general matrices. Used in this work along with IRLM from ARPACK. 32, 42

Chapter 1

Introduction

1.1 Motivation

We live in the age of information, massive amounts of data arise on a daily basis from multiple sources and we need techniques to automatically analyze and understand such data. Data Clustering is one of such techniques, which aims to automatically group the data according to a given notion of similarity [39]. According to Luxburg [35] "In virtually every scientific field dealing with empirical data, people attempt to get a first impression on their data by trying to identify groups of *similar behavior* in their data".

Spectral Clustering is an important branch of Data Clustering, where those data groups (clusters) are found using Linear Algebra methods. To be more concrete, the usage of numerical algorithms for computing eigenvalues (aka spectra) or eigenvectors is what gives this branch of Data Clustering its distinctive name. The matrices used to compute the eigenpairs are special constructions, produced out of the domain data. In this thesis we focus entirely on the Laplacian matrix L, an explanation of how we build the Laplacian matrix appears in (section 2.1.4).

The quality of the clusters obtained by Spectral Clustering is usually better than those coming from "traditional algorithms" like k-means [35]. Another potential advantage is the nowadays ubiquity of Numerical Linear Algebra libraries, in particular of the so called eigen-solvers.

1.2 Objective

Spectral Clustering has plenty of applications ranging from engineering to social sciences, but this thesis revolves about a particular real-life cloud-based application that needed help. The application is proprietary code, hence we can not reveal further details about it.

Aiming to perform Spectral Clustering, the application in question computes an eigenvector called Fiedler Vector (see more details in (chapter 2)), but the algorithm they are using does not seem to scale properly and represents a performance bottleneck.

Hence, the main objective of this thesis, is to give an algorithm recommendation for the mentioned real-life application. That is, to suggest an algorithm for computing the Fiedler Vector that outperforms the one they are using today. For this goal, we restrict our attention to the matrices produced by the application, and to the algorithms that run in serial fashion (single processor/core). This later requirement comes from the application architecture itself, for which we can not reveal further details either (proprietary code).

1.3 Scope

The field of Numerical Analysis, and in particular Linear Numerical Algebra, is as interesting as immense [19]. There are plenty of algorithms out there that solve the Eigenproblem [18]. Furthermore, the theory behind these algorithms is dense and hard to grasp for the non-initiated in these topics. Hence, given the time constraints we had for making this thesis project, we focused on the following points only:

- Main ideas of the algorithms, seeking to compare them.
- How to use their implementations in practice.
- Restrict the search to serial versions of the algorithms (application requirement).
- Look for algorithms that leverage the Laplacian's properties (Symmetric, Sparse, Positive-Semi-Definite and with first eigenpair $(0, \mathbf{1})$). The

more the algorithms took into account these, the better they performed in practice.

Using the above criteria, we focused only on three algorithms: MRRR, IRLM and LOBPCG (see (chapter 3) for more details about this selection). Still, entire books could be written with the details of any of these, but again we restrict ourselves to the points stated above. Additionally, we paid special attention to avoid the issues observed on the presence of Clustered Eigenvalues (see (section 4.2.3) for more details about the used technique).

1.4 Outline

The rest of this work is organized as follows. In (chapter 2) we provide more details about the particular problem we are attacking (computing the Fiedler Vector), by setting a bit of context about its origin. Then on (chapter 3) we explain, within our stated scope, the details of the chosen algorithms (both theoretical and practical). The (section 3.1) of this chapter aims to justify the selection. After that, (chapter 4) describes the experiments we did and the results of them, they are basically a comparison between the chosen algorithms for some selected matrices [1]. The (section 4.4) of same chapter, makes an attempt to explain the outcome of the results, based on the theory from (chapter 3). Finally, based on the results we present the conclusions in (chapter 5), where the main contribution of course, is the set of recommendations for the real-life application.

[1] Taken from the application domain.

Chapter 2

Background

2.1 Spectral Clustering

It is convenient to set a little context, in order to appreciate better what is the operation that this thesis tries to optimize. For this we enter a bit into the Spectral Clustering world. In such context the Laplacian Matrix and its second smallest eigenvector [1] will emerge, and it will become clearer why computing an eigenvector can serve Data Clustering. The original merit of finding the value of the Fiedler Vector goes to Fiedler [16], considered the father of Algebraic Graph Theory. But the further developments brought by Hagen et al [21] and Shi et al [44], are closer to the usage seen in our target application.

The stuff we briefly discuss on this chapter could be considered part of the more theoretical area of Spectral Graph Theory [7]. But in more practical terms, the material exposed here comes mainly from the standard introductory tutorial to Spectral Clustering by Luxburg [35], and from the excellent lecture that Gao gives about the topic on its Data Mining Course at the University of Buffalo [25].

[1] This is an abuse of the language of course, when we say the smallest or biggest eigenvector/eigenpair, we actually refer to the property of the associated eigenvalue.

2.1.1 Similarity Graphs

If one is doing Data Clustering, one encodes the applications' data into vectors $\mathbf{x} \in \mathbb{R}^n$, but if one is aiming to do Spectral Clustering we will also need to build an auxiliary graph. This is because in Spectral Clustering, the operation of computing clusters is reduced to that of computing graph partitions. To do that, we need to label our encoded data (vectors) as nodes in an undirected weighted graph $G = (V, W)$, where the edges' weights will be given by a similarity function defined for the problem. This graph is known as the *Similarity Graph*.

Besides the similarity function, which determines the weights on the graph, there are other sources of variation in the way we connect the nodes. Below some of the most popular constructions used in Spectral Clustering, taken from [35].

- The $\epsilon-$neighborhood graph: In this graph we connect a couple of nodes x and y only if the value of the similarity function is smaller than certain ratio called ϵ. As the resulting edges will have values of roughly the same scale, there is not much additional information in leaving the weights in this graph.

- The $k-$nearest neighbor graph: In this case a pair of vertices x and y will be connected, only if y is among the $k-$nearest neighbors of x. This construction leads to a directed graph, but there are ways to transform it into an undirected one (like simply ignoring the direction of the edges, or considering mutual neighborhood).

- The fully connected graph: Here the only requirement for connecting a couple of vertices x and y is that their similarity function gives a positive value. This type of construction usually requires a more sophisticated similarity function than a mere euclidean distance, one that does consider the local neighborhood of the underlying data.

In the case of this thesis project, the similarity graph was already given as input data. Therefore, we did not need to deal with the question of what

particular technique to use for its construction. The details of the similarity function and the graph itself, can not be disclosed here due patents involved.

2.1.2 Computing partitions

Once the *Similarity Graph* is built, the next task in Spectral Clustering is to use an algorithm to partition it, and from those partitions to derive the data clusters. This derivation could be immediate, if we assume that each node in the *Similarity Graph* represents one vector from our data set.

While Luxburg presents more sophisticated and practical algorithms [35], we prefer the simpler but more illustrative one presented by Shi and Malik [44]. This is because such algorithm, allows one to see explicitly how the repetitive computation of the Fiedler Vector can produce a complete partition (clustering). The (algorithm 1) shows a template version of the method, where we can mention the variations related to our work.

Although the general application of Spectral Clustering is to produce a total partition of the graph, we could consider that the basic operation is the bi-partition. Given that we can not reveal the actual algorithm used by our application, let us just say that it also has as a basic block the bi-partition task. That is why, we focus this thesis on optimizing such single operation. The (section 2.1.3) provides deeper details about it, and (section 2.1.4) explains its relationship with the Fiedler Vector.

2.1.3 Minimal Bi-Partitional RatioCut

The work of this thesis lies around the basic operation of bi-partitioning the similarity graph (removing certain edges such that we disconnect the vertices set). This is to be done in such way that the *cut* function $\left(\text{cut}(A, B) = \sum_{i \in A \, , \, j \in B} w_{ij} \right)$ is minimized. This will bring as result two partitions containing quite similar nodes (according to the similarity function defined), and then, they could be thought as clusters. The (fig. 2.1), adapted from [25], illustrates this operation.

Algorithm 1: Recursive 2-way Cut Algorithm

Input : Similarity matrix and number k of maximum allowed bipartitions (cuts).

Output: A partitioned graph, where each partition represents a cluster.

1. Construct a similarity graph, for example, with one of the techniques mentioned in (section 2.1.1).

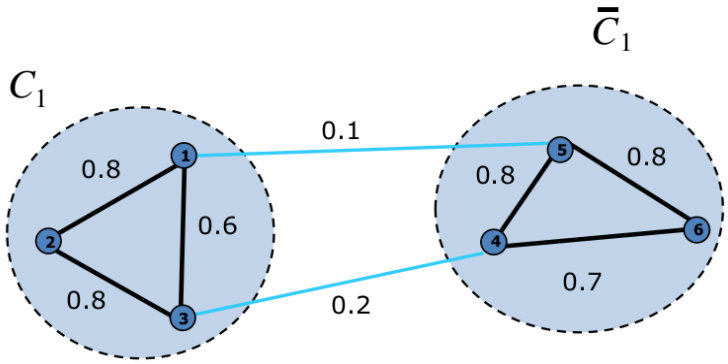
2. Compute the Laplacian of the graph. There is more than one type of Laplacian, but the one we will use later is the so called Unnormalized Laplacian (see (section 2.1.4)).

3. Compute the second eigenvector (Fiedler Vector) of the Laplacian, as it is the one optimizing the chosen objective function. The purpose of this function is to seek for well balanced partitions, and the most popular choices are Ncut [44] and RatioCut [21]. We focused on RatioCut function in this thesis, as that was the version used by the application (see (section 2.1.3) for further details).

4. Using the computed Fiedler Vector proceed to calculate the bi-partition of the *Similarity Graph*. There is more than one way of achieving this, and [21] presents several of them for the RatioCut function. The one we used in the further sections is simply the sign function (see (section 2.1.3) for further details).

5. If the number of cuts has not exceeded the given parameter k, and if the current partitions need further splitting (according to a predefined criteria), then proceed to recursively bi-partition the selected pieces.

Figure 2.1: Minimal Bi-Partitional RatioCut



The sky blue edges are the minimal *cut* for this example. This results in two partitions C_1 and $\overline{C_1}$. While there is more than one way of formalizing this problem, we care about the one using the *RatioCut* function (Minimal Bi-Partitional RatioCut Problem), which favors the solutions containing partitions of roughly the same size (eq. (2.1)).

$$\min_{C_1 \subset V} \underbrace{\frac{1}{2}\left[\frac{\text{cut}(C_1, \overline{C_1})}{|C_1|} + \frac{\text{cut}(\overline{C_1}, C_1)}{|\overline{C_1}|}\right]}_{\text{RatioCut}(C_1, \overline{C_1})} \quad \ni \quad \text{cut}(A, B) = \sum_{i \in A\ ,\ j \in B} w_{ij} \tag{2.1}$$

Asking for the partitions to have similar sizes, avoids the non interesting solutions which include singletons[2], but it also makes the optimization problem NP-Hard (see [48]). This is where Spectral Clustering comes to rescue us: By using Linear Algebra techniques, it can approximate the solutions to the problem above.

[2] That is, solutions which include a partition with a single node.

2.1.4 The Laplacian and its Fiedler Vector

It turns out that if we define the (unnormalized) Laplacian Matrix as $L = D - W$ [3], then the Minimal Bi-Partitional RatioCut Problem reduces to that of minimizing the Rayleigh-Quotient Function (see (eq. (2.2))). The details can be consulted at [35] or [25].

$$\min_{C_1 \subset V} \text{RatioCut}(C1, \overline{C1}) \quad \equiv \quad \min_{\mathbf{f} \in \mathbb{R}^n} \underbrace{\left(\frac{\mathbf{f}^T L \, \mathbf{f}}{\mathbf{f}^T \mathbf{f}} \right)}_{\text{Rayleigh-Quotient}} \quad \ni \quad \mathbf{f} \in \mathbb{R}^n \ \wedge \ \mathbf{f} \perp \mathbf{1} \tag{2.2}$$

The solution given by the vector $\mathbf{f}$ is an approximation, and it acts as an indicator function. This means that each one of its cells represents one node in the graph, and its sign tells whether the associated node belongs to partition C_1 or its complement $\overline{C_1}$. The property of f being orthogonal to the vector $\mathbf{1}$, is a consequence of the Laplacian matrix (L) properties (see [35]).

Having formulated the problem in terms of the Rayleigh-Quotient function, allows us to use the immense arsenal that Linear Algebra has at our disposal. This step is usually skip in literature, but here we include a more explicit argument. The concrete theorem that clearly gives a solution to the above problem is the Courant-Fischer Theorem. In (eq. (2.3)) we see an specialized version of such theorem, for the case of the second smallest eigenvalue of a real-symmetric matrix A (see [19] for its general formulation).

$$\lambda_2(A) = \max_{\dim(U) = n-1} \left[\min_{\mathbf{x} \in U \ \wedge \ \|\mathbf{x}\| \neq 0} \left(\frac{\mathbf{x}^T A \, \mathbf{x}}{\mathbf{x}^T \mathbf{x}} \right) \right] \tag{2.3}$$

The above result basically tells us that, if we were able to navigate throughout all the subspaces of dimension $(n-1)$ [4] (where n is the dimension of the Laplacian matrix), then we would know that the answer to our

[3] Where D is the diagonal containing the nodes' degrees and W is the weights matrix of the graph.

[4] Beware that the formulation of the theorem by Golub in [19], may look different on the surface, because it mentions that the dimension of the subspaces is k for λ_k. But this difference is just cosmetic, because Golub likes to sort the eigenvalues on descending order, contrary to our convention here (ascending). Thus for him, λ_2 is actually the second

problem is the eigenvector associated to the second-smallest eigenvalue of the Laplacian (called Fiedler Vector). The only obstacle in applying this result, is the non practical task of iterating over an infinite number of subspaces, but the previously stated property of $\mathbf{f} \perp \mathbf{1}$ solves the problem (see [35] for more details about this property). We do not need to explore all those subspaces, because we know where to look for: in $1^{\perp}$. This is because the Laplacian is known to have $\mathbf{1}$ as first eigenvector, and by properties of symmetric matrices, the rest of the eigenvectors (including the Fiedler Vector) will need to live on the orthogonal subspace to $\mathbf{1}$.

2.2 The Symmetric Eigenproblem

The previous sections showed that our Minimal Bi-Partitional RatioCut Problem can be reduced to minimize the Rayleigh-Quotient function, which in turn can be solved by computing the Fiedler Vector of the Laplacian matrix (or in general, to that of computing the second eigenpair[5]). Our restated problem appears in (eq. (2.4)).

$$\min_{\mathbf{x} \perp \mathbf{1} \, \wedge \, \|\mathbf{x}\| \neq 0} \left(\frac{\mathbf{x}^T L \, \mathbf{x}}{\mathbf{x}^T \mathbf{x}} \right) = \lambda_2 \;\; \ni \;\; L\mathbf{x} = \lambda_2 \mathbf{x} \tag{2.4}$$

Before moving on with the algorithm details of further chapters, this section provides minimal background theory about such new problem (Symmetric Eigenproblem). Conscious that this fascinating branch of Numerical Linear Algebra is huge, and that we can provide here just a little taste, we suggest the authoritative references of [40], [43], [9], [11] or [2] for a deeper dive.

2.2.1 The Spectral Theorem

In dealing with our restated problem, perhaps one of the first things we should ask ourselves is when it has a solution. For symmetric matrices like the

biggest eigenvalue. Therefore, the dimension of the subspaces where we need to search is actually k, because k happens to be $n-1$ for the second smallest eigenvalue in Golub's notation (we used $n-1$ in the theorem statement due consistency with our ascending convention for eigenvalues).

[5] By eigenpair we mean a certain eigenvalue and its associated eigenvector.

Laplacian, this question has a definite answer given by the Spectral Theorem (symmetric version):

Theorem 2.1 (Spectral Theorem)**.** *If A is a real symmetric matrix $\implies$ there exists a real orthogonal matrix Q such that $Q^T AQ$ is diagonal.*

In the context of (theorem 2.1), matrix Q contains the eigenvectors and matrix $Q^T AQ$ contains the eigenvalues on its diagonal. There are several proofs of the above theorem, but the one we recommend for its originality is by Wilf [49].

2.2.2 Numerical Considerations

Having the guarantee of a theoretical solution does not imply that we can approximate it numerically without problems. Numerical Analysis, and in particular Linear Numerical Algebra, are immense fields and the amount of context required to grasp their theory is abundant. But here we just briefly focus on a couple of aspects that are the bare minimum required, before moving on to the more practical chapters: We are talking about conditioning and numerical stability. The information mentioned below was taken from [6].

2.2.2.1 Conditioning

Regardless of the algorithm that we employ to solve the Symmetric Eigenproblem, is worth to note that the problem itself has a property called conditioning. In essence, it aims to measure whether small perturbations on the input will produce small or big perturbations on the answer (problems are called ill-conditioned if the answer perturbation is big, and well-conditioned otherwise). For our problem there are two subproblems: computing eigenvalues and computing eigenvectors [6]. The (eq. (2.5)) and (eq. (2.6)) summarize the conditioning of both subproblems (respectively).

$$\kappa_i(A) = \frac{|\lambda_{max}|}{|\lambda_i|} \tag{2.5}$$

[6] We just need the eigenvector, but software routines typically compute both the eigenvalue and the eigenvector, which we call an eigenpair.

$$\Delta \mathbf{v}_i \approx \sum_{i \neq j} \frac{{\mathbf{v}_j}^T \Delta A \mathbf{v}_i}{\lambda_i - \lambda_j} \tag{2.6}$$

The (eq. (2.5)) defines the condition number for each eigenvalue λ_j , and we can see that is a ratio between the maximum eigenvalue and the one in question. This means that the closer we get to the maximum (in magnitude), the smallest (better) our condition number will be. On the other hand, (eq. (2.5)) describes the perturbation of an eigenvector $\mathbf{v}_i$ of A regarding an small perturbation on the matrix A (called ΔA). We can observe that the perturbation of $\mathbf{v}_i$ has a contribution from every eigenpair, and the critical part is the denominator. The closer the eigenvalues are, the bigger the perturbation will be.

The above paragraph basically says that for eigenvalues, we only face trouble if we target quite small ones, which is not the case for our particular application [7]. But above also says that eigenvectors could be ill-conditioned even if eigenvalues are not, which happens when these are clustered (quite close to each other, and from now on referred as Clustered Eigenvalues). This is a situation we faced in practice for this project, but later chapters will mention the workaround that we found for this scenario (plus the actual, more practical reasons, that we had to avoid it, see (section 4.2.3)).

2.2.2.2 Numerical Stability

Numerical Stability is a property of the algorithms we use, and in short it tells how much additional "numerical noise" the computation adds to the solution (in addition to the perturbation inherent to the problem conditioning). There is more than one way of formalizing this notion, but a particularly useful one in Numerical Linear Algebra is Backward Numerical Stability. The (definition 2.2.1) is taken from [11].

Definition 2.2.1. Backward Numerical Stability If alg($\mathbf{x}$) is our algorithm for computing f($\mathbf{x}$), including the effects of roundoff, we call alg($\mathbf{x}$) a *backward stable algorithm* for f($\mathbf{x}$) if for all $\mathbf{x}$ there is a "small" Δx such that alg(x) = f($\mathbf{x} + \Delta \mathbf{x}$). Δx is called the *backward error*.

[7] For our particular matrices, even if we compute the second smallest eigenvalue, it is still big enough to avoid spikes on the condition number $\kappa_2(A)$.

Backward stability means in essence, that the algorithms compute the exact answer of a very close problem (in our case $L + \Delta L$). Backward stable algorithms are nice, in the sense that they do not add a significant error to the answer. Thus, they allow us to focus only on the the perturbations given by the problem conditioning. In our concrete case of the Symmetric Eigenproblem, and the type of matrices our application produces, this would reduce to avoiding Clustered Eigenvalues.

When we started this thesis, we thought that it was custom to have formal proofs regarding the overall numerical stability of algorithms, in particular for the case of eigensolvers. But for the concrete algorithms we selected and experimented with, only MRRR has references to formal proofs about its numerical stability (and not precisely for the overall algorithm, but for some portions of the calculations [8], see [14]).

Above does not mean that IRLM or LOBPCG do not consider stability at all. The literature (see [33] and [29]) does indeed mention concerns, precautions and a general focus on achieving this property. Furthermore, all that work is reflected on the good quality of the results produced (when comparing against MRRR, the gold standard in accuracy). It is just that, the expectation of finding an explicit proof for the whole algorithm's stability, does not seem to be a common practice by Numerical Analysts. We contacted Knyazev himself, in order to ask about any proof of LOBPCG. He kindly explained that this expectation of the proof was too high. Therefore, we are happy with claiming that IRLM and LOBPCG are stable in a "practical sense", and that the reputation of their authors, along with the good results seen in experiments, are enough for us to trust them.

[8] Which ultimately makes sense, as the numerical software is quite complex, built from a lot of subroutines and layers. In general, it should be easier to discuss numerical stability at a more granular level, rather than for the whole algorithm.

Chapter 3

The Algorithms

3.1 Proposal

There is an immense quantity of algorithms out there, for solving the Symmetric Eigenproblem. But by considering the symmetry, we can focus on a considerably smaller subset of algorithms. As mentioned already, the properties of the Laplacian drive our further pruning as it involves sparse and positive semi-definite matrices.

Our proposal is essentially to pick just three algorithms from that immense tree and to evaluate their performance with a set of matrices generated from the application domain. From such experiment, we will pick the winner and hence, emit our official recommendation for the application. While (chapter 4) and (chapter 5) detail the experiment and its outcome. This chapter focuses on explaining the rationale behind this algorithms selection, along with a little background about their theory and practical usage.

Let us begin by saying that our intention with this selection, was to represent three families or approaches to the solution of the problem: Current approach taken by the application (MRRR), mainstream recommendation for Spectral Clustering (IRLM) and a modern proposal (LOBPCG). In the subsequent sub-sections we provide further justification for the selection of these algorithms.

3.1.1 MRRR

Although the Laplacian is a sparse matrix (70% of zeros for our target application), we still consider the family of algorithms that care about dense matrix representations. This is mostly due comparison purposes, as the current implementation the application uses, is based on a method from this family. This will allow us to know how much we improved the execution time with the new sparse matrix algorithms. This is the argument behind the selection of the MRRR algorithm, which is not exactly the one used today by the application. Thus, it can be considered as a lower bound in terms of its performance. Whatever the application uses today is hardly better than MRRR, which is considered the "holy grail" of Numerical Linear Algebra [23] (at least for dense matrices).

But the real search is within the sparse matrix sub-branch of algorithms, and we justify our selection of IRLM and LOBPCG in next two sub-sections.

3.1.2 Lanczos: variant IRLM

Members of the Lanczos algorithms family, usually appear in Spectral Clustering literature as the mainstream option for computing the eigenpairs of the Laplacian (see [35] for example). Among them, the Implicitly Restarted Lanczos Method (IRLM), is perhaps the most known and widely available variant, thanks to the ARPACK [33] software package. This situation made us chose IRLM then, as it represents the common practice against which we want to compare further algorithms (In reality, we are also going to compare against a dense matrix algorithm).

3.1.3 LOBPCG

The real proposal is to use LOBPCG, which is a more modern algorithm for solving the Symmetric Eigenproblem for sparse matrices. The existing literature, see [30], suggest the superiority of LOBPCG for scenarios like ours (computation of the Fiedler Vector).

3.2 Preliminaries

This section introduces the common mathematical tools used by the selected algorithms.

3.2.1 Power Method (for Symmetric Matrices)

The most primitive algorithm for computing eigenvectors is the so called Power Method. Although the slowest algorithm, it compensates its limitations with its extremely simple definition. It can be summarized by the two steps in (eq. (3.1)), where $\mathbf{x}$ is the initial approximation to the wanted eigenvector.

$$\mathbf{x}_0 = \mathbf{x} \quad \wedge \quad \mathbf{x}_{k+1} = A\mathbf{x}_k \tag{3.1}$$

The iteration from (eq. (3.1)), surprisingly, converges to the eigenvector associated to the biggest eigenvalue of the matrix A (or to the one associated with the smallest eigenvalue, if one uses A^{-1} instead [1]). There is a generic proof of this on [43], but a more accessible argument is presented in (eq. (3.2)) for the particular case of symmetric matrices. The vectors $\mathbf{v}_i$ are the eigenvectors of the matrix, and the key argument for the development is that we assumed the matrix was symmetric, which implies its eigenvectors form a full basis. This means any vector, including x_0, can be expressed as a linear combination of them (see [46] or [19] for more details).

[1] The presence of the inverse matrix A^{-1} is mostly due notation, in practice we do not compute $A^{-1}x$, instead we solve the associated linear system $A\mathbf{y} = \mathbf{x}$.

$$\begin{aligned}
\mathbf{x}_k &= A^k \mathbf{x_0} \\
&= A^k \left(\sum_{i=1}^{n} \alpha_i \mathbf{v}_i \right) \\
&= \sum_{i=1}^{n} \alpha_i A^k \mathbf{v}_i \\
&= \sum_{i=1}^{n} \alpha_i \lambda_i^k \mathbf{v}_i \\
&= \alpha_n \lambda_n^k \mathbf{v}_n + \sum_{i=1}^{n-1} \alpha_i \lambda_i^k \mathbf{v}_i \\
&\overbrace{\approx}^{k \to \infty} \beta \mathbf{v}_n
\end{aligned} \tag{3.2}$$

The Power Method does not always converge in exact arithmetic [2], but here we focus on its more commonly found numeric limitation: slow convergence (the reader can consult further details about the Power Method in [19] and [40]). This is because it relies on the eigengap (distance between the desired eigenvalue and the next one on that side of the spectra[3]). To be more concrete, the convergence of the Power Method is proportional to $\frac{\lambda_{n-1}}{\lambda_n}$ or $\frac{\lambda_1}{\lambda_2}$, depending on whether we are seeking the biggest or smallest eigenvector. In our particular case, if λ_2 and λ_1 are very close to each other (Clustered Eigenvalues), the ratio $\frac{\lambda_1}{\lambda_2}$ will be big, implying that we need a lot of iterations. This limitation will appear later, when comparing the algorithms.

3.2.2 Krylov Subspaces

The next tool is Krylov Subspaces (see [40] or [43]), which are used to search for approximations of the desired eigenvectors. A Krylov subspace of dimension m, for a given matrix A and generating vector $\mathbf{x_0}$, is defined in (eq. (3.3)).

[2] As it requires that the initial vector has a non zero component coordinate, regarding the dominant eigenvector.

[3] Spectra is a commonly found name for the set of eigenvalues of a matrix.

$$\mathrm{K}_m(A, \mathbf{x}_0) = \mathrm{span}\left\{A^0\mathbf{x}_0\ ,\ A^1\mathbf{x}_0\ ,\ \ldots\ ,\ A^{(m-1)}\mathbf{x}_0\right\} \tag{3.3}$$

Both [40] and [43] have chapters with more properties about Krylov Subspaces, but the intuitive idea is that the Power Method wastes a lot of information: The iteration produces the vectors $A^k\mathbf{x_0}$, but only the last one is actually used. Krylov Subspaces keep all these vectors, and use them to generate a subspace where there are more chances to find good approximations of the eigenvectors.

3.2.3 Rayleigh-Ritz Method

This is another way of extracting eigenvector approximations called the Rayleigh-Ritz Method (see [43]). Assuming that you already have a subspace where you want to search for the approximations of eigenvectors of matrix A, then the high level algorithm looks like this:

- Compute orthonormal basis of that subspace, and arrange it as columns of a matrix called V.
- Solve the (smaller) eigenproblem $R\mathbf{y} = \lambda\mathbf{y} \quad \ni \quad R = V^T AV$.
- Compute the Ritz pairs $(\widetilde{\lambda}_i, \widetilde{\mathbf{x}}_i) = (\lambda_i, V\mathbf{y}_i)$

As you can see on the above steps, in order for this method to work the dimension of this subspace needs to be much smaller than $\dim(A)$, because you are ultimately solving an small eigenproblem anyway. The rationale is that it becomes easier to solve the eigenproblem on a much smaller dimension [4].

Another interesting detail to note, is that the matrix R is something like a projection of the original matrix A on the approximating subspace. The two matrices are pretty much the same thing (similar), except for a change of basis. This implies the spectra of R is a subset of the spectra of A, so the eigenvalues obtained with Rayleigh-Ritz method can be used directly as

[4] As we will see later, algorithms targeting large sparse matrices can ultimately rely on dense matrix algorithms. This is with the premise that they do so, at a much smaller scale.

the desired approximations. The eigenvectors need a change of basis though ($V\mathbf{y_i}$), which takes them from the coordinates used on the approximating subspace to the canonical coordinates where the matrix A operates. More details about the theory behind this method can be found in [43].

3.3 Dense Matrix Algorithms

As mentioned already, the only reason to consider this family of algorithms is to have a reference point. But we just provide references for dense matrix algorithms in this section, as their details fall out of the scope of this thesis (our focus is on sparse matrix methods). The canonical encyclopedic reference for this family is [19], though each algorithm may have more dedicated resources. We mention some of them on the following subsections.

3.3.1 Symmetric Tridiagonal QL Algorithm

The application currently uses the Symmetric Tridiagonal QL Algorithm (see [40]). Like other algorithms from same family, this method converts first the input matrix into tridiagonal form, to compute from there the eigenpairs. Once in this reduced form, the simplest version of the algorithm could be thought as a block version of the Power Method. On each iteration we need to orthogonalize with some matrix factorization (QL) and to multiply the resulting orthogonal vectors by the matrix (represented in a possibly different basis).

Nevertheless, practical implementations have far more points to consider. The reader interested in more details about this QL Algorithm or its cousin the QR Algorithm, can consult standard literature on the topic like [19] or [40]. Additionally we want to mention here that this algorithm has an inherent limitation for our application: It is designed to compute all the eigenpairs. This sounds like a waste of resources, given that we just need one particular eigenpair (the second smallest).

3.3.2 The MRRR Algorithm

The Multiple Relatively Robust Representations Algorithm (MRRR or MR^3, for short), is a more suitable option for computing the second eigenpair [15], [13], [14] or [41]. Is a quite sophisticated procedure, but the only detail we will mention here, is the ability of the algorithm to compute the eigenpairs in isolation. Actually, this algorithm is perhaps the best the application can do, in terms of dense matrix methods.

3.4 Sparse Matrix Algorithms

If we really want to leverage the properties of the Laplacian, we need to consider algorithms which are suitable for sparse matrix representations. They are usually designed to calculate an small portion of the spectra (smallest or biggest), and the reason why they work better with sparse formats is because they do not require an explicit representation of the matrix at any moment. Instead, these algorithms only require a callback mechanism to perform an operation involving the matrix (eg $A\mathbf{x}$). Therefore, the internals of those operations can be optimized to take advantage of the sparse format in question (instead of using a generic logic that assumes a dense representation). We will present a couple of algorithms from this family: A variant of Lanczos and LOBPCG.

3.4.1 Lanczos (IRLM)

Lanczos procedures are a family of algorithms, and the particular variant we are presenting here is called Implicitly Restarted Lanczos Method (IRLM) [33]. The algorithm is iterative (it progressively approximates the desired eigenpairs), and the main idea is to apply the Rayleigh-Ritz method against a Krylov subspace $\mathrm{K}_m(A, \mathbf{x_0}) \ni m > k$. This implies that it needs to solve on each iteration, an small eigenproblem for an $m \times m$ symmetric matrix H. Where m is a bit bigger than the number of desired eigenpairs k, but still much smaller than the dimension of the matrix.

One of the issues with Krylov Subspaces is that we do not know in advance, how big the dimension m needs to be such that the associated subspace contains good eigenvectors approximations. Since such dimension is in function of the number of vectors $A^k\mathbf{x_0}$ that we keep in memory, this can be

a serious problem (as we do not have infinite memory). In order to tackle this issue, the IRLM uses a tool called Implicitly Restarted QR Algorithm to apply p shifts against the matrix H. The algorithm is presented in compact form in (eq. (3.4)).

$$j = 1 \dots p : \; QR = \mathrm{qr}(H - \lambda_j I) \wedge H = \; Q^T \, HQ \quad \ni \quad m = k + p \qquad (3.4)$$

After the shift a truncation with a factor k takes place, and on next iteration the matrix H is filled up again to size m (using powers of the input matrix, among other stuff). At any iteration we have $m = k + p$ eigenpairs from matrix H, where k is the actual number requested by the user and p is the number of extra eigenpairs. The purpose of the shift procedure, is to discard the p vectors from H that do not contribute to the interesting eigenvectors. The definite reference, for all the gory details of this variant of the Lanczos algorithm, can be found at [33].

As far as practical usage of the IRLM routine, we take the following considerations:

- We set $k = 2$ and by default the implementation sets $m = 2k + 1$.
- As we want the smallest eigenpairs, we used shift-invert mode with $\sigma = 0$ [5]. This requires also to use a linear sparse solver. We tested two actually, SuperLU ([12], [34]) and Cholmod ([8], [10]). These solvers use respectively the LU and Cholesky factorizations, hence are considered direct methods (as opposed to iterative linear solvers). While both are specialized for sparse matrices, only Cholmod leverages the Laplacian properties (we will see in the results chapter, how much that affects the performance). The linear solver we pass when invoking the routine in shift-invert mode, accounts for $\approx$ %80 of the execution time. Thus, is among the most important input parameters to consider.

We observed slow convergence for Clustered Eigenvalues, but this was expected (is a known limitation inherited from the Power Method, on which Krylov Subspaces are based on).

[5] Which means we do not want the shift $A - \sigma I$, just the invert mode.

3.4.2 LOBPCG

The second algorithm being considered is the so called Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) [29], [31]. It is also an iterative method that applies Rayleigh-Ritz Method every time (for simplicity we focus on the single vector version). The main difference with Lanczos is the subspace where it searches for the eigenvector approximations, which for LOBPCG is span$\{\mathbf{x}_i, T\mathbf{r}_i, \mathbf{x}_{i-1}\}$. The generating set has 3 vectors only, being respectively the current and previous approximations (x_i, x_{i-1} [6]) plus the preconditioned residual $T\mathbf{r_i}$. The residual vector r_i measures how well x_i approximates the desired eigenvector and is defined in (eq. (3.5)).

$$\mathbf{r}_i = A\mathbf{x}_i - \rho(\mathbf{x}_i)\mathbf{x}_i \ \wedge \ \rho(\mathbf{x}) = \frac{\mathbf{x}^T A\mathbf{x}}{\mathbf{x}^T\mathbf{x}} \tag{3.5}$$

The function $\rho(\mathbf{x})$ is the already familiar Rayleigh-Quotient, and another of its properties justifies the inclusion of the residual r_i. It turns out that $\rho(\mathbf{x})$ has as critical point precisely the smallest eigenpair $(\mathbf{v_1}, \lambda_1)$. Furthermore, its gradient $\nabla(\rho\,\mathbf{x_i})$ is proportional to r_i. This means that r_i is proportional to the direction where the Rayleigh-Quotient Function approximates better the smallest eigenpair. Hence, it makes sense to include such direction when building the linear combinations that will approximate the eigenpair. The details of this algorithm are quite technical, but the reader can consult the original article from its creator Knyazev ([29]). An easier version appears on the PhD thesis of one of Knyazev's students (see [31]).

The reader may have noticed a little inconsistency. Above we talked about the smallest eigenpair but was not the Fiedler Vector part of the second one instead? That is still the case, but LOBPCG has a nice feature that allow us to focus on the smallest eigenpair indeed. When invoking the routine, the user can pass a matrix Y (called constraints matrix) whose columns generate certain subspace. Then the algorithm searches for the approximations only in the orthogonal subspace $Y^{\perp}$. This fits perfectly our problem, as we know that the first eigenvector of the Laplacian is the vector $\mathbf{1}$, and since we are dealing with a symmetric matrix, we know that the rest of the eigenvectors (in particular the Fiedler Vector) are going to live inside $\mathbf{1}^{\perp}$. Thus, we set

[6] In practice $x_i - \beta x_{i-1}$ is used instead of the previous approximation x_{i-1}, as it tends to loose orthogonality respect to x_i.

$Y = \mathbf{1}$, when calling the LOBPCG routine. This makes the search of the smallest eigenpair in $\mathbf{1}^{\perp}$, equivalent to the search of the second smallest on the original space.

Let us summarize now all the practical considerations taken when calling the LOBPCG implementation (which were actually grabbed from the Python Package NetworkX [20]):

- As mentioned, given we set the constraints matrix $Y = \mathbf{1}$, we only need to request the smallest eigenpair (set $k = 1$).
- The matrix T multiplying the residual vector $\mathbf{r_i}$ is called the preconditioner, and its purpose is to accelerate convergence [7]. In practice, and for the data involved with our application, setting $T = \frac{1}{\text{diag}(L)}$ worked pretty well (actually, this is perhaps the most important input parameter for our case, without this preconditioner LOBPCG was even slower than IRLM).
- Rather than slow convergence, the implementation showed numerical errors on Clustered Eigenvalues (for some reason, the dense 3×3 matrix produced at each iteration eventually looses the properties that the code expects). We did not dig further into this problem, but just assumed that we needed to do something to avoid Clustered Eigenvalues on the data, given that both sparse algorithms have problems with them.

[7] Going into the theory behind is out of our scope, but again, the interested reader can consult [29] or the more digested summary in [31].

Chapter 4

The Experiment

4.1 Setup

4.1.1 Hardware

All the tests were conducted on a laptop with hardware and utilities described in table (table 4.1).

Table 4.1: Hardware used on the experiment

Processor	Intel®Core™ *i*5 at 2.40GHz (nowadays considered a commodity processor).
Serial execution enforcement	The Linux *taskset* command, needed in case low level libraries attempted multi-threading (we are talking about BLAS or LAPACK, as the high level algorithms used were serial).
Memory	8Gb of RAM. No more than 512Mb were used on each experiment, and only for dense matrix algorithms (sparse ones used much less) .

4.1.2 Software

The implementations of the algorithms we used, were basically Python wrappers against native libraries. This is a common pattern found in scientific computing, and it tries to combine the best of two worlds:

- On one side, the high efficiency and quality of established native libraries (usually made in C or Fortran).

- On the other side, the high flexibility and productivity of an scripting language like Python.

The mechanism is basically to use the native interface of the high level language (Python in this case), and to bind high level APIs to the low level libraries. This approach should be familiar to the users of Matlab.

The high level Python packages were SciPy [27] and NetworkX [20]. On the other hand, the native libraries actually offering the algorithm implementations were:

- LAPACK [17], which provides (among many other things) the MRRR implementation. LAPACK is actually the industrial standard for dense matrix computations. LAPACK is implemented in Fortran90 [1].

- ARPACK [33], which provides the variant of the Lanczos algorithm that we tested (IRLM). It also relies on LAPACK as a lower layer. ARPACK is implemented in Fortran77.

- Scipy [27], which delivers the implementation of LOBPCG that we tested. While Scipy may also use LAPACK for several lower level operations, is interesting that LOBPCG is the only algorithm that is implemented in Python. This makes reading its code a much easier task, than say, reading the Fortran77 ARPACK.

As it can be perceived, all the algorithm implementations ultimately rely on LAPACK [17], and this in turn, relies on BLAS [32]. All those libraries were installed on the laptop, of course. The SCC algorithm mentioned in (section 4.2.3) is also offered by the Scipy package [2]. Last but not least, same SciPy also provides the sparse formats we tested (CSR and CSC).

[1] At least the opensource version we used, as there are other commercial implementations as well.

[2] The routine is *scipy.sparse.csgraph.connected_component*

Overall, Scipy makes quite accessible a lot of scientific software, it feels like a new generation MATLAB that is becoming more and more popular [3]. Another merit of Scipy, is that its performance is quite close to that of native code (we also tested calling the libraries, like ARPACK, from low level C code). The little penalty in performance by using the Python wrappers, is definitely worth it in terms of development productivity.

4.2 Data Preparation

4.2.1 The actual data

We created 10 random matrices from application's domain data, using the same encoding techniques they use in production. The sizes of such matrices are in the range $[867, 4500]$, so they fit in memory without problem.

4.2.2 Sparse formats

Two sparse matrix formats were tested with IRLM and LOBPCG algorithms. Namely the Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats. For an overview of these and related formats, in the actual context where we experimented with them, the reader can consult [26]. While both formats showed similar performance results, we prefer CSR because is the proper format to use with the Clustered Eigenvalues removal pre-processing (see (section 4.2.3) below).

4.2.3 Avoiding Clustered Eigenvalues

As we mentioned on previous chapter, Clustered Eigenvalues were a headache for both IRLM and LOBPCG. Thus, we needed to do something about them. By researching deeper why they were occurring, we found that graphs behind the Laplacians produced by the application, presented a common pattern: A

[3] A new competitor entered the arena recently, the Julia Programming Language (see [5]), which was specifically designed for scientific computing. We tried this language as well, for some of the experiments, but ended up preferring the maturity and wider repertoire of the Python + SciPy combo.

disconnected graph with an small component (2-3 nodes), and a big component (the rest of the nodes).

The theory says that for a disconnected graph of two components, the first and second eigenvalues of the Laplacian will be zero (see [35]). In practice, what you get instead are two very small numbers, and that is an extreme case of clustered eigenvalues. Meaning, they are very close to each other, because both try to approximate zero.

The solution for this issue was to simply remove the already disconnected small component. This makes sense given that ultimately, the high level operation we want to perform on the graph is a bi-partition (thus, expectation is that the graph is connected). The caveat was to compute the Strongly Connected Components (SCC), and the new weights matrix quite efficiently, such that this pre-processing did not become a performance penalty. The SCC computation can be done efficiently (sub-second) with algorithm documented in [42]. For which we did not dig its internals but just used the implementation available in SciPy. This algorithm/implementation is actually the reason why we prefer CSR format over CSC (if the weights matrix is not passed in CSR format, the routine takes much more time).

The re-computation of the weights matrix W though, required a bit more of thought. It turned out that the CSR format is not very friendly with row/column removal operations (which we need to do, in order to simulate that nodes got removed from the graph). Based on an StackOverflow post [1], we took the idea of using an intermediate sparse sparse format (COO) which allows for faster column/row removals. But at the same time, COO format also allows for fast CSR conversion. The current StackOverflow post has an even faster option published now[4], but the Python code below was good enough for our experiments:

[4] This comment was written on 2016-08-25.

```
1  def split_cc_sparse2(W, cclab):
2      idx_del = np.nonzero(cclab)[0]
3      keep_row = np.logical_not(np.in1d(W.row, idx_del))
4      keep_col = np.logical_not(np.in1d(W.col, idx_del))
5      keep = np.logical_and(keep_row, keep_col)
6      W.data = W.data[keep]
7      W.row = W.row[keep]
8      W.col = W.col[keep]
9      W.row -= np.less.outer(idx_del, W.row).sum(0)
10     W.col -= np.less.outer(idx_del, W.col).sum(0)
11     k = len(idx_del)
12     W._shape = (W.shape[0] - k, W.shape[1] - k)
13     return W
```

The snippet above works as follows:

- The argument *cclab* contains the node labels for the components, namely 0 and 1, for the big and small components respectively.

- Our goal is to eliminate the nodes with label 1 from the weights matrix W. For that, that lines $2-8$ begin by shrinking the index and data arrays (after removal of unwanted columns/rows).

- The lines $9-10$ eliminate the potential gaps on the indices.

- Finally lines $11-12$ truncate the W matrix to the new size.

- The key idea is to do all the operations above in terms of the NumPy [47] array primitives, which are quite efficient.

The overall pre-processing to eliminate the Clustered Eigenvalues showed an average time of 1.2 secs for the biggest matrices, this includes: the SCC computation, the routine above to calculate new weights matrix, its conversion from COO to CSR format and the recalculation of the Laplacian.

There is an alternate approach for LOBPCG, that we did not explore, but that is worth mentioning (as it comes from Knyazev himself, the creator of LOBPCG). Rather than pre-processing the graphs to remove the disconnected component, one could simply increase the k (see (section 3.4.2)) to something not less than the size of the eigenvalues cluster (which seems 2 or 3 for our application). It will be interesting to explore that option, and see how it compares in execution time with our current approach. Actually, a comment, by Strang [46] (When talking about the convergence of the "Block Power Method" [5]), suggests that this may also help IRLM. This is just a quick thought, further research will be needed to confirm it. We did not have time to explore these ideas, but they could be part of a follow up of this thesis.

4.2.4 Shifting the spectra

The Cholmod linear solver ([8], [10]) mentioned on (section 3.4.1), requires that the matrix is Symmetric Positive Definite. However, the Laplacian is not (It is symmetric, but has one zero eigenvalue). Doing a little shift on the eigenvalues ($L + 0.01I$), makes the Laplacian Positive-Definite as required. At the end of the algorithm execution, we can just subtract the same shift from the obtained eigenvalue to get the actual answer. This is not really that relevant, as we mostly want the eigenvector. The eigenvalues are just side products that the algorithm also produces.

4.3 Results

During the time the tests were performed on the laptop, no other user processes were launched. However, as the laptop has a desktop OS (Ubuntu 15.04), there are still background processes that may take resources sporadically during the experiment. Let us remember that we are forcing serial execution of the test program, by launching with the Linux *taskset* command. However, even if the test is "attached" to a single CPU core, other processes may still try compete for the same. Aiming to minimize the effect of those background processes, we executed each test 100 times and took the average elapsed time out of them. Each test consists in running an specific algorithm against an specific matrix, as well as measuring its elapsed time.

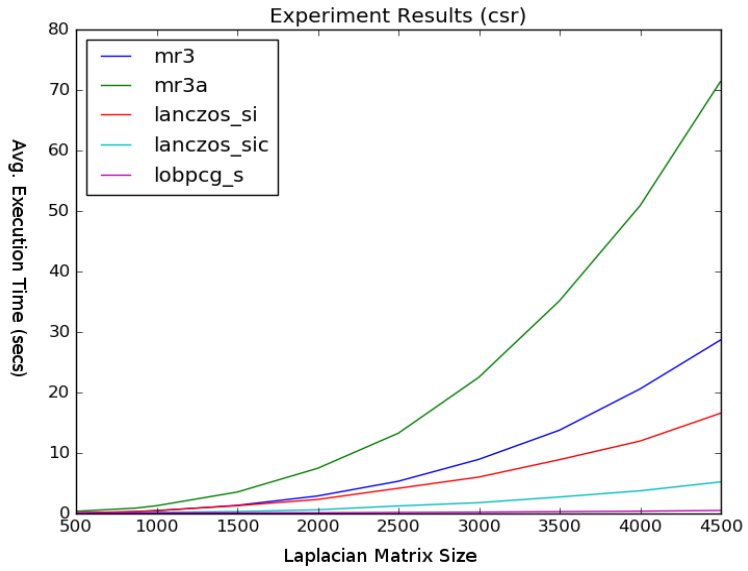
[5] A version of the Power Method that can approximate several eigenvectors.

The total duration of a complete run including all tests, was bigger than 24hrs (this was mainly due inclusion, for comparison reasons, of the slower dense-matrix algorithms).

Even with the average times mentioned above and an apparently idle laptop, there were still minor variations on the results. A more precise mechanism could be to boot the laptop into an special mode where there are less OS admin tasks running on the background. This was not considered critical for our experiment, as the pattern in the execution times was consistent. Only difference detected among complete runs of the test, was a shift of the whole pattern (but LOBPCG, the winner algorithm, consistently showed times in sub-second scale).

The (fig. 4.1) shows the results obtained with the matrices encoded in CSR format. We present a two-dimensional graph, with the X-axis representing the matrix size and the Y-axis the average execution time in seconds.

Figure 4.1: Experiment Results in CSR format



Let us review each line in more detail (we talk mainly about the times for the biggest matrix):

- The green line (label *mr3a*) represents the times of the MRRR algorithm, for computing all the eigenpairs. This is not exactly the same time the application will get today, as the algorithm is different, but it could be considered as a lower bound (given that MRRR is the state of the art for dense matrices). We can see that the time for the biggest matrix goes a bit higher than 70 seconds.

- The blue line (label *mr3*) is also the MRRR algorithm, but taking advantage of its main feature [6]: The ability to compute in isolation just the eigenpair we need. We can see that doing so, reduces the time to a bit less than 30 seconds (for the biggest matrix).

- The orange line (label *lanczos_si*) is the first sparse matrix algorithm. It consists in the Lanczos variant IRLM using SuperLU as the linear solver. Even when such solver is not specialized for the Laplacian matrix, it can reduce the time to a bit less than 20 seconds.

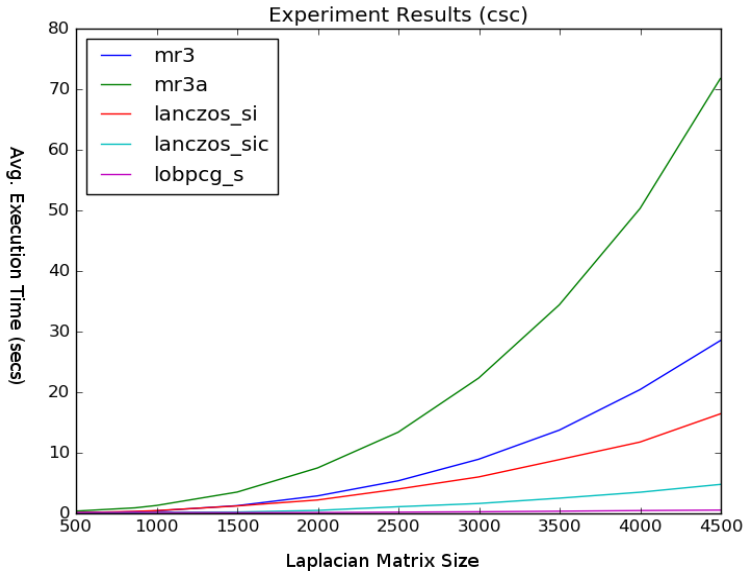
- The sky blue line (label *lanczos_sic*) is the very same IRLM algorithm, but this time using the Cholmod linear solver. We can see that using a more specialized solver, that is optimized for Symmetric Positive Definite matrices, pays off. The time for the biggest matrix goes down to 5 seconds, approximately.

- Putting aside LOBPCG, the Lanczos variant IRLM combined with Cholmod linear solver, has the best time. But such title was taken by LOBPCG, after we discovered how to avoid the exceptions raised on the presence of Clustered Eigenvalues (see (section 4.2.3)). The absolute dominance of LOBPCG can be seen in the purple line (label *lobpcg_s*) of (fig. 4.1). The time goes into sub-second scale even for the biggest matrix (the actual average time is around half a second, sometimes a bit bigger but still within a second). This makes LOBPCG the definite winner of the experiment.

Putting aside the fluctuations mentioned already, the CSC results are basically the same than the CSR ones. A sample execution is (fig. 4.2).

[6] Main feature from the perspective of our problem, as the literature may consider other features of the algorithm, more relevant in a general context.

Figure 4.2: Experiment Results in CSC format



Even when both formats behave well with the sparse algorithms, we prefer CSR for reasons explained already on this chapter (see (section 4.2.3)).

4.4 Why LOBPCG beats IRLM?

The dramatic advantage that LOBPCG shows against Lanczos/IRLM requires an explanation, and we will provide such based on the theory presented in (chapter 3). The (table 4.2) summarizes all the advangates that we see in LOBPCG over Lanczos/IRLM, and the rest of this section will detail each point.

4.4.1 No need for restarting

Given that LOBPCG uses a fixed-size generator set for the searching-subspace, $\text{span}\{\mathbf{x}_i, T\mathbf{r}_i, \mathbf{x}_{i-1}\}$, it does not need to invest extra time ensuring the size remains low. This is contrary to IRLM, which needs to ensure the dimension of $\text{K}_m(L^{-1}, \mathbf{x}_0)$ does not grow too much (causing unbounded memory consumption).

Table 4.2: Lanczos(IRLM) vs LOBPCG

Feature/Issue	*Lanczos(IRLM)*	*LOBPCG (single)*
Need Restarting	Yes, due $\mathrm{K}_m(L^{-1}, \mathbf{x}_0)$	No, span$\{\mathbf{x}_i, T\mathbf{r}_i, \mathbf{x}_{i-1}\}$
Search strategy	Plain search	$\nabla(\rho(\mathbf{x_i}))$ $\ni \mathbf{r}_i = L\mathbf{x}_i - \rho(\mathbf{x}_i)\mathbf{x}_i$
Search Constraints	No	Yes ($Y^{\perp} = \mathbf{1}^{\perp}$)
Requested spectra	$k = 2$	$k = 1$
Matrix operation	solve $L\mathbf{x} = \mathbf{b}$	$L\mathbf{x}$
Uses Preconditioning	Only with iter solvers	Yes $\left(T = \frac{1}{\mathrm{diag}(L)}\right)$
Clustered eigenvalues	Inherited from PM	Bug?

4.4.2 Clever search strategy

While IRLM does not seem to follow a particular search strategy, LOBPCG leverages the gradient of the Rayleigh-Quotient, $\nabla(\rho(\mathbf{x_i}))$, on each iteration. Let us remember that the search space for LOBPCG is span$\{\mathbf{x}_i, T\mathbf{r}_i, \mathbf{x}_{i-1}\}$, where $\mathbf{r}_i$ is the residual vector that measures how well is the current approximation $\mathbf{x}_i$. This residual vector is defined as $\mathbf{r}_i = L\mathbf{x}_i - \rho(\mathbf{x}_i)\mathbf{x}_i$, where $\rho(\mathbf{x}_i)$ is precisely the Rayleigh-Quotient function applied to $\mathbf{x}_i$. Thus, $\rho(\mathbf{x}_i)$ approximates the eigenvalue associated with $\mathbf{x}_i$ (thinking $\mathbf{x}_i$ itself as an approximated eigenvector), and the residual $\mathbf{r}_i$ measures how well $(\rho(\mathbf{x}_i), \mathbf{x}_i)$ complies with the definition of an eigenpair.

The rationale behind including $\mathbf{r}_i$ in the spanning set, is that it is proportional to the gradient of Rayleigh-Quotient function $\nabla(\rho(\mathbf{x_i}))$. The gradient itself generates a one-dimensional space that includes the direction where Rayleigh-Quotient function reaches its minimum (hence approximates better the desired eigenvalue), and that is why it makes sense to include such direction. This was mentioned already in (section 3.4.2), and the references for further details are [29] and [31].

4.4.3 Search constrains / Requested spectra

LOBPCG can focus on searching for the smallest eigenpair, while IRLM needs to compute the first and the second (because IRLM is designed for computing "continuous" sections of the spectra, eg the k smallest ones, see

[33]). Let us recall that LOBPCG routine, see (section 3.4.2), allows the user to pass a matrix Y as input parameter. The columns of such matrix Y span a subspace, and the LOBPCG implementation restricts the search to its orthogonal complement $Y^{\perp}$. This feature fits perfectly our use case, as the Laplacian matrix L has a known first eigenvector $\mathbf{1}$ (the all ones vector associated with eigenvalue zero, see (section 2.1.4)). Therefore, we set $Y = \mathbf{1}$, and allow LOBPCG routine to focus on a single eigenpair instead of two.

4.4.4 Cheaper iterations

One of the main goals in the design of LOBPCG (see [29], [30]), was to be as cheap as the regular invocation of Lanczos algorithms (like IRLM). Regular means here, that we use the algorithm to compute the biggest eigenpairs, as opposed to the use of the shift-invert-mode, which aims to compute the opposite side of the spectra.

Going back to the point, the only matrix-dependent operation that LOBPCG performs on every iteration, is the matrix-vector product $L\mathbf{x}$. Furthermore, such matrix multiplication is likely to be optimized for the sparse format being used (CSR or CSC). This contrasts with the far heavier operation that IRLM needs to execute on each iteration, when called in shift-invert mode: To solve the linear system $L\mathbf{x} = \mathbf{b}$. Even with an state of the art solver based on the Cholesky Factorization ([8], [10]), and even if the factorization is performed just once for all the IRLM iterations, the time spent there is big enough to accommodate several complete executions of LOBPCG. As an example, for the biggest matrix of 4497 entries the algorithm LOBPCG finished in sub-second scale while the IRLM computed the factorization in more than 4 seconds (see (fig. 4.1)).

4.4.5 Use of preconditioner

The IRLM algorithm only allows for preconditioners in the sub-problem of solving the linear system $L\mathbf{x} = \mathbf{b}$, when we opt to use iterative solvers instead of direct-methods [7]. But unless those linear solvers are used to compute high accuracy answers (which may not be easy with an iterative solver), they could

[7] The deal here, is that direct methods are forbidden for huge matrices. This was fortunately not our case, as they fit just fine in memory.

slow down overall convergence of IRLM (see [30]). The situation is different for LOBPCG, which considers a preconditioner T for the eigensolver itself. It is an optional argument, but we found that using it significantly speeds up the execution times. The preconditioner used , $T = \frac{1}{\text{diag}(L)}$, was taken from NetworkX package [20].

4.4.6 Clustered eigenvalues

This is perhaps the only point where we may declare a tie, as both implementations have issues with Clustered Eigenvalues. But LOBPCG still seems to show an advantage here: It does not inherit (at least not directly), the slow convergence that IRLM gets from the Power Method. Instead, the implementation of LOBPCG seems to rather have a defect (exception risen on the presence of Clustered Eigenvalues).

Chapter 5

Conclusions

5.1 Recommendations

Let us remember (see (section 1.2)) that the objective of this thesis, is to recommend a better algorithm for computing the Fiedler Vector (better in the sense of running faster than current option, on a single processor). The context is a real-life application that does Spectral Clustering.

This section provides the actual recommendations, which take into account the potential costs of adopting each of the reviewed algorithms. In further subsections the titles contain the recommendation, summarized in a single statement, and the contents of the subsection detail the rationale behind it.

5.1.1 Prefer Sparse Matrix Algorithms

This is perhaps the most obvious recommendation that comes out of this work: Even if the data fits in memory (as it happens here), it is a waste of CPU resources to compute against the zero values of the Laplacian.

5.1.2 If dense, go with MRRR

If for some reason (like a prohibition to perform drastic code changes), the application needs to go with dense matrices, at least they could use MRRR (the best algorithm available). This can be obtained from opensource imple-

mentations of LAPACK [1], though Java Native Interface (JNI) will be needed to call the routines from Java (which is the applications' language). Alternatively, they can try the Java-Netlib opensource project [38], which offers Java wrappers against BLAS, LAPACK and other native libraries.

5.1.3 If development time is a constraint, consider Lanczos/IRLM

If they can invest more time (probably some weeks), and assuming they are allowed to introduce more drastic changes like the usage of sparse matrices, then Lanczos/IRLM is the way to go. Its ARPACK implementation is also opensource, and also readily available through the Java Netlib project. Of course the recommendation would be to use it in combination the Cholmod linear solver, though that may add more time (in case JNI wrappers do not exist yet).

5.1.4 If speed is a concern, go with LOBPCG

Finally, if they are willing to invest even more time (some months perhaps), and if speed is the main concern, then they can explore the winner of the competition: LOBPCG. Porting the Python code from Scipy into Java [2] may certainly be a non trivial task, but the impact could be minimized if they restrict their attention to the single vector version. Ultimately, that version is all we need to compute the Fiedler Vector (the actual code implements a block version that can approximate multiple vectors, but we do not need that).

Another option to explore, shared by Knyazev [3], is to consider the existing Java ports of LOBPCG. In particular, there is one implementation available at `https://github.com/bahaelaila7/sparse-eigensolvers-java`, which comes from Rico Argentati. He is one of Knyazev's former students and also co-author of BLOPEX [28] (the original native implementation of the LOBPCG).

[1] There are also commercial, hardware optimized versions like Intel's, which could be considered.

[2] An interesting note about this code, is that the algorithm creator himself (Knyazev), helped with its development. This speaks about its quality.

[3] Who kindly replied quickly to this, and other inquiries we had about his algorithm.

5.1.5 Use an specialized algorithm for computing SCC

As explained on the (section 4.2.3), the phenomenon of Clustered Eigenvalues translates into a disconnected graph, and it seems better for the algorithms to prevent such cases as input. The application already detects these cases, but it does so by using the byproducts of the eigenproblem it already solves. But if we want to leverage the more advanced sparse matrix algorithms, is better to use a more specialized method for detecting the disconnected graph, to be called before the eigensolver. In concrete, we propose the one documented in [42] (whose implementation performed good enough in our tests, offering sub-second times for the biggest matrices).

5.2 Additional considerations

5.2.1 Need for sparse format

As explained in previous chapters, we could have used either CSR or CSC formats, but the need to compute the SCC of the underlying graph, made us pick the CSR format (see (section 4.2.3)). If they are going with either IRLM or LOBPCG, this is the format we recommend to use.

The matrix format selection will not come for free, as it brings some development costs. This is because the application currently uses a serial version of the Colt library [24], which does not seem to offer any sparse format. Thus, additional effort will be needed to research which libraries support CSR in Java. And from those found, we will need to seek which one allows a better integration with the Colt library. This integration could be minimal and just become a conversion of format (from the dense one used by Colt, to the sparse CSR).

Let us recall that the sparse matrix algorithms will not care if the caller uses CSR or not, as they do not require an explicit representation of the matrix. However, while implementing the callback mechanisms that compute using the Laplacian (either $L\mathbf{x}$ or solving $L\mathbf{y} = \mathbf{x}$), we should definitely use the sparse format, otherwise the promised gains will not show up.

5.2.2 Port of the algorithm for computing the SCC

The algorithm used for efficiently computing the SCC, along with the code to recalculate the weights matrix (see (section 4.2.3)), is all in Python (probably using other underlying native libraries). Assuming we manage to make the CSR format available to Java, another task to include will be to look for a port of this algorithm (or to implement it using the article [42]). While this task may look non trivial, at least does not require deep knowledge of Numerical Analysis, which should make it tractable by regular programmers [4].

5.3 Future Work

Finally, we would like to dedicate a few lines about some challenges that were found during the elaboration of this project; specially regarding the deadlines we had for its completion. The total duration was 6 months (part-time), therefore, there were several areas that could not be explored due the time limitation. These areas could be further explored as future work.

5.3.1 Numerical Linear Algebra is hard

For the non initiated, meaning students who have not taken advanced courses in Linear Algebra, Numerical Linear Algebra or Numerical Analysis in general, is quite a challenge to grasp the algorithms explanations found in literature. There seems to be a tradition to assume that the reader has this immense background, as many details are omitted. Therefore, one needs to go back to more elementary material to try filling some of the gaps. This makes the overall progress a bit slow, and that is the reason why only three algorithms were presented (more on this below).

We dared to embrace this topic, thinking naively that a previous course project (related to the SVD factorization), would provide the required context, but this was true just partially. The specialization required to fully grasp the theory behind the three algorithms compared in this work, goes

[4] We did not use the word "regular" in a pejorative sense, of course. We just try to make the point that "regularly", programmers working in corporations do not know about Numerical Analysis but rather about more standard Computer Science topics.

far beyond the context acquired during the last year (which includes the last Master's courses and the elaboration of this thesis). That is why we reduced the scope to understand only the main ideas, and to learn how to use in practice the implementations. But given more time, it would be beneficial to take some advanced courses in Numerical Linear Algebra. Then, with the enhanced understanding of the algorithms and its theory, we could review the recommendations and either expand them or better justify them from a theoretical perspective.

5.3.2 There is a zoo of algorithms out there

The three algorithms compared in this thesis, definitely do not represent the entire set of options available. Even if we restrict ourselves to the properties of the Laplacian, and to serial execution, there are several more options that offer opensource implementations to explore. Just to give an idea of the diversity available online, the following is a partial list of additional options we also considered (meaning that we read a bit about the algorithms, that we installed the software and tried it at least once):

- TRACEMIN-fiedler: A parallel algorithm for computing the Fiedler Vector [36] [5].
- Anasazi: Software for the numerical solution of large-scale eigenvalue problems [3].
- Slepc: Scalable and flexible toolkit for the solution of eigenvalue problems [22].
- Primme: preconditioned iterative multimethod eigensolver-methods and software description [45].
- Lanczos/IRLM from ARPACK, but using iterative linear solvers (like those reported in [37] [6]).

All the options from the above list are quite interesting to explore, along with the publications about their theoretical foundations. Except for the last

[5] There is no opensource package for this algorithm, but we contacted directly the author to get a copy of the code used in the article.

[6] In practice we did not find an implementation for the preconditioners mentioned in the article [37], but tried with PyAMG [4]

one (for which we did not find an available implementation), the rest were tried indeed. But we did not have enough time to research more about their inner workings, nor to understand how to use them in practice to compute the Fiedler Vector. If more time becomes available, it would be an interesting follow up for this thesis, to explore the options listed above.

5.3.3 Measure the error in the predictions vs real-life execution

On the recommendations section we mentioned some of the required work, for adopting the algorithms we suggested. This work may not be trivial, and actually require some months of effort. But once such stage gets done, it would be interesting to measure what was the real speedup of the application when computing the Fiedler Vector. Actually, we could take those numbers and build an error graph, showing how good our predictions were regarding the speedup (we claimed for example, that LOBPCG will have sub-second scale for the biggest matrices of 4.5k).

These measurements, along with the related implementation and instrumentation efforts, will be another pending point that we would like to explore as future work.

Bibliography

[1] M Ali. Delete columns of matrix of csr format in python. Stackoverflow.com response to a question, 2015. `http://stackoverflow.com/questions/23966923/delete-columns-of-matrix-of-csr-format-in-python`.

[2] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*, volume 11. Siam, 2000.

[3] Christopher G Baker, Ulrich L Hetmaniuk, Richard B Lehoucq, and Heidi K Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)*, 36(3):13, 2009.

[4] W. N. Bell, L. N. Olson, and J. B. Schroder. PyAMG: Algebraic multigrid solvers in Python v3.0, 2015. `http://www.pyamg.org`.

[5] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[6] David Bindel and Jonathan Goodman. Principles of scientific computing. *New York University, New York*, 2009.

[7] Andries E. Brouwer and Willem H. Haemers. *Spectra of Graphs*. New York, NY, 2012.

[8] Yanqing Chen, Timothy A Davis, William W Hager, and Sivasankaran Rajamanickam. Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):22, 2008.

[9] Jane K Cullum and Ralph A Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations: Vol. 1: Theory*, volume 41. Siam, 2002.

[10] Timothy A Davis. User guide for cholmod: a sparse cholesky factorization and modification package. *Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA*, 2008.

[11] James W Demmel. *Applied numerical linear algebra*. Siam, 1997.

[12] James W Demmel, John R Gilbert, and Xiaoye S Li. *SuperLU user's guide*. Citeseer, 1997.

[13] Inderjit S Dhillon and Beresford N Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and its Applications*, 387:1–28, 2004.

[14] Inderjit S Dhillon, Beresford N Parlett, and Christof Vömel. The design and implementation of the mrrr algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 32(4):533–560, 2006.

[15] Inderjit Singh Dhillon. *A New O n2 Algorithm for the Symmetric Tridiagonal Eigenvalue Eigenvector Problem*. PhD thesis, University of California, Berkeley, 1997.

[16] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak mathematical journal*, 23(2):298–305, 1973.

[17] National Science Foundation and Department of Energy. LAPACK – linear algebra PACKage. http://www.netlib.org/lapack/, 2010. `http://www.netlib.org/lapack/`.

[18] Gene H Golub and Henk A Van der Vorst. Eigenvalue computation in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1):35–65, 2000.

[19] Gene H Golub and Charles F Van Loan. *Matrix computations*. Johns Hopkins University Press, 4th edition, 2012.

[20] A Hagberg, D Schult, and P Swart. Networkx: Python software for the analysis of networks. Technical report, Technical report, Mathematical Modeling and Analysis, Los Alamos National Laboratory, 2005., 2005. `http://networkx.lanl.gov`.

[21] Lars Hagen and Andrew B Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

[22] Vicente Hernandez, Jose E Roman, and Vicente Vidal. Slepc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):351–362, 2005.

[23] Leslie Hogben. *Handbook of linear algebra*. CRC Press, 2006.

[24] Wolfgang Hoschek. The colt distribution: Open source libraries for high performance scientific and technical computing in java. 2002. `http://nicewww.cern.ch/hoschek/colt/index.htm`.

[25] Gao Jing. Clustering (lecture 6): Spectral methods. University Lecture (CSE 601: Data Mining and Bioinformatics), 2013. `http://www.cse.buffalo.edu/~jing/cse601/fa13/materials/clustering_spectral.pdf`.

[26] Robert Johansson. *Numerical Python: A Practical Techniques Approach for Industry*. Apress, 2015.

[27] Eric Jones, Travis Oliphant, and Pearu Peterson. {SciPy}: open source scientific tools for {Python}. 2014.

[28] A Knyazev. Blopex: Block locally optimal preconditioned eigenvalue xolvers. `https://bitbucket.org/joseroman/blopex`.

[29] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[30] Andrew V Knyazev. Modern preconditioned eigensolvers for spectral image segmentation and graph bisection. In *Workshop on Clustering Large Data Sets Third IEEE International Conference on Data Mining (ICDM 2003)*, 2003.

[31] Ilya Lashuk. *On preconditioning for linear equations and eigenvalue problems*. ProQuest, 2007.

[32] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[33] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK user's guide: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. Available from netlib@ornl.gov, 1997.

[34] Xiaoye S Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.

[35] Ulrike Von Luxburg. A tutorial on spectral clustering, 2007.

[36] Murat Manguoglu, Eric Cox, Faisal Saied, and Ahmed Sameh. Tracemin-fiedler: A parallel algorithm for computing the fiedler vector. In *International Conference on High Performance Computing for Computational Science*, pages 449–455. Springer, 2010.

[37] Ángeles Martínez. Tuned preconditioners for the eigensolution of large spd matrices arising in engineering problems. *Numerical Linear Algebra with Applications*, 2016.

[38] Opensource. netlib-java: High performance linear algebra (low level). Github.com. `https://github.com/fommil/netlib-java`.

[39] Tan Pang-Ning, Michael Steinbach, Vipin Kumar, et al. Introduction to data mining. In *Library of congress*, volume 74, 2006.

[40] Beresford N Parlett. *The symmetric eigenvalue problem*, volume 7. SIAM, 1980.

[41] Beresford N Parlett and Christof Vömel. *LAPACK Working Note 163: How the MRRR Algorithm Can Fail on Tight Eigenvalue Clusters*. Computer Science Division, University of California, 2004.

[42] David J Pearce. An improved algorithm for finding the strongly connected components of a directed graph. Technical report, Technical report, Victoria University, Wellington, NZ, 2005.

[43] Youcef Saad. *Numerical methods for large eigenvalue problems*, volume 158. SIAM, 1992.

[44] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[45] Andreas Stathopoulos and James R McCombs. Primme: preconditioned iterative multimethod eigensolver—methods and software description. *ACM Transactions on Mathematical Software (TOMS)*, 37(2):21, 2010.

[46] Gilbert Strang. *Linear algebra and its applications*. Saunders, 3rd edition, 1988.

[47] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[48] Dorothea Wagner and Frank Wagner. Between min cut and graph bisection. In *International Symposium on Mathematical Foundations of Computer Science*, pages 744–750. Springer, 1993.

[49] Herbert S Wilf. An algorithm-inspired proof of the spectral theorem in e n. *The American Mathematical Monthly*, 88(1):49–50, 1981.