HOCHSCHULE MITTWEIDA
UNIVERSITY OF APPLIED SCIENCES

MASTER THESIS

Mr.
Raghuveera Kori

Numerical Comparison of Eigenvalue and Eigenvector Determination by Oja Sanger, Jacobi Rotations and the Power method

2021

Faculty of **Applied Computer and Life Sciences**

MASTER THESIS

Numerical Comparison of Eigenvalue and Eigenvector Determination by Oja Sanger, Jacobi Rotations and the Power method

Author:
Raghuveera Kori

Study Programme:
Applied Mathematics for Network and Data Sciences

Seminar Group:
MA17w1-M

First Referee:
Prof. Dr. Thomas Villmann

Second Referee:
Dr Marika Kaden

Mittweida, July 2021

Thanks to

my parents,
for filling confidence in me and blessing me.

Special thanks to

Prof. Dr. Thomas Villmann,
for the kind support and guidance,
Dr. Marika Kaden,
for proofreading the manuscript,
and giving valuable feedback
throughout the thesis.

Bibliographic Information

Kori, Raghuveera: Numerical Comparison of Eigenvalue and Eigenvector Determination by Oja Sanger, Jacobi Rotations and the Power method, 77 pages, 26 figures, Hochschule Mittweida, University of Applied Sciences, Faculty of Applied Computer and Life Sciences

Master Thesis, 2021

Abstract

Computationally solving eigenvalue problems is a central problem in numerical analysis and as such has been the subject of extensive study. In this thesis we present four different methods to compute eigenvalues, each with its own characteristics, strengths and weaknesses. After formally introducing the methods we use them in various numerical experiments to test speed of convergence, stability as well as performance when used to compute eigenfaces, denoise images and compute the eigenvector centrality measure of a graph.

I. Contents

II. List of Figures

III. List of Tables

List of Symbols

We will denote with

- $\mathbb{N}, \mathbb{Z}$ and $\mathbb{R}$ the sets of natural, integer and real numbers respectively;
- $\mathbb{R}_+$ the set of positive real numbers;
- $a \equiv b \pmod N$ the congruence modulo $N \in \mathbb{N}$ of two numbers $a, b \in \mathbb{R}$ (i.e. $\exists k \in \mathbb{Z}$ s.t. $a - b = kN$);
- $|a|$ the absolute value of an integer or real number;
- $\mathbb{R}^n$ the $\mathbb{R}$-vector space of dimension $n \in \mathbb{N}$;
- $||\cdot||$ the norm on an $\mathbb{R}$-vector space V, i.e. a function $f : V \to \mathbb{R}$ s.t.
  - $\forall x, y \in V \quad f(x+y) \leq f(x) + f(y)$
  - $\forall x \in V, \forall s \in \mathbb{R} \quad f(sx) = |s| f(x)$
  - $\forall x \in V$ $f(x) \geq 0$ and $f(x) = 0 \Longleftrightarrow x = 0$;
- $\{a_{ij}\}_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}} \subset \mathbb{R}^{m \times n}$ an $m \times n$ matrix with i, j-th entry equal to $a_{ij} \in \mathbb{R}$;
- A_{ij} the i, j-th entry of a matrix $A \in \mathbb{R}^{m \times n}$;
- $A_{\cdot j}$ the j-th column of a matrix $A \in \mathbb{R}^{m \times n}$;
- $A_{i\cdot}$ the i-th row of a matrix $A \in \mathbb{R}^{m \times n}$;
- A^T the transpose of a matrix $A \in \mathbb{R}^{m \times n}$ for $m, n \in \mathbb{N}$;
- A^{-1} the inverse of a matrix $A \in \mathbb{R}^{m \times n}$ for $m, n \in \mathbb{N}$, if this exists;
- $|A|$ the determinant of a matrix $A \in \mathbb{R}^{m \times n}$;
- $tr(A)$ the trace of a square matrix $A \in \mathbb{R}^{n \times n}$, i.e. $tr(A) = \sum_{i=1}^{n} A_{ii}$
- δ_{ij} for $i, j \in \mathbb{Z}$ the so-called Kronecker-delta defined by $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$;
- $diag(d_1, d_2, \ldots, d_n) = D \in \mathbb{R}^{n \times n}$ a so-called *diagonal* matrix with elements $d_1, d_2, \ldots, d_n$ on the diagonal, i.e. $D_{ij} = d_i \delta_{ij}$ for $i = 1, 2, \ldots, n,\ j = 1, 2, \ldots, n$;
- $I_n \in \mathbb{R}^{n \times n}$, or simply with I if n is clear from the context, the so-called identity matrix $diag(1, 1, \ldots, 1)$;
- $\{a_n\}_{n \in \mathbb{N}} \subseteq S$ a succession in a set S, i.e. a function $f : \mathbb{N} \to S$ s.t. $f(n) = a_n$; we will also simply denote the succession with $\{a_n\}_{n \in \mathbb{N}}$ if it is obvious from the context it has values in S;
- $a_n \to \alpha$ the convergence of the succession $\{a_n\}_{n \in \mathbb{N}} \subset \mathbb{R}$ to $\alpha \in \mathbb{R}$, i.e. $\forall \varepsilon \in \mathbb{R}, \varepsilon > 0, \exists N \in \mathbb{N}$ s.t. $\forall n > N$ $|a_n - \alpha| < \varepsilon$;
- $v_n \to w$ the component-wise convergence of the succession $\{v_n\}_{n \in \mathbb{N}} \subset \mathbb{R}^k$ to $w \in \mathbb{R}^k$, i.e. $\forall i = 1, 2, \ldots, k$ $v_{ni} \to w$ where $v_n = (v_{n1}, v_{n2}, \ldots, v_{nk})$.

1 Introduction

Given a matrix $A \in \mathbb{R}^{m\times n}$, a number $\lambda \in \mathbb{R}$ and a vector v are called *eigenvalue* and *eigenvector* of A if $Av = \lambda v$. Why is this simple definition so important? In first instance one could make the argument that eigenvalues are intrinsic properties of the linear map associated to A, since they are invariant under a change of basis, and thus independent of the particular reference we choose in $\mathbb{R}^n$. Because of this and other fundamental properties, their study is thus certainly worthwhile from a theoretical point of view, and in fact they always occupy a big share of the first lectures one receives in Linear Algebra.

Secondly, they have extensive applications in applied mathematics, physics, engineering, and really any science that at any point has to use linear operators. To give just a few examples, they are used in the theory of any Hamiltonian system such as those that appear in quantum mechanics: the diagonalization of the Hamiltonian operator is used to describe how a system evolves through time. They can be used to discern between attractive and non-attractive equilibriums in the theory of dynamical systems, where arbitrary differential equations are locally approximated with linear operators. Finally, and we shall see and example of this in Chapter 4 of this Thesis, they can be used to compute the principal component analysis of an arbitrary data set, allowing one to achieve a good low-dimensional approximation of the data.

In this thesis we will present 4 numerical methods to solve the eigenvalue problem: Oja, Oja-Sanger, Jacobi and the Power method. Each of these have different origin and can be applied to different classes of matrices. Oja and Oja-Sanger's in particular have an interesting history, which we will give a brief overview of in Section 3.1.1: they were born in the context of simple neural networks as a rule to update the weights that had better behavior than the previously standard Hebbian rule.

The structure of this thesis is as follows: in Chapter 2 we will introduce some notation as well as some fundamental results from linear algebra, statistics and graph theory that we will need in the main part of the thesis. In Chapter 3 we will introduce the eigenvalue methods themselves, discussing in detail the main properties, the convergence, the convergence rate as well as the algorithms and their complexities. For each method we will further manually compute a few iterations on a small matrix.

Finally in Chapter 4 we will put the methods to the test in five different contexts: we will compare the convergence rate on a randomly generated matrix, we will then add noise to this matrix to test the stability, we will use them to compute the so-called eigenfaces, to denoise images and finally to compute the eigenvector centrality of a graph.

2 Preliminaries

In this Chapter we want to briefly establish some notation and fundamental results that we will need in the main Chapters of the Thesis.

Given a norm $||\cdot||$ on $\mathbb{R}^n$, a vector $v \in \mathbb{R}^n$ is of *unitary norm* if $||v|| = 1$; the *normalization* of v is $\frac{1}{||v||}v$, which is always of unitary norm. Two vectors $v, w \in \mathbb{R}^n$ are orthogonal if $v^T w = 0$. Vectors $v_1, v_2, \dots, v_n \in \mathbb{R}^n$ are said to be *orthonormal* if they are all of unitary norm and orthogonal with one another, i.e. $(v_i)^T v_j = 0$ for all $i, j = 1, 2, \dots, n$, $i \neq j$.

We will say a matrix $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A = A^T$; we will say it is *orthogonal* if $AA^T = A^T A = I$, or equivalently if it is invertible and its inverse is also its transpose.

Two matrices A and B in $\mathbb{R}^{n\times n}$ are said to be *symmetric* if there exists an invertible $S \in \mathbb{R}^{n\times n}$ such that $A = S^{-1}BS$.

Matrix norms

Given a matrix $A \in \mathbb{R}^{m\times n}$ an *induced matrix norm* $||\cdot||$ is a norm on $\mathbb{R}^{m\times n}$ s.t.

$$||A|| = \sup_{\substack{x\in\mathbb{R}^n \\ x\neq 0}} \frac{||Ax||}{||x||} \ . \tag{2.1}$$

Remark 2.1 An induced norm always satisfies the *sub multiplicative property*:

$$\forall A \in \mathbb{R}^{m\times n} \ , \ \forall B \in \mathbb{R}^{n\times k} \quad ||AB|| \leq ||A||\,||B|| \ . \tag{2.2}$$

Proof: From the definition of induced norm we have that for all non-zero $y \in \mathbb{R}^n$, $||A|| \leq \frac{||Ay||}{||y||}$ and for all non-zero $x \in \mathbb{R}^k$, $||AB|| \leq \frac{||ABx||}{||x||}$. Therefore, given $x \in \mathbb{R}^k$ s.t. $Bx \neq 0$, we have

$$\begin{aligned} ||AB|| &\leq \frac{||A(Bx)||}{||x||} \\ &= \frac{||A(Bx)||}{||Bx||}\frac{||Bx||}{||x||} \\ &\leq ||A||\,||B|| \ . \end{aligned}$$

□

For $p \in [1, \infty]$ the *p-norm* of a vector $v = (v_1, v_2, \dots, v_n) \in \mathbb{R}^n$ is defined as $||v||_p =$

$(\sum_{i=1}^{n}(v_i)^p)^{\frac{1}{p}}$ if $p \neq \infty$ and $||v||_\infty = \max_{i=1,2,\dots,n}|x_i| =$. For $p=2$ we have the *Cauchy-Schwarz* inequality for two vectors $u, v \in \mathbb{R}^n$:

$$|u^T v| \leq ||u||_2 \, ||v||_2 \ . \tag{2.3}$$

The induced p-norm for a matrix $A \in \mathbb{R}^{m\times n}$ is denoted with $||A||_p$ i.e.

$$||A||_p = \sup_{\substack{x\in\mathbb{R}^n \\ x\neq 0}} \frac{||Ax||_p}{||x||_p} \ . \tag{2.4}$$

In the special case of $p = 1, \infty$ the p-norms of a matrix $A \in \mathbb{R}^{m\times n}$ can be equivalently defined as

$$||A||_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |A_{ij}| \tag{2.5}$$

$$||A||_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |A_{ij}| \tag{2.6}$$

respectively.

We will also use the *Frobenius norm* $||\cdot||_F$ which is defined for a matrix $A = \{a_{ij}\}_{\substack{1\leq i\leq m \\ 1\leq j\leq n}} \in \mathbb{R}^{m\times n}$ as $||A||_F = \left(\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2\right)$.

Computational cost and big-O notation

We will define the *computational cost* of an algorithm as the number of elementary machine operations required for its completion - this is usually given as a function of one of the inputs of the algorithm. When we talk about computational cost we will often use the so called *big-O* notation: given two successions $\{a_n\}_{n\in\mathbb{N}} \subset \mathbb{R}$ and $\{b_n\}_{n\in\mathbb{N}} \subset \mathbb{R}_+$, we will say that a is $O(b_n)$ if $\exists N \in \mathbb{N}, K \in \mathbb{R}$ s.t. $\forall n > N \ |a_n| \leq K b_n$. Intuitively this means that asymptotically $\{|a_n|\}_{n\in\mathbb{N}}$ cannot diverge faster than $\{b_n\}_{n\in\mathbb{N}}$.

2.1 The Eigenvalue Problem

In this thesis we will consider various numerical methods to solve the real *eigenvalue problem*: given a matrix $A \in \mathbb{R}^{n\times n}$, find a number $\lambda \in \mathbb{R}$ and a vector $v \in \mathbb{R}^n$, $x \neq 0$ with

$$Av = \lambda v \ . \tag{2.7}$$

Such λ and v are called *eigenvalue* and *eigenvector* of A respectively. Eigenvalues and eigenvectors are fundamental concepts in linear algebra and extensively treated in books such as [19], [16] and [8].

Proposition 1 *If $A \in \mathbb{R}^{n\times n}$ is invertible and has eigenvalue λ with eigenvector v, then*

1. *$\alpha\lambda$ is an eigenvalue of αA with same eigenvector;*
2. *λ^{-1} is an eigenvalue of A^{-1} with the same eigenvector;*
3. *if $B = S^{-1}AS$, where $S \in \mathbb{R}^{n\times n}$ is invertible, then B has also λ as eigenvalue.*

Proof: For the first part, we have $(\alpha A)v == \alpha(Av) = \alpha\lambda v$. For the second we have

$$\begin{aligned} A^{-1}v &= A^{-1}\frac{1}{\lambda}(\lambda v) \\ &= \frac{1}{\lambda}A^{-1}(Av) \\ &= \frac{1}{\lambda}v\,. \end{aligned}$$

Finally, we have

$$\begin{aligned} B = (S^{-1}AS) &\Rightarrow BS^{-1} = S^{-1}A \\ &\Rightarrow B(S^{-1}v) = \lambda S^{-1}v\,, \end{aligned}$$

i.e. $S^{-1}v$ is eigenvector of B with eigenvalue v. □

We recall that a homogeneous linear system $Ax = 0$, for $A \in \mathbb{R}^{m\times n}$, has only the zero solution $x = 0$ if and only if $|A| \neq 0$ or equivalently if and only A is invertible. Using this basic fact we can define a polynomial with coefficients in $\mathbb{R}$ that has the eigenvalues of A as all and only roots: the *characteristic polynomial* $p(t)$ of real variable $t \in \mathbb{R}$ of a matrix $A \in \mathbb{R}^{n\times n}$ is defined as $p(t) = |A - tI|$, and we have that $p(t) = 0$ if and only if there exists v s.t. $(A - tI)v = 0$, i.e. if and only if there exists an eigenvector v of A with eigenvalue t.

Matrix diagonalization

We will say that a matrix $A \in \mathbb{R}^{n\times n}$ is 6 *diagonalizable* if there exists an invertible matrix P s.t. the matrix $P^{-1}AP$ is a diagonal matrix. In the following proposition we state fundamental properties of diagonalizable matrices, for proofs of which see [19] or [16].

Proposition 2 *Let $A \in \mathbb{R}^{n\times n}$, then:*

- *A is diagonalizable if and only if there exists $u_1, u_2, \ldots, u_n \in \mathbb{R}^n$ that form a basis of $\mathbb{R}^n$ and that are also eigenvectors of A;*
- *if A is symmetric, then it is diagonalizable with $A = PDP^{-1}$, its eigenvalues are all real, its eigenvectors can be chosen to be an orthonormal basis of $\mathbb{R}^n$ and P can be chosen to be orthogonal.*

2.2 Numerical stability and Conditioning

In this section we want to give a brief overview of the topics of numerical stability and conditioning in so far as they concern the eigenvalue problem. A much more in-depth treatment can be found for example in [3] and [18].

Suppose we have a given problem and a space X whose elements are specific instances of this problem, and suppose we have a space of solutions Y. Suppose both X and Y are equipped with a norm and suppose there is a function $f : X \to Y$ that we interpret as the function associating a specific problem $x \in X$ to its exact solution $y = f(x)$; furthermore suppose we have an algorithm $f^\star : X \to Y$ which attempts to approximate f. Let $x \in X$ be a specific problem we wish to solve and define $y^\star = f^\star(x)$ as the approximate solution proposed by the algorithm: in general this will not be the same as the exact solution, i.e. $y^\star \neq f(x)$. Let then $\Delta x \in X$ be such that $f(x+\Delta x) = y^\star$, i.e. $x+\Delta x$ represents the problem that $f^\star$ actually solved. We can define the *forward error* as

$$\mathscr{F}_x = \frac{||y^\star - y||}{||y||},$$

and the *backward error* as

$$\mathscr{B}_x = \frac{||\Delta x||}{||x||}.$$

The forward error is the relative error committed in the output of the algorithm, while the backward error is the relative error in the inputs when considering the instance of the problem the algorithm actually solved. An algorithm is called *backwards stable* or simply *stable* if the backward error is small for all x.

It is useful to introduce $\boldsymbol{K}_x$, the *condition number* associated to the specific problem x, which is the smallest possible number such that

$$\mathscr{F}_x \leq \boldsymbol{K}_x \mathscr{B}_x \tag{2.8}$$

holds independently of the chosen algorithm $f^\star$ (see [5] for a more formal definition). If we are able to determine the condition number for a given problem, we then have a measure of which specific instances of X it is possible to solve accurately. In fact, if $\boldsymbol{K}_x$ is small, then a small perturbation in the inputs of the algorithm (i.e. a small $\mathscr{B}_x$) will result in a small error in the outputs (i.e. small $\mathscr{F}_x$). We will use equation 2.8 in Section 4.2 to estimate the stability of various eigenvalue methods.

For example, consider the linear system problem of finding $x \neq 0$ such that $Ax = b$ for a given invertible matrix $A \in \mathbb{R}^{m\times n}$ and vector $b \in \mathbb{R}^n$. Here we would have $X = \mathbb{R}^{m\times n\times n}$, $Y = \mathbb{R}^n$ and $f(A,b) = A^{-1}b$. With these hypothesis we have

Proposition 3 *Given the above hypothesis, consider the perturbation $\delta b \in \mathbb{R}^n$, and let $\delta x \in \mathbb{R}^n$ be such that $x + \delta x$ is a solution to the perturbed system, i.e.*

$$A(x + \delta x) = b + \delta b \, . \tag{2.9}$$

Then we have

$$\frac{||\delta x||}{||x||} \leq \left|\left|A^{-1}\right|\right| ||A|| \frac{||\delta b||}{||b||} \, . \tag{2.10}$$

Proof: From 2.9 and from $Ax = b$ it follows $A\delta x = \delta b$. Using the sub multiplicative property for induced norms we have

$$\begin{aligned} ||\delta x|| &= \left|\left|A^{-1}\delta b\right|\right| \\ &\leq \left|\left|A^{-1}\right|\right| ||\delta b|| \, , \end{aligned} \tag{2.11}$$

and

$$\begin{aligned} ||b|| &= ||Ax|| \\ &\leq ||A|| \, ||x|| \, , \end{aligned}$$

which implies

$$\frac{1}{||x||} \leq \frac{||A||}{||b||} \, . \tag{2.12}$$

The thesis then follows from 2.11 and 2.12. □

In [3] an analogous result (albeit with a more technical proof) is shown for when a perturbation $\Delta A \in \mathbb{R}^{m\times n}$ is considered on A. We can conclude that in the case of solving a linear system $Ax = b$ the condition number can be defined as $\boldsymbol{K} = ||A|| \left|\left|A^{-1}\right|\right|$; we will denote this particular condition number with $\boldsymbol{K}(A)$.

For the eigenvalue problem 2.7 we have $X = \mathbb{R}^{n\times n}$, $Y = \mathbb{R}^{1\times n}$, $f(A) = (\lambda, v)$. In order to obtain the condition number for the eigenvalue problem, we need the following which is known as Bauer-Fike Theorem and its Corollary:

Theorem 2.2 *Let $||\cdot||$ be an induced matrix norm such that $||D|| = \max_{1\leq i\leq n} |d_i|$ holds for any diagonal matrix $D = diag(d_1, d_2, \dots, d_n)$, let $A \in \mathbb{R}^{n\times n}$ be a diagonalizable matrix, i.e. there is an invertible matrix Q s.t. $Q^{-1}AQ = D = diag(\lambda_1, \lambda_2, \dots, \lambda_n)$ where $\lambda_1, \lambda_2, \dots, \lambda_n$ are the eigenvalues of A. Furthermore let $\Delta A \in \mathbb{R}^{n\times n}$ be any other $n\times n$ matrix and let μ be an eigenvalue of $A + \Delta A$. Then there exists λ eigenvalue of A s.t.*

$$|\mu - \lambda| \leq \boldsymbol{K}(Q)\,||\Delta A|| \;. \tag{2.13}$$

Proof: If $\mu = \lambda_{\tilde{i}}$ for some $\lambda_{\tilde{i}}$ then the thesis is trivial. Suppose thus $\mu \neq \lambda_i$ for all i, and let $v \in \mathbb{R}^n$ be s.t. $(A + \Delta A)v = \mu v$; then we have

$$\begin{aligned}\Delta A v &= (\mu I - A)v \\ &= (\mu I - QDQ^{-1})v \\ &= Q(\mu I - D)Q^{-1}v \;.\end{aligned}$$

By multiplying the first and last terms in the previous equation by Q^{-1} on the left we obtain

$$\begin{aligned}(\mu I - D)Q^{-1}v &= Q^{-1}\Delta A v \\ &= (Q^{-1}\Delta A Q)Q^{-1}v \;.\end{aligned}$$

Now we multiply on the left by $(\mu I - D)^{-1}$ and obtain

$$Q^{-1}v = (\mu I - D)^{-1}Q^{-1}\Delta A Q(Q^{-1}v) \;,$$

and using the sub multiplicative property for induced matrix norms on square matrices we have

$$||Q^{-1}v|| \leq ||(\mu I - D)^{-1}||\,||Q^{-1}\Delta A Q||\,||Q^{-1}v|| \;. \tag{2.14}$$

Since $(\mu I - D) = diag(\mu - \lambda_1, \mu - \lambda_2, \dots, \mu - \lambda_n)$, its inverse is

$$(\mu I - D)^{-1} = diag((\mu - \lambda_1)^{-1}, (\mu - \lambda_1)^{-2}, \dots, (\mu - \lambda_n)^{-1})$$

and thus

$$||(\mu I - D)^{-1}|| = \max_{1\leq i\leq n} |(\mu - \lambda_i)^{-1}| \;.$$

We can then divide both terms in 2.14 by $\left|\left|Q^{-1}v\right|\right|$ and obtain

$$\begin{aligned}1 &\leq \max_{1\leq i\leq n} |(\mu-\lambda_i)^{-1}| \left|\left|Q^{-1}\Delta A Q\right|\right| \\ &\leq \max_{1\leq i\leq n} |(\mu-\lambda_i)^{-1}| \left|\left|Q^{-1}\right|\right| ||Q|| \, ||\Delta A|| \\ &= \max_{1\leq i\leq n} |(\mu-\lambda_i)^{-1}| \boldsymbol{K}(Q) \, ||\Delta A|| \ ,\end{aligned}$$

and thus

$$\begin{aligned}\min_{1\leq i\leq n} |\mu-\lambda_i| &= \frac{1}{\max_{1\leq i\leq n} |\mu-\lambda_i|} \\ &\leq \boldsymbol{K}(Q) \, ||\Delta A|| \ .\end{aligned}$$

□

Remark 2.3 Examples of matrix norms satisfying the hypothesis on diagonal matrices are $||\cdot||_1\, , ||\cdot||_2$ and $||\cdot||_\infty$.

Corollary 2.4 *Under the same hypothesis as for Theorem 2.2, there exists λ eigenvalue of A s.t.*

$$\frac{|\lambda-\mu|}{|\lambda|} \leq \boldsymbol{K}(Q) \left|\left|A^{-1}\Delta A\right|\right| \qquad (2.15)$$

Proof: Let v be s.t. $(A+\Delta A)v = \mu v$, then

$$A^{-1}(A+\Delta A)v = \mu A^{-1} v \ .$$

Define now $M = \mu A^{-1}$ and $N = -A^{-1}\Delta A$; we have

$$\begin{aligned}(M+N)v &= \mu A^{-1}v - A^{-1}\Delta A v \\ &= A^{-1}(A+\Delta A)v - A^{-1}\Delta A v \\ &= v \ ,\end{aligned}$$

i.e. v is an eigenvector of eigenvalue 1 for $M+N$. Note that if $\lambda_1, \lambda_2, \ldots, \lambda_n$ are the eigenvalues of A, then because of Proposition 1 the eigenvalues of M are $\mu/\lambda_1, \mu/\lambda_2, \ldots, \mu/\lambda_n$. Applying Theorem 2.2 to $M+N$ then gives us that there must exist λ eigenvalue of A

s.t.

$$\begin{aligned} \frac{|\lambda - \mu|}{|\lambda|} &= \left|\frac{\mu}{\lambda} - 1\right| \\ &\leq \boldsymbol{K}(Q)\,||N|| \\ &= \boldsymbol{K}(Q)\,\left|\left|A^{-1}\Delta A\right|\right| \ . \end{aligned}$$

□

From Corollary 2.4, we can deduce that for the eigenvalue problem 2.7 for a diagonalizable matrix $A = Q^{-1}AQ$ we can define the conditioning number as $\boldsymbol{K}(Q) = ||Q||\,||Q^{-}1||$.

2.3 Covariance matrix and Principal Component Analysis

We want here to establish some basic notions of statistics that we will need in the main Chapters of the thesis - these are all subjects that are extensively treated in introductory books to probability and statistics, such as [13].

Let $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_d)$ be a random vector with values in $\mathbb{R}^d$ and suppose we have N independent samples $x^1, x^2, \ldots, x^N \in \mathbb{R}^d$ that are realizations of $\boldsymbol{x}$; this means they are empirical measurements of $\boldsymbol{x}$ obtained by some experiment. In Section 4.3 we will deal with a data set composed of gray-scale face images, i.e. each face is an element in $[0,1]^d$ for some d. In that case $\boldsymbol{x}$ will be "a random picture of a human face" and $x^1, x^2, \ldots, x^N$ will be the actual faces in the data set.

We will denote the mean of $\boldsymbol{x}$ with $\mathbb{E}[\boldsymbol{x}]$ and to simplify computations we will always suppose $\boldsymbol{x}$ to be *centered*, i.e. $\mathbb{E}[\boldsymbol{x}_1] = \mathbb{E}[\boldsymbol{x}_2] = \ldots = \mathbb{E}[\boldsymbol{x}_d] = 0$. We define the *sample mean* $\overline{X} = \frac{1}{N}\sum_{i=1}^{N}(x^i)$, where with $(x^i)_j$ we mean the j-th component of x^i. Because we are supposing $\boldsymbol{x}$ to be centered, we will also suppose for the sample mean to be 0 for all components, i.e.

$$\forall j = 1, 2, \ldots, d \quad \frac{1}{N}\sum_{i=1}^{N}(x^i)_j = 0 \ .$$

We further define the *data matrix* associated to the N independent samples of $\boldsymbol{x}$ as the matrix $X \in \mathbb{R}^{N \times d}$ having x^i as i-th row; $(x^i)_j$ is thus also the entry i, j of the data matrix.

For the centered random vector $\boldsymbol{x}$, the *covariance* is defined as $\mathscr{C} = \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^T] \in \mathbb{R}^{d \times d}$; the k,l-th entry of $\mathscr{C}$ is given by the random variable $\mathbb{E}(\boldsymbol{x}_k \boldsymbol{x}_l)$. The *sample covariance* is defined as the matrix $C \in \mathbb{R}^{d\times d}$ matrix whose k,l-th entry is given by $\frac{1}{N-1}\sum_{i=1}^{N}(x^i)_k(x^i)_l$. We can write

$$C = \frac{1}{N-1}X^TX \ , \tag{2.16}$$

in fact consider the k,l-th entry of X^TX: this is the scalar product of the k-th and l-th column of X, i.e.

$$(X^TX)_{k,l} = \sum_{i=1}^{N}(x^i)_l(x^i)_k \ .$$

Observe that both $\mathscr{C}$ and C are real symmetric matrices and thus have a set of orthonormal eigenvectors. Let $\lambda_1,\lambda_2,\ldots,\lambda_d$ be the eigenvalues of $\mathscr{C}$ and suppose $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_d|$; the corresponding orthonormal eigenvectors $u_1,u_2,\ldots,u_d$ are called *principal components* of $\boldsymbol{x}$. In practice we are only able to compute eigenvectors of C because we have only access to the samples of the random vector; we will call these *principal components* of the data in X. It can be seen (for example [17]) that one can equivalently define the principal components as the orthonormal directions along which the data has the maximum variance, i.e.

$$\begin{aligned}
u_1 &= \underset{||w||=1}{\arg\max}\, \mathbb{V}[Xw] \\
u_2 &= \underset{\substack{||w||=1 \\ w^Tu_1=0}}{\arg\max}\, \mathbb{V}[Xw] \\
&\vdots \\
u_d &= \underset{\substack{||w||=1 \\ w^Tu_1=w^Tu_2=\ldots=w^Tu_n=0}}{\arg\max}\, \mathbb{V}[Xw] \ ,
\end{aligned}$$

where with $\mathbb{V}[Xw] = \mathbb{E}[(Xw-\mathbb{E}[Xw])^2]$ we indicate the variance of the vector Xw, which has as components the projections of the data samples (rows of X) onto the direction w.

2.3.1 Principal Component Analysis for dimension reduction

Principal Component Analysis is often used for *dimension reduction*, i.e. for obtaining a lower-dimensional approximation of an input data set. Suppose $x^1,x^2,\ldots,x^N \in \mathbb{R}^d$ are independent samples from a random vector $\boldsymbol{x} \in \mathbb{R}^d$, suppose $X \in \mathbb{R}^{N\times d}$ is the associated data matrix and $C \in \mathbb{R}^{d\times d}$ the associated sample covariance matrix with eigenvalues $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_d|$ and corresponding eigenvectors $u_1,u_2,\ldots,u_d$.

If we choose $K \in \mathbb{N}$, $K < d$, we can define a K-dimensional approximations of each data sample x^i as the vector $x^i_{|K} = \big((x^i)^T u_1, (x^i)^T u_2, \dots, (x^i)^T u_K\big) \in \mathbb{R}^K$. In fact, knowing this vector we can reconstruct the projection of x^i onto the subspace generated by the first K principal components, which is defined as

$$\pi^K(x^i) = \sum_{j=1}^{K} ((x^i)^T u_j) u_j \ \in \mathbb{R}^d \ .$$

Even for low values of K, $\pi^K(x^i)$ is generally a good approximation of x^i because of the property that the first principal components encapsulate most of the variance in the data. The higher the value of K the better the approximation is, the trade-off being we need to compute and store more scalar products to obtain $x^i_{|K}$ or equivalently $\pi^K(x^i)$. If $K = d$ then $\pi^K(x^i) = x^i$.

2.4 Graphs and eigenvector centrality

In this Section we want to give a brief introduction to the notion of graphs and specifically eigenvector centrality; we will need these in Section 4.5. The main reference in this Section will be [10].

Graphs are used in the study and analysis of complex networks, such as those derived by social media, scientific paper citations or computer networks. Each of these examples is described in graph theory as a set of vertices (social media users, scientific authors, server computers) and a set of edges that encodes some form of relation between vertices (being a follower, citing another author, being connected).

Formally, a graph is a structure describing the relations between certain entities; specifically we will say G is a *directed graph* if $G = (V,E)$, where $V = \{v_1, v_2, \dots, v_n\}$, $n \in \mathbb{N}$, is a set whose elements are known as *vertices* and $E \subseteq V \times V$ is a set of pairs of vertices whose elements are known as *edges*. The graph is directed because the pairs that constitute edges are ordered - for non-directed graphs the pairs are unordered. We will be exclusively interested in directed graphs, which we will simply call graphs for brevity.

In the study of complex networks an important concept is that of centrality: a *centrality measure* is a function $\mu : V \to \mathbb{R}_+$ that assigns a score to each vertex, where vertices that are more important should receive a higher score. There are various competing definitions for centrality measures, the simplest of which being that of *degree centrality*. The degree of a vertex $v \in V$ is defined as the number of incoming edges in v, i.e. $|\{w \in V \mid (w,v) \in V\}|$, and the degree centrality measure simply associates this number to each vertex. This means that, according to the degree centrality, the most important vertices are the ones with most incoming edges. While this may be a good enough measure in some situations, in most applications it is quite crude and not very useful in identifying important vertices. In fact typically in a network different vertices have different importance, and simply counting the number of incoming edges does not account for the fact that an incoming edge from a vertex that is itself important should count more.

An attempt to solve this problem is to require that the measure on any node $v \in V$ be proportional to the sum of the measures for all other nodes in V connected to v, i.e.

$$\forall v \in V \quad \mu(v) = K \sum_{\substack{w \in V \\ (w,v) \in E}} \mu(w) \tag{2.17}$$

for some $K \in \mathbb{R}$. This requirement on μ can be translated into an eigenvector problem; to see this we need to define the *adjacency matrix* $A \in \{0,1\}^{n \times n}$ associated to the graph G, whose i,j-th element A_{ij} is defined by

$$\forall i,j = 1,2,\ldots,n \quad A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

By defining $x = (\mu(v_1), \mu(v_2), \ldots, \mu(v_n)) \in \mathbb{R}^n$, we can then rewrite 2.17 as

$$\forall i = 1,2,\ldots,n \quad x_i = K \sum_{j=1}^{n} A_{ij} x_j \tag{2.18}$$

for some $K \in \mathbb{R}$, which in vector form becomes

$$Ax = \frac{1}{K} x\,. \tag{2.19}$$

Then if we find an eigenvector x of A with positive entries and eigenvalue λ we could define the centrality measure as

$$\forall i = 1,2,\ldots,n \quad \mu(v_i) = x_i\,, \tag{2.20}$$

and 2.17 would be satisfied with $K = \frac{1}{\lambda}$.

The *eigenvector centrality* of a graph G is then defined as the eigenvector of its adjacency matrix A corresponding to the largest eigenvalue and with positive entries; thanks to the following, known as Perron-Frobenius Theorem, we are guaranteed the largest eigenvalue is unique and it indeed has a corresponding positive eigenvector.

Theorem 2.5 *Let $A \in \mathbb{R}^{n \times n}$ be a square matrix with positive entries, i.e. $A_{ij} \geq 0$ for all $i,j = 1,2,\ldots,n$, then there exists $\lambda \in \mathbb{R}$, $\lambda > 0$ which is an eigenvalue of A such that $|\mu| < \lambda$ for any other eigenvalue μ of A and there exists an eigenvector $u = (u_1, u_2, \ldots, u_n)$ corresponding to λ with all positive entries, i.e. $u_i > 0$ for all $i = 1,2,\ldots,n$.*

3 Numerical methods for computing eigenvalues and eigenvectors

In this Chapter we will describe four methods to compute a numerical solution to the eigenvalue problem 2.7: the Oja, Oja-Sanger, Jacobi and Power method. Each of these have different characteristics that we will describe in detail in this Chapter and further test on a real-world data set in Chapter 4.

As noted in [14], a method for calculating the eigenvalues of a matrix is necessarily an iterative method. Indeed, any polynomial can be considered as the characteristic polynomial of the companion matrix associated with it. So if it were possible to have a direct method of calculating eigenvalues, this would mean that we can calculate all the roots of a polynomial of arbitrary degree in a finite number of operations which is impossible due to the famous theory of Galois.

While Oja and the Power method give an approximation of only the largest eigenvalue and its respective eigenvector, Jacobi and Oja-Sanger's method will give approximation of all of them.

We will start by describing Oja and its variant Oja-Sanger, which have the particularity that they can be applied only on a covariance matrix $X^T X$ where X is a data matrix. They will converge to eigenvalues and eigenvectors of $X^T X$ without the need to compute this, but using only the rows of X in the computation. In practice this is a big advantage because it spares computing the product $X^T X$, which for high-dimensional data points can be considerably expensive. For these two Sections our main references will be [4] and [11].

Jacobi's method instead can be applied to any symmetric matrix A. Differently than Oja-Sanger's method, for each iteration it will improve the approximation of all eigenvalues, while Oja-Sanger's method will only improve one eigenvalue at the time. For this Section our main reference will be [18].

Finally the Power method is the only of the considered method that has no special requirements on the matrix A: it can always be used. For this section our main reference will be [2].

Please not that, unless otherwise specified, in this whole Chapter we will denote the 2-norm $||\cdot||_2$ in $\mathbb{R}^n$ simply with $||\cdot||$ for simplicity of notation.

3.1 Oja and Oja-Sanger's method

Oja and Oja-Sanger's method were introduced in the context of neural network theory, and although they can work to find eigenvalues and eigenvectors for a covariance matrix coming from any type of data, we think it is useful to give a brief overview of their origin to better understand how they work. Thus we will start this Section by giving a brief overview of Hebbian learning, which is the theory of neural network they are part of. This is to be considered a brief and non-technical overview: a complete and in-depth discussion can be found for example in [4].

We will then be able to describe in detail Oja and Oja-Sanger's methods in Sections 3.1.2 and 3.1.3. Our main reference for these Sections will be [11].

3.1.1 Hebbian learning

Oja's method was first proposed in 1982 by Erkki Oja in [12] as variant of Hebbian learning in the context of neural networks. Hebbian learning is a neuroscientific theory claiming that synapses between neurons increase in strength if the neurons are both activated in a very short time span. This evolution of synapses can be modeled mathematically as the change in weights in a neural network, which in the simplest case is a linear function $W : \mathbb{R}^n \to \mathbb{R}$ defined represented by a vector $w \in \mathbb{R}^n$ and is defined by

$$\begin{aligned} \forall x \in \mathbb{R}^n \quad W(x) &= w^T x \\ &= \sum_{i=1}^{n} w_i x_i \ . \end{aligned} \tag{3.1}$$

In the theory of neural networks this would be called a simple linear neuron: the weights w_i represent the strength of the incoming n connections with other neurons and $w^T x$ represents the output of the neuron. The particularity here is that the neuron is subject to *learning*, a process that will iteratively update the weights w_i in order to obtain a specific behavior of the neuron that will depend on the application.

Modern neural networks (so-called *deep* neural networks) consist of very complex architecture composed of millions of simple neurons, and the learning is done by defining an error function at the output of the network, computing the contribution of each weight via *back-propagation* (essentially the differentiation chain-rule) and minimizing this via a technique called *gradient descent* - see for example [6].

Hebbian learning instead is not trying to minimize a specific error function, but is simply updating directly the weights based on the input data, and namely it defines the succession of vectors $\{w^n\}_{n\in\mathbb{N}} \subset \mathbb{R}^d$ by

$$\begin{cases} w^0 & = x^0 \\ w^{n+1} & = w^n + \alpha y^n x^n \quad \forall n \geq 1 \\ y^n & = (w^n)^T x^n \quad \forall n \in \mathbb{N} \,, \end{cases} \tag{3.2}$$

where $\alpha \in (0,1]$ is known as the *learning rate* and $\{x^n\}_{n\in\mathbb{N}} \subset \mathbb{R}^d$ the sequence of input data; 3.2 is known as *Hebb's rule*. At each step n the simple neuron 3.1 is applied, its output stored in $y^n \in \mathbb{R}$ and a portion α of it used to multiply the new input x^n; the resulting vector is then used to update w^n. By drawing a connection with the theory of dynamical systems, in [11] it is shown that 3.2 will grow the weights $\{w^n\}_{n\in\mathbb{N}}$ in the direction of the first principal component of the input data: this is very useful for us, because we could then take a data matrix $X \in \mathbb{R}^{d\times N}$, with $N \in \mathbb{N}$ input samples, and iterate 3.2 to obtain an approximation of the principal component, i.e. the eigenvector of $X^T X$ with largest eigenvalue (see Section 2.3).

However, there is one big problem: while the direction of $\{w_n\}_{n\in\mathbb{N}}$ will converge to that of the first principal component, the norm may explode and lead to an overflow condition, i.e. numbers bigger than the maximum number a computer can do computations with. The obvious solution would be to normalize the weights by their euclidean norm at each iteration, i.e. modifying the definition of w^{n+1} in 3.2 like this:

$$w^{n+1} = \frac{w^n + \alpha y^n x^n}{||w^n + \alpha y^n x^n||_2} \,. \tag{3.3}$$

While this would work, the solution proposed by Oja in [12] is more computationally efficient, and it is derived by considering the right side of 3.3 as a function $f(\alpha)$ and writing the Taylor expansion of f in $\alpha^\star = 0$, obtaining

$$w^{n+1} = f(\alpha^\star) + f'(\alpha^\star)\alpha + O(\alpha^2) \quad \text{for small } \alpha \,. \tag{3.4}$$

Since $f(0) = \frac{w^n}{||w^n||}$ and, as shown in [21], $f'(0) = y^n(x^n - w^n y^n)$, we can suppose w^0 to be of unitary norm and define

$$\begin{cases} w^0 & = x^0 \\ w^{n+1} & = w^n + \alpha y(x^n - y w^n) \\ y^n & = (w^n)^T x^n \quad \forall n \in \mathbb{N} \,, \end{cases} \tag{3.5}$$

which is known as *Oja's rule*. Because the right side in the definition of w^{n+1} is derived from 3.4 by simply ignoring the $O(\alpha^2)$ term, we can assume that for small α the succession $\{w_n\}_{n\in\mathbb{N}}$ will now be of bounded norm. Applying Oja's rule will thus converge to a principal component of the data, as shown in the next Section. However, if we choose a learning rate α that is too big, the approximation deriving from the Taylor expansion doesn't hold anymore and we may still incur in overflow when computing w^{n+1}, just like for Hebb's rule.

3.1.2 Oja's method

We can now properly introduce Oja's method. Suppose $\boldsymbol{x}$ is a random vector with values in $\mathbb{R}^d$ and suppose without loss of generality it has zero mean, i.e. $\mathbb{E}[x] = 0$; furthermore suppose we can draw an infinite number of samples $\{x^n\}_{n\in\mathbb{N}} \subset \mathbb{R}^d$ from $\boldsymbol{x}$ - when we introduce the Algorithm, we will introduce the necessary modifications needed to operate on only a finite number $N \in \mathbb{N}$ of samples. Finally let $\mathscr{C} = \mathbb{E}[\boldsymbol{x}\boldsymbol{x}^T]$ be the covariance matrix and C the respective sample covariance matrix - see Section 2.3.

We state once again here Oja's rule for the succession of weights $\{w^n\}_{n\in\mathbb{N}} \subset \mathbb{R}^d$:

$$\begin{cases} w^0 &= x^0 \\ w^{n+1} &= w^n + \alpha y(x^n - yw^n) \\ y^n &= (w^n)^T x^n \quad \forall n \in \mathbb{N}\,, \end{cases} \tag{3.6}$$

where $\alpha \in [0,1)$ is the learning rate. Proving convergence of $\{w^n\}_{n\in\mathbb{N}}$ to a $w^\star$ is not straightforward and requires some connections with the theory of dynamical systems: in [12] Oja claims that if the covariance matrix has eigenvalues $\lambda_1 > \lambda_2 \geq \ldots \geq \lambda_d$ (note: $\lambda_1 \neq \lambda_2$) and if the step α is not constant for all iterations but rather defined as $\alpha^n = \frac{1}{n}$, then w^n converges to the asymptotically stable solutions of the associated differential equation

$$\frac{d}{dt}w(t) = Cw(t) - (w(t)^T Cw(t))w(t)\,, \tag{3.7}$$

where $w(t)$ for $t \in \mathbb{R}$ is the continuous version of the weights w^n. However, we can easily prove that, if $\{w^n\}_{n\in\mathbb{N}}$ converges, then the limit vector must indeed be an eigenvector of C. We have in fact the following:

Theorem 3.1 *If $w^n \to w^\star$, then $w^\star$ is an eigenvector of C with eigenvalue $\lambda^\star = \mathbb{E}[(y^n)^2]$. Furthermore, $||w^\star|| = 1$.*

Proof: Since $w^n \to w^\star$ we have that $\varepsilon^n := w^{n+1} - w^n \to 0$. Observe that since by definition $y^n = (x^n)^T w^n \in \mathbb{R}$ we have

$$\begin{aligned} \varepsilon^n &= w^{n+1} - w^n \\ &= \alpha y^n (x^n - y^n w^n) \\ &= \alpha \left(x^n (x^n)^T w^n - (y^n)^2 w^n \right)\,. \end{aligned}$$

Averaging over the input data samples we have $C = \mathbb{E}[x^n (x^n)^T]$ and thus

$$Cw^n = \mathbb{E}[(y^n)^2]w^n + \frac{\varepsilon^n}{\alpha}\,,$$

which in the limit for $n \to \infty$ gives us

$$Cw^\star = \lambda^\star w^\star \, .$$

Now consider

$$\begin{aligned}\mathbb{E}[(y^n)^2] &= \mathbb{E}[\left((w^n)^T x^n\right)\left((x^n)^T w^n\right)] \\ &= (w^n)^T \mathbb{E}[x^n (x^n)^T] w^n \, .\end{aligned}$$

In the limit $n \to \infty$ this gives us

$$\begin{aligned}\lambda^\star &= (w^\star)^T C w^\star \\ &= \lambda^\star (w^\star)^T w^\star \\ &= \lambda^\star \, ||w^\star||^2 \, ,\end{aligned}$$

which implies $||w^\star|| = 1$. □

Regarding the rate of convergence of 3.6, in [1] the authors give an overview of results for rates of convergence of the Oja and Oja-Sanger methods: the proofs of these are all highly technical and out of the scope of this manuscript. For Oja's method they claim the difference between the approximated and the correct eigenvalue λ_1 will be below ε after $O\left(\frac{\lambda_1}{(\lambda_1-\lambda_2)^2} \cdot \frac{1}{\varepsilon}\right)$ iterations, where $\lambda_1 > \lambda_2 > \ldots > \lambda_d$ are the eigenvalues of C.

The pseudo code for Oja's method is given in Algorithm 1. Since in practice we never have an infinite number of data samples, we substitute x^n from equation 3.6 with x^k with $k \equiv i \pmod N$, i.e. we repeatedly iterate through the data samples. The assignment at line 8 is equivalent to (but computationally more efficient because it avoids a sum of i elements) writing $\lambda_{\text{NEW}} \leftarrow \frac{1}{i+1}\sum_{k=0}^{i-1}(y^k)^2$, where y^k are the values of y from previous iterations.

Each iteration consists only of elementary vector operations and thus the cost is simply $O(d)$, thus the total computational cost in the worst case is $O(\mathscr{I} d)$.

In practice, when we need to compute the first eigenvector of the covariance matrix $X^T X$, the Oja method has the advantage of needing only the data matrix X and thus spares the potentially costly computation of $X^T X$.

Algorithm 1 Oja's method

```
Require: data matrix $X \in \mathbb{R}^{N \times d}$, learning rate $\alpha \in \mathbb{R}$, tolerance $\varepsilon \in \mathbb{R}$, maximum iterations $\mathscr{I} \in \mathbb{N}$
Ensure: $w \in \mathbb{R}^d$ approximation of first eigenvector of $X^T X$, $\lambda$ approximation of respective eigenvalue
 1: Randomly initialize $w^0 \in \mathbb{R}^d$
 2: $\lambda_{\text{OLD}} \leftarrow 0$
 3: for $i = 0, 1, \ldots, \mathscr{I}$ do
      ▷ % Repeatedly iterate through the $N$ rows of $X$
 4:     $k \equiv i \pmod N$
 5:     $x \leftarrow X_{k.}$
 6:     $y \leftarrow x^T w^i$
 7:     $w^{i+1} \leftarrow w^i + \alpha y(x - y w^i)$
 8:     $\lambda_{\text{NEW}} \leftarrow \lambda_{\text{OLD}} \frac{\max\{1, i-1\}}{i+1} + \frac{y^2}{i+1}$
 9:     if $|\lambda_{\text{OLD}} - \lambda_{\text{NEW}}| < \varepsilon$ then
10:         Break loop
11:     end if
12:     $\lambda_{\text{OLD}} \leftarrow \lambda_{\text{NEW}}$
13: end for
14: $w \leftarrow w^i$
15: $\lambda \leftarrow \lambda_{NEW}$
```

Example 3.2 The data matrix

$$X = \begin{pmatrix} -3.4 & -1 & -2.2 \\ 3.6 & -1 & 0.8 \\ 3.6 & 1 & -3.2 \\ -3.4 & 1 & 1.8 \\ -0.4 & 0 & 2.8 \end{pmatrix}$$

consists of 5 centered (i.e. with mean 0) data points in $\mathbb{R}^3$ and has sample covariance matrix

$$\begin{aligned} C &= \frac{1}{4} X^T X \\ &= \begin{pmatrix} 12.3 & 0 & -2.1 \\ 0 & 1 & 0 \\ -2.1 & 0 & 6.7 \end{pmatrix} , \end{aligned}$$

with largest eigenvalue 13 and associated eigenvector $u = [0.94868, 0, -0.31622]$. Suppose now we set the step to $\alpha = 0.01$ and we initialize w^0 by sampling from a normal distribution with mean 0 and standard deviation 0.25 obtaining $w^0 = [0.18, -0.1, 0.16]$. The first iteration of Oja's rule would then give

$$\begin{aligned} y^1 &= 0.18 \cdot (-3.4) + -0.1 \cdot (-1) + 0.16 \cdot (-2.2) \\ &= -0.864 , \\ w^1 &= \begin{pmatrix} 0.18 \\ -0.1 \\ 0.16 \end{pmatrix} - 0.00864 \left(\begin{pmatrix} -3.4 \\ -1 \\ -2.2 \end{pmatrix} + 0.864 \begin{pmatrix} 0.18 \\ -0.1 \\ 0.16 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.20803 \\ -0.09061 \\ 0.17781 \end{pmatrix} . \end{aligned}$$

The estimation of the eigenvalue after one iteration would then be $0.864^2 = 0.7465$. The second iteration would give

$$\begin{aligned} y^2 &= 0.20803 \cdot 3.6 + -0.09061 \cdot (-1) + 0.17781 \cdot 0.8 \\ &= 0.98178\,, \\ w^2 &= \begin{pmatrix} 0.20803 \\ -0.09061 \\ 0.17781 \end{pmatrix} + 0.0098178 \left(\begin{pmatrix} 3.6 \\ -1 \\ -0.8 \end{pmatrix} - 0.98178 \begin{pmatrix} 0.20803 \\ -0.09061 \\ 0.17781 \end{pmatrix} \right) \\ &= \begin{pmatrix} 0.17264 \\ -0.09955 \\ 0.15450 \end{pmatrix}. \end{aligned}$$

The estimation of the eigenvalue after two iterations would then be $\frac{0.864^2+0.98178^2}{2} = 0.8552$. It would look that, while the eigenvector approximation w^2 is a much better approximation of u than w^0, the eigenvalue is still very far from the correct value 13. However the relative errors for the eigenvector approximation is worse than for the eigenvalue:

$$\frac{||w^2 - u||_2}{||u||_2} = 1.7038$$
$$\frac{|0.8552 - 13|}{|13|} = 0.9342\,.$$

3.1.3 Oja-Sanger's Method

Oja-Sanger's method is a variant of Oja's method that enables one to compute approximation of all eigenvalues and eigenvectors of the covariance matrix. It consists of recursively applying Oja's method on a transformed version of the data matrix X; the transformations are computed at the beginning of each iteration and have the effect of projecting the data onto a space orthogonal to that generated by the previously computed eigenvectors. With this frame, Oja's rule will successively approximate all the eigenvectors, by decreasing order of their respective eigenvalues. For this Section our main references will be [4] and [11].

We will be using in this Section the same hypothesis and notations as in the precedent one. Oja-Sanger's method is based on a modification of equation 3.6 to obtain approximations $w_1, w_2, \dots, w_d$ of all d eigenvectors of C:

$$\begin{cases} w_i^0 &= x_i^0 \\ w_i^{n+1} &= w_i^n + \alpha y_i (x_i^n - y_i^n w_i^n) \,, \\ y_i^n &= (x_i^n)^T w_i^n \quad \forall n \in \mathbb{N} \,, \end{cases} \tag{3.8}$$

where for all $n \in \mathbb{N}$

$$\begin{cases} x_1^n &= x^n \\ x_i^n &= x_{i-1}^n - y_{i-1}^n w_{i-1}^n \,, \end{cases} \tag{3.9}$$

where $\alpha \in [0,1)$ is the learning rate and $i = 1,2,\ldots,d$ is the index corresponding to the eigenvector; this is known as *Oja-Sanger's* rule. Because x_i^n depends on quantities referring with index $i-1$, the application of these formulas must consist of an outer and inner loop: the outer loop will iterate on i, going from 1 to d. The inner loop corresponds to iterating 3.8 for $n = 0,1,\ldots$, which is simply Oja's rule applied to $\{x_{i-1}\}_{n\in\mathbb{N}}$, to obtain an approximation of w_i. For a certain i in the outer loop, the inner loop is using only quantities from previous iterations of the outer loop; in other words, to compute approximations w_k only values w_i, x_i, y_i with $i \leq n$ are used.

The definition of x_i^n in 3.9 is similar to the so-called Gram-Schmidt orthogonalization process (see for example [19], [16] or [8]): suppose the first $I \leq d$ iterations of the outer loop already gave a good approximation of the eigenvectors $w_1^\star, w_2^\star, \ldots, w_I^\star$ of C, then the transformed data x_{I+1}^n will be (in the limit for $n \to \infty$) orthogonal to all these already computed eigenvectors. In fact we have:

Theorem 3.3 *Suppose the random samples are limited in norm, i.e. $\exists K \in \mathbb{R}$ s.t. for all $i,n \in \mathbb{N}$ we have $||x_i^n|| \leq K$, and suppose that $w_i^n \to w_i^\star$ for all $i = 1,2,\ldots,d$. Then $w_1^\star, w_2^\star, \ldots, w_d^\star$ are eigenvectors of C, are orthonormal and for all $i = 1,2,\ldots,d$ we have*

$$\forall \varepsilon > 0 \quad \exists N \text{ s.t. } \forall k \leq i \quad \forall n \geq N \quad |(w_k^\star)^T x_{i+1}^n| < \varepsilon \,. \tag{3.10}$$

Proof: Observe that from 3.9 and from the definition of $y_i^n = (x_i^n)^T w_i^n = (w_i^n)^T x_i^n$ in 3.8 we have:

$$\begin{aligned} (w_k^\star)^T x_{i+1}^n &= (w_k^\star)^T x_i^n - y_i^n (w_k^\star)^T w_i^n \\ &= (w_k^\star)^T x_i^n - (w_i^n)^T x_i^n (w_k^\star)^T w_i^n \,. \end{aligned} \tag{3.11}$$

To prove the Theorem we will use induction on i. For the base case $i = 1 = k$ Oja-Sanger's rule reduces to Oja's rule and from Theorem 3.1 we know $w_1^\star$ is an eigenvector of C and $||w_1^\star|| = 1$. Since $w_1^n \to w_1^\star$, we can choose $N \in \mathbb{N}$ s.t. for all $n \geq N$ we can write $w_1^n = w_1^\star + \bar{\varepsilon}$ with $||\bar{\varepsilon}|| \leq \varepsilon_1^\star$, where $\varepsilon_1^\star \in \mathbb{R}$ will be specified later. Then, for $n \geq N$, 3.11

becomes

$$(w_1^\star)^T x_2^n = (w_1^\star)^T x_1^n - (w_1^\star + \bar{\varepsilon})^T x_1^n (w_1^\star)^T (w_1^\star + \bar{\varepsilon})$$
$$= (w_1^\star)^T x_1^n - (w_1^\star)^T x_1^n (w_1^\star)^T w_1^\star - (w_1^\star)^T x_1^n (w_1^\star)^T \bar{\varepsilon} - \bar{\varepsilon}^T x_1^n (w_1^\star)^T w_1^\star - \bar{\varepsilon}^T x_1^n (w_1^\star)^T \bar{\varepsilon} \, .$$

Since $(w_1^\star)^T w_1^\star = ||w_1^\star|| = 1$, the first two terms cancel. Taking the absolute value and using Cauchy-Schwarz's inequality we then have

$$\left|(w_1^\star)^T x_2^n\right| \le ||(w_1^\star)||\,||x_1^n||\,||(w_1^\star)||\,||\bar{\varepsilon}|| + ||\bar{\varepsilon}||\,||x_1^n||\,||(w_1^\star)||\,||(w_1^\star)|| + ||\bar{\varepsilon}||\,||x_1^n||\,||(w_1^\star)||\,||\bar{\varepsilon}||$$
$$= 2\,||x_1^n||\,||\bar{\varepsilon}|| + ||x_1^n||\,||\bar{\varepsilon}||^2$$
$$\le 2K\varepsilon_1^\star + K(\varepsilon_1^\star)^2 \, .$$

Because the discriminant of the quadratic equation (in $\bar{\varepsilon}_1^\star$)

$$K(\bar{\varepsilon}_1^\star)^2 + 2K\bar{\varepsilon}_1^\star - \varepsilon = 0$$

is $4K^2 + 4K\varepsilon > 0$, we can surely choose $\bar{\varepsilon}_1^\star$ s.t. $\left|(w_1^\star)^T x_2^n\right| \le \varepsilon$ for all $n \ge N$.

For the induction step, supposing the Theorem is true for all $i \le \bar{i} - 1$ we will prove it for $\bar{i}$. Since we have $w_{\bar{i}}^n \to w_{\bar{i}}^\star$, we can choose $N \in \mathbb{N}$ s.t. for all $n \ge N$ we can write $w_{\bar{i}}^n = w_{\bar{i}}^\star + \bar{\varepsilon}$ with $||\bar{\varepsilon}|| \le \varepsilon_{\bar{i}}^\star$, where $\varepsilon_{\bar{i}}^\star \in \mathbb{R}$ will be specified later.

We distinguish two cases, and namely if $k = \bar{i}$ and $k < \bar{i}$. For $k = \bar{i}$ and $n \ge N$, 3.11 becomes

$$(w_{\bar{i}}^\star)^T x_{\bar{i}+1} = (w_{\bar{i}}^\star)^T x_{\bar{i}}^n - (w_{\bar{i}}^\star + \bar{\varepsilon})^T x_i^n (w_{\bar{i}}^\star)^T (w_{\bar{i}}^\star + \bar{\varepsilon}) \, ,$$

and, with computations that are completely analogous for the base case $i = 1 = k$, one can show that $\bar{\varepsilon}_{\bar{i}}^\star$ can be chosen small enough to have $|(w_{\bar{i}}^\star)^T x_{\bar{i}+1}| < \varepsilon$.

Suppose now that $j < \bar{i}$ and $n \ge N$, equation 3.11 then becomes

$$(w_k^\star)^T x_{\bar{i}+1} = (w_k^\star)^T x_{\bar{i}}^n - (w_{\bar{i}}^\star + \bar{\varepsilon})^T x_i^n (w_k^\star)^T (w_{\bar{i}}^\star + \bar{\varepsilon})$$
$$= (w_k^\star)^T x_{\bar{i}}^n - (w_{\bar{i}}^\star)^T x_i^n (w_k^\star)^T w_{\bar{i}}^\star - (w_{\bar{i}}^\star)^T x_i^n (w_k^\star)^T \bar{\varepsilon} - (\bar{\varepsilon})^T x_i^n (w_k^\star)^T w_{\bar{i}}^\star - (\bar{\varepsilon})^T x_i^n (w_k^\star)^T \bar{\varepsilon} \, .$$

The second and fourth term in the last expression are null, because $(w_k^\star)^T w_{\bar{i}}^\star = 0$. Furthermore because of the induction hypothesis, we can suppose $|(w_k^\star)^T x_{\bar{i}}^n| < \varepsilon_1$ for large enough n and for some $\varepsilon_1 \in \mathbb{R}$ yet to determine. Using again the Cauchy-Scwharz inequality we then have

$$|(w_k^\star)^T x_{\bar{i}+1}| \le \varepsilon_1 + \left|\left|x_{\bar{i}}^n\right|\right|\,||\bar{\varepsilon}|| + ||x_{\bar{i}^n}||\,||\bar{\varepsilon}||^2$$
$$\le \varepsilon_1 + K\varepsilon_{\bar{i}}^\star + K(\varepsilon_{\bar{i}}^\star)^2 \, .$$

Consider the quadratic equation

$$K(\varepsilon_{\bar{i}}^{\star})^2 + K\varepsilon_{\bar{i}}^{\star} + \varepsilon_1 - \varepsilon = 0\,;$$

its discriminant is $K^2 - 4K(\varepsilon_1 - \varepsilon)$ which, if we choose $\varepsilon_1 < \frac{K^2}{4K} + \varepsilon$, is positive. Thus fur such a choice of ε_1 a choice of $\varepsilon_{\bar{i}}^{\star}$ can be made so that

$$|(w_k^{\star})^T x_{\bar{i}+1}^n| < \varepsilon$$

holds for all $n \geq N$.

$\square$

Remark 3.4 The hypothesis in the previous Theorem on the boundedness of the data samples is always verified when, as it usually happens in practice, we have a limited number of data samples and a limited number of iterations.

Regarding the convergence rate of Oja-Sanger's method, in [1] the authors claim the error will be below ε after $O\left(\frac{\sum_{i=1}^{d}\lambda_i}{(\lambda_1-\lambda_2)^2}\cdot\frac{1}{\varepsilon}\right)$ iterations, where $\lambda_1 > \lambda_2 > \ldots > \lambda_d$ are the eigenvalues of C.

Algorithm 2 Oja-Sanger's method

```
Require: data matrix $X \in \mathbb{R}^{N\times d}$, learning rate $\alpha \in \mathbb{R}$, tolerance $\varepsilon \in \mathbb{R}$, Oja iterations $\mathscr{I} \in \mathbb{N}$
Ensure: $w_1, w_2, \ldots w_d \in \mathbb{R}^d$ approximation of eigenvectors of $X^TX$, $\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{R}$ approximations of respective eigenvalues
1: $X_1 \leftarrow X$
2: for $i = 1, 2\ldots, d$ do
3:     $w_i, \lambda_i \leftarrow$ Oja$(X_i, \alpha, \varepsilon, \mathscr{I})$          ▷ apply Oja's rule on $X_i$
4:     for $k = 1, 2, \ldots, N$ do
5:         $y_i^k \leftarrow (x_i^k)^T w_i$
6:         $x_{i+1}^k \leftarrow x_i^k - y_i^k w_i$          ▷ $x_i^k$ is the $k$-th row of $X_i$
7:     end for
8:     $X_{i+1} \leftarrow$ stack $x_{i+1}^1, x_{i+1}^2, \ldots, x_{i+1}^N$ as row vectors
9: end for
```

The pseudo code for Oja-Sanger's method is given in Algorithm 2. The computational cost of one iteration is dominated by the application of Oja's rule; the total cost in the worst case will then be $O(\mathscr{I}d^2)$.

One of the big advantages of of Oja-Sanger's method is that we can easily modify line 2 to iterate simply over i from 1 to only a certain $K < d$, in the case we only need approximations of the first K eigenvectors and eigenvalues. This is often the case when we interested in the principal component analysis of some data set, as we will be in Section 4.3.

Furthermore, just like for Oja's method, Oja-Sanger's method does not require the potentially very costly computation of the covariance matrix $X^T X$ in order to approximate its eigenvectors.

Example 3.5 We take the same data matrix as in Example 3.2, whose sample covariance matrix has eigenvalues $13, 6$ and 1 with eigenvectors

$$u_1 = \begin{pmatrix} 0.94868 \\ 0 \\ -0.31622 \end{pmatrix}, \; u_2 = \begin{pmatrix} 0.31622 \\ 0 \\ 0.94868 \end{pmatrix}, \; u_3 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} .$$

Suppose we set a step of $\alpha = 0.01$ and a tolerance of $\varepsilon = 1^{-6}$; the first iteration of the Oja-Sanger method will then be equivalent to Oja's method on the data with these parameters. After 41 iterations it achieves a residue below the tolerance and yields:

$$w_1^\star = \begin{pmatrix} 0.94868 \\ 1^{-12} \\ -0.31622 \end{pmatrix}$$

$$\lambda_1^\star = 10.4 .$$

Then by iterating on the rows of $X_1 = X$ the values x_2^k for $k = 1, 2, \ldots, 5$ are computed and give the matrix

$$X_2 = \begin{pmatrix} -1 & -1 & -3 \\ 0.6 & -1 & 1.8 \\ -0.6 & 1 & -1.8 \\ 0.2 & 1 & 0.6 \\ 0.8 & 0 & 2.4 \end{pmatrix} .$$

In the next iteration Oja's method is applied to X_2; after 55 iterations it achieves a residue below the threshold and yields

$$w_2^\star = \begin{pmatrix} 0.31622 \\ 6 \cdot 1^{-8} \\ 0.94869 \end{pmatrix}$$

$$\lambda_2^\star = 4.8 .$$

After 96 iterations we then have approximations of the first two eigenvalues and eigenvectors with relative errors:

$$\frac{||w_1^\star - u_1||_2}{||u_1||_2} = 1^{-12}$$

$$\frac{|10.4 - 13|}{|13|} = 0.2$$

$$\frac{||w_2^\star - u_2||_2}{||u_2||_2} = 1^{-5}$$

$$\frac{|4.8 - 6|}{|6|} = 0.2 \, .$$

3.2 Jacobi's method

Jacobi's method allows one to compute approximations of all eigenvalues and eigenvectors of a symmetrical matrix $A \in \mathbb{R}^{n \times n}$, and it does so by a series of geometrical transformations known as Givens rotations that are applied to the matrix. The Givens rotations allow to find a sequence of invertible matrices $\{Q^k\}_{k \in \mathbb{N}}$ such that $A^k = (Q^k)^{-1} A Q^k$ and

$$\lim_{k \to \infty} A^k = \lim_{k \to \infty} (Q^k)^{-1} A Q^k = D$$

where D is a diagonal matrix with the eigenvalues of A on the diagonal. Since the matrices are all similar to the original matrix A, they have the same eigenvalues (see Proposition 1).

Differently than Oja-Sanger's method, at each iteration Jacobi's method improves the approximation of all the eigenvalues and eigenvectors simultaneously. It is however not possible to restrict the method to find approximation of only a limited number of eigenvalues and eigenvectors, like it's possible with Oja-Sanger's method.

We will start in Section 3.2.1 by introducing the matrices associated to Givens rotations and some of their properties. This will allow us in Section 3.2.2 to describe Jacobi's method, prove its convergence and convergence rate as well as computational cost. For both Sections our main reference will be [18].

3.2.1 Givens rotation

The *Givens matrix* Q_θ^{pq} is defined by :

$$Q_\theta^{pq} = \left(\begin{array}{ccccccc} & & (p) & & (q) & & & \\ 1 & 0 & \dots & \dots & \dots & \dots & 0 & \\ \vdots & \ddots & \vdots & & \vdots & & \vdots & \\ \vdots & & \cos(\theta) & \dots & -\sin(\theta) & & \vdots & (p) \\ \vdots & & \vdots & \ddots & \vdots & & \vdots & \\ \vdots & & \sin(\theta) & \dots & \cos(\theta) & & \vdots & (q) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & \dots & \dots & \dots & \dots & \dots & 1 & \end{array}\right), \tag{3.12}$$

i.e. it is a modification of the identity matrix with $\cos(\theta)$ in position p,p, $-\sin(\theta)$ in position p,q, $\sin(\theta)$ in position q,p and $\cos(\theta)$ in position p,p. A basic property at the heart of Jacobi's method iterations is that when we apply the so-called *Givens rotation*

$$(Q_\theta^{pq})^{-1}AQ_\theta^{pq}$$

to A we are modifying only the rows and columns p and q. In fact we have:

Proposition 4 *Let $A \in \mathbb{R}^{n\times n}$ be a symmetrical matrix with $A_{pq} \neq 0$ and $B = Q_\theta^{pq-1}AQ_\theta^{pq}$; then B differs from A only in rows and columns p and q, is symmetric and if*

$$\theta = \frac{1}{2}\arctan\frac{2A_{pq}}{A_{qq}-A_{pp}}$$

then $B_{pq} = B_{qp} = 0$

Proof: By applying a permutation to the rows and columns of the matrices, we can always assume without loss of generality that p and q are the last indices. We can thus write

$$\begin{aligned} B =& Q_\theta^{pq-1}AQ_\theta^{pq} \\ =& \left(\begin{array}{c|c} I & 0 \\ \hline 0 & R_\theta^{-1} \end{array}\right) \left(\begin{array}{c|cc} \tilde{A}_{pq} & \multicolumn{2}{c}{\tilde{a}_{pq}} \\ \hline \tilde{a}_{pq}^T & a_{pp} & a_{pq} \\ & a_{qp} & a_{qq} \end{array}\right) \left(\begin{array}{c|c} I & 0 \\ \hline 0 & R_\theta \end{array}\right) \qquad (3.13) \\ =& \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right), \qquad (3.14) \end{aligned}$$

where

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} ,$$

$\tilde{A}_{pq}$ is the sub matrix extracted from A by eliminating columns p and q while $\tilde{a}_{pq}$ is the sub matrix obtained by extracting only those columns.

We will now consider each block of B separately. For B_{11} we have

$$\begin{aligned} B_{11} &= \left(I\tilde{A}_{pq} + 0 \begin{pmatrix} a_{p\cdot} \\ \hline a_{q\cdot} \end{pmatrix} \right) I + (\ldots)0 \\ &= \tilde{A}_{pq} , \end{aligned}$$

i.e.

$$b_{ij} = a_{ij} \quad \forall i,j \neq p,q .$$

For B_{12} and B_{21}, considering that R_θ is orthogonal and $\tilde{a}_{pq}$ symmetric, we have

$$\begin{aligned} B_{12} &= \tilde{a}_{pq} R_\theta \\ &= R_\theta^{-1} \tilde{a}_{pq}^T \\ &= B_{21} . \end{aligned}$$

Component-wise we obtain

$$\begin{aligned} b_{pj} = b_{jp} = a_{pj}\cos\theta - a_{qj}\sin\theta \quad \forall j \neq p \\ b_{qj} = b_{jq} = a_{qj}\sin\theta + a_{qj}\cos\theta \quad \forall j \neq q . \end{aligned}$$

Finally, consider

$$\begin{aligned} B_{22} &= R_\theta^{-1} \begin{pmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq} \end{pmatrix} R_\theta \\ &= \begin{pmatrix} b_{pp} & b_{pq} \\ b_{qp} & b_{qq} \end{pmatrix} , \end{aligned} \tag{3.15}$$

which is again symmetric (and thus the whole B is symmetric). Identification of the coordinates gives

$$b_{pp} = a_{pp}\cos^2\theta + a_{qq}\sin^2\theta - a_{pq}\sin 2\theta$$

$$b_{qq} = a_{pp}\sin^2\theta + a_{qq}\cos^2\theta + a_{pq}\sin 2\theta$$

$$b_{pq} = a_{pq}\cos 2\theta + \frac{a_{pp}-a_{qq}}{2}\sin 2\theta\ .$$

By choosing $\theta \in (-\frac{\pi}{4}, \frac{\pi}{4})$ such that $\tan 2\theta = \frac{2a_{pq}}{a_{qq}-a_{pp}}$ (or $\theta = \frac{\pi}{4}$ if $a_{qq} = a_{pp}$) we obtain $b_{pq} = b_{qp} = 0$ and the Proposition is proved. □

To prove the next Proposition we will need the following Lemma, which establishes the invariance of the Frobenius norm under an orthogonal rotation.

Lemma 3.6 *Let Q be orthogonal; then $||QA||_F = ||A||_F$*

Proof: By the cyclic property of the trace [1], we have

$$\begin{aligned} ||QA||_F^2 &= tr(QA(QA)^T) \\ &= tr(QAA^TQ^T) \\ &= tr(Q^TQAA^T) \\ &= tr(AA^T) \\ &= ||A||_F^2\ . \end{aligned}$$

□

Let's define the squared sum of off-diagonal elements of matrix A as

$$\Gamma(A) = \sum_{i\neq j} |A_{ij}|^2\ . \tag{3.16}$$

To prove convergence of the Jacobi method in the next Section, we will need the following Proposition which clarifies how a Givens rotation affects the value of $\Gamma(A)$. We have

Proposition 5 *Let $A \in \mathbb{R}^{n\times n}$ be a symmetrical matrix with $A_{pq} \neq 0$ and define $B = Q_\theta^{pq^{-1}} A Q_\theta^{pq}$ with $\theta = \frac{1}{2}\arctan\frac{2A_{pq}}{A_{qq}-A_{pp}}$. Then B has the same eigenvalues of A and*

$$\Gamma(B) = \Gamma(A) - 2|A_{pq}|^2\ . \tag{3.17}$$

[1] $tr(ABCD) = tr(BCDA) = tr(CDAB) = tr(DABC)$

Proof: Since B is symmetric to A, from Proposition 1 it follows they must have the same eigenvalues. By applying Lemma 3.6 to 3.15 we are able to obtain

$$\begin{aligned}|b_{pp}|^2+|b_{qq}|^2 &= ||B_{22}||_F^2 \\ &= \left|\left|\begin{pmatrix} a_{pp} & a_{pq} \\ a_{qp} & a_{qq}\end{pmatrix}\right|\right|_F^2 \\ &= |a_{pp}|^2+|a_{qq}|^2+2|a_{pq}|^2 \,,\end{aligned}$$

where we are again using the block matrix notation from the proof of Proposition 4. We then have

$$\begin{aligned}\Gamma(B) &= ||B||_F^2 - \sum i|b_{ii}|^2 \\ &= ||A||_F^2 - \sum i \neq p,q|a_{ii}|^2 - (|b_{pp}|^2+|b_{qq}|^2) \\ &= \Gamma(A)+|a_{pp}|^2+|a_{qq}|^2-(|b_{pp}|^2+|b_{qq}|^2) \\ &= \Gamma(A)-2|a_{pq}|^2 \,.\end{aligned}$$

□

3.2.2 The algorithm

We saw in Proposition 5 that applying a Givens rotation with to a matrix does not The idea of Jacobi's method is to iteratively apply Givens rotations eliminating off-diagonal elements while preserving eigenvalues. The Jacobi method is an iterative method defined on a symmetric matrix $A \in \mathbb{R}^{n\times n}$ as follows:

$$\begin{cases} A^0 & = A \\ A^{(k+1)} & = Q^{(k)^{-1}}A^{(k)}Q^{(k)} \end{cases}$$
$$\text{where } Q^{(k)} = Q_\theta^{pq}$$
$$\text{with } p,q \in \arg\max_{i,j}|a_{ij}^{(k)}| \tag{3.18}$$
$$\text{and } \theta = \frac{1}{2}\arctan\frac{2a_{pq}^{(k)}}{a_{qq}^{(k)}-a_{pp}^{(k)}} \,.$$

Note that the choice of θ at iteration k is exactly what we need in order for $Q^{(k)^{-1}}A^{(k)}Q^{(k)}$ to have a 0 in its p,q-th entry - see Proposition 4. This means that at each iteration k we choose the largest off-diagonal element in A^k and apply a Givens rotation to eliminate it. In the following Theorem we use the results from the previous Section to prove the convergence and convergence rate of the Jacobi method.

Theorem 3.7 *The sequence $\Gamma(A^{(k)})$ converges to 0 with rate of convergence $\sqrt{1-\frac{1}{N}}$ where $N=\frac{n(n-1)}{2}$ and thus the diagonal elements of $A^{(k)}$ converge to the eigenvalues of A.*

Proof: Define $\mu = |a_{pq}| = \max_{i,j} |a_{ij}^{(k)}|$. Since all matrices in the Jacobi sequence have $n \times n$ elements, the number of off-diagonal elements is $2N = n(n-1)$; we thus have

$$\mu^2 \leq \Gamma(A^{(k)})^2 \leq 2N\mu^2$$

$$\Rightarrow -2\mu^2 \leq -\frac{\Gamma(A^{(k)})^2}{N} .$$

From Proposition 5 it then follows

$$\begin{aligned}\Gamma(A^{(k+1)})^2 &= \Gamma(A^{(k)}) - 2\mu^2 \\ &\leq \Gamma(A^{(k)})^2 - \frac{\Gamma(A^{(k)})^2}{N} \\ &= \Gamma(A^{(k)})^2(1-\frac{1}{N}) ,\end{aligned}$$

or

$$\frac{\Gamma(A^{(k+1)})}{\Gamma(A^{(k)})} \leq \sqrt{1-\frac{1}{N}} .$$

□

Remark 3.8 Since $A^{(k)} = Q^{-1}AQ$ with $Q = Q^{(0)}Q^{(1)} \dots Q^{(k-1)}$ orthogonal, the columns of Q approach the eigenvectors of A

In Section 6.5.2 of [18] it is reported that, one can achieve even quadratic convergence, by a simple modification of Jacobi's method 3.18 that consists in another choice of the p,q indices, namely row-wise $((1,2),(1,3),\dots,(1,n),(2,3),(2,4),\dots,(2,n),\dots(n-1,n))$.

In Algorithm 3 the pseudo code for Jacobi's method is given. Building the matrix X on line 8 can be done in $O(n)$ time thanks to the component-wise formulas found in the proof of Proposition 4. The cost per iteration is then dominated by line 6 which is $O(n^2)$, making the cost of the whole procedure $O(\mathscr{I}n^2)$. As alternative stop criteria one could set a threshold ε and check if the difference between diagonal elements of X between two successive iterations falls below ε.

Algorithm 3 Jacobi's method

```
Require: data matrix $A \in \mathbb{R}^{n\times n}$ symmetric, tolerance $\varepsilon \in \mathbb{R}$, maximum iterations $\mathscr{I} \in \mathbb{N}$
Ensure: $u_1, u_2, \dots, u_n$ approximation of eigenvectors of $A$ and $\lambda_1, \dots, \lambda_n \in \mathbb{R}$ approximation of the respective eigenvalues.
 1: $X \leftarrow A$
 2: for $i = 1, 2, \dots, n$ do
 3:     $\lambda_i^{\text{OLD}} \leftarrow x_{ii}$
 4: end for
 5: for $i = 1, 2, \dots, \mathscr{I}$ do
 6:     $p, q \leftarrow \arg\max_{i,j} |x_{ij}|$
 7:     $\theta \leftarrow \frac{1}{2} \arctan \frac{2x_{pq}^{(k)}}{x_{qq}^{(k)} - x_{pp}^{(k)}}$
 8:     Build $X = {Q^{(k)}}^{-1} X Q^{(k)}$          ▷ Use component-wise formulas from Proposition 4
 9:     all_below ← True
10:     for $j = 1, 2, \dots, n$ do
11:         $\lambda_j^{\text{NEW}} \leftarrow x_{jj}$
12:         if $|\lambda_j^{\text{NEW}} - \lambda_j^{\text{OLD}}| \geq \varepsilon$ then
13:             all_below ← False
14:         end if
15:         $\lambda_j^{\text{OLD}} \leftarrow \lambda_j^{\text{NEW}}$
16:     end for
17:     if all_below == True then
18:         Break loop
19:     end if
20: end for
21: $Q \leftarrow Q^0 Q^1 \dots Q^i$
22: for $i = 1, 2, \dots, n$ do
23:     $\lambda_i \leftarrow \lambda_i^{\text{NEW}}$
24:     $u_i = Q_{\cdot i}$
25: end for
```

Example 3.9 Let's find the eigenvalues and eigenvectors of the following matrix, which is different from the previous examples in order to simplify computations:

$$A = \begin{pmatrix} 1 & \sqrt{2} & 2 \\ \sqrt{2} & 3 & \sqrt{2} \\ 2 & \sqrt{2} & 1 \end{pmatrix}.$$

The highest value off diagonal is 2 correspond to $a_{31} == 2$: thus $p = 1$ and $q = 3$ and $a_{pp} = 1$,$a_{qq} = 1$ and $a_{pq} = a_{qp} = 2$. In this case $\theta = \frac{\pi}{4}$ and the rotation matrix is given by

$$Q^0 = \begin{pmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{pmatrix}$$

Then

$$A^1 = Q^{1^T} A Q^1$$

$$= \begin{pmatrix} 3 & 2 & 0 \\ 2 & 3 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

We repeat the same instructions on A_1, obtaining $a_{21} = a_{11} = 2$: thus $p = 1$ and $q = 2$ and $a_{pp} = 3$, $a_{qq} = 3$ and $a_{pq} = a_{qp} = 2$. Then again $\theta = \frac{\pi}{4}$ and

$$Q^2 = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Finally we obtain :

$$A^2 = Q^{2^T} A_1 Q^2$$

$$= \begin{pmatrix} 5 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

which is an exact diagonalization of A_1 (and thus A), i.e. $\Gamma(A^2) = 0$. The eigenvalues and eigenvectors given by the Jacobi method are then in this case exact: the eigenvalues are $(5, 1, -1)$ and the eigenvectors the columns of

$$\begin{aligned} Q &= Q^1 Q^2 \\ &= \begin{pmatrix} -\frac{1}{2} & -\frac{1}{2} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{\sqrt{2}} \end{pmatrix} . \end{aligned}$$

3.3 Power method

The Power method (also known as *von Mises* method) is the last iterative method for the eigenvalue problem we will treat in this thesis. Out of the methods considered in this thesis, the Power method is the only one that can be applied to non-symmetric matrices. It converges exclusively to the largest eigenvalue (in absolute value) of the matrix, however depending on the application this might be all that is needed: for example in section 4.5 we will use the Power method to compute the eigenvector centrality of a graph, which is given by the eigenvector associated to the largest eigenvalue of the adjacency matrix. By a simple transformation on the matrix (which involves a matrix inversion) the Power method can be made to converge to the smallest eigenvalue as well - we will see this variant at the end of the section. For this whole section our main reference will be [2].

Suppose $A \in \mathbb{R}^{n\times n}$ with eigenvalues

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|$$

and associated eigenvectors $u_1, u_2, \ldots, u_n$ which we will suppose of unitary norm. The power method is defined as the succession of values

$$\begin{cases} x^{k+1} & = \frac{Ax^k}{||Ax^k||} \\ x^0 & \in \mathbb{R} \, , \, x \neq 0 \end{cases} . \tag{3.19}$$

In the following we prove convergence and convergence rate of the Power method.

Theorem 3.10 *If $(x^0)^T u_1 \neq 0$, the power method converges for $k \to \infty$ to the first eigenvector $x^k \to u_1$ (with possibly opposite sign) with convergence rate $O(\left|\frac{\lambda_2}{\lambda_1}\right|)$*

Proof: Suppose we can write x^0 in the basis $u_1, u_2, \ldots u_n$ as $x^0 = \alpha_1 u_1 + \alpha_2 u_2 + \ldots +$

$\alpha_n u_n$ where $\alpha_1 = (x^0)^T u_1 \neq 0$. We have:

$$\begin{aligned} x^{k+1} &= \frac{Ax^k}{||Ax^k||} \\ &= \frac{A^k x}{||A^k x||} \\ &= \frac{\sum_{i=1}^n \alpha_i \lambda_i^k u_i}{||\sum_{i=1}^n \alpha_i \lambda_i^k u_i||} \\ &= \frac{\lambda_1^k}{|\lambda_1^k|} \frac{\alpha_1 u_1 + \sum_{i=2}^n \left(\frac{\lambda_i}{\lambda_1}\right)^k \alpha_i u_i}{\left|\left|\alpha_1 u_1 + \sum_{i=2}^n \left(\frac{\lambda_i}{\lambda_1}\right)^k \alpha_i u_i\right|\right|} . \end{aligned} \tag{3.20}$$

Since $\frac{\lambda_i}{\lambda_1} < 1$for all $i \geq 2$, it follows that $x^k \to sgn(\lambda_1) sgn(\alpha_1) u_1$. Since the dominant term in the sum is $\left(\frac{\lambda_2}{\lambda_1}\right)^k$, the convergence is $O\left(|\frac{\lambda_2}{\lambda_1}|^k\right)$. □

Remark 3.11 The requirement that the first two eigenvalues are not the same, i.e. $\lambda_2 < \lambda_1$, is essential, otherwise the series 3.20 will not in general converge.

Remark 3.12 We can use the *Rayleigh quotient* $\mu^k = \frac{x^{k^T} A x^k}{x^{k^T} x}$ as approximation of λ_1, since $\frac{u_1^T A u_1}{u_1^T u_1} = \lambda_1$.

In Algorithm 4 the pseudo code for the power method is given. The cost of one iteration is dominated by the matrix-vector multiplication and is thus $O(n^2)$; the total cost is then $O(\mathscr{I} n^2)$ where $\mathscr{I}$ is the number of iterations. Alternatively one could set a threshold ε and iterate the power method until the difference between the approximated eigenvalues at successive steps falls beneath ε.

Inverse Power Method

Using Proposition 1, it is straightforward to check that for any $\mu \in \mathbb{R} \setminus \{0\}$ the matrix $(A - \mu I)^{-1}$ has same eigenvectors as A and eigenvalues $(\lambda_i - \mu)$: applying the power method to this matrix is known as applying the *inverse power method* to matrix A. It will converge to the eigenvector with eigenvalue closest to μ, as long as this is not the same in absolute value to the second closest one. In particular with $\mu = 0$, if $\lambda_{n-1} > \lambda_n$, it will converge to λ_n. The computational cost however will become $O(n^3)$ due to the cost of inverting the matrix.

Algorithm 4 Power iteration method

```
Require: data matrix $A \in \mathbb{R}^{n\times n}$, tolerance $\varepsilon \in \mathbb{R}$, maximum iterations $\mathscr{I} \in \mathbb{N}$
Ensure: $x^\star \in \mathbb{R}^n$ approximation of first eigenvector of $A$ and approximation of respective
    eigenvalue $\lambda$
 1: Initialize $x \in \mathbb{R}^n$ randomly
 2: $\lambda_{\text{OLD}} \leftarrow \frac{x^T A x}{x^T x}$
 3: for $i = 0, 1, \ldots, \mathscr{I}$ do
 4:     $x \leftarrow Ax$
 5:     $x \leftarrow \frac{x}{||x||}$
 6:     $\lambda_{\text{NEW}} \leftarrow \frac{x^T A x}{x^T x}$
 7:     if $|\lambda_{\text{NEW}} - \lambda_{\text{OLD}}| < \varepsilon$ then
 8:         Break loop
 9:     end if
10:     $\lambda_{\text{OLD}} \leftarrow \lambda_{\text{NEW}}$
11: end for
12: $x^\star \leftarrow x$
13: $\lambda = \lambda_{\text{NEW}}$
```

Example 3.13 Let

$$A = \begin{pmatrix} -3 & 0 & 4 \\ 17 & 13 & -7 \\ 16 & 14 & -8 \end{pmatrix}$$

that has as greatest eigenvalue in absolute value $\lambda_1 = 10.93$ and has as respective eigenvector

$$u = \begin{pmatrix} -0.1967 \\ -0.7014 \\ -0.685 \end{pmatrix}.$$

We initialize x^0 by sampling from a normal distribution with mean 0 and standard deviation 0.25 obtaining $x^0 = [0.0285, -0.3772, -0.4016]$. The first iteration of the Power method then gives us

$$
\begin{aligned}
x^1 &= \frac{Ax^0}{||Ax^0||} \\
&= \frac{1}{2.8384} \begin{pmatrix} -1.6918 \\ -1.6094 \\ -1.6135 \end{pmatrix} \\
&= \begin{pmatrix} -0.5960 \\ -0.567 \\ -0.5684 \end{pmatrix} ,
\end{aligned}
$$

the second

$$
\begin{aligned}
x^2 &= \frac{Ax^1}{||Ax^1||} \\
&= \begin{pmatrix} -0.0259 \\ -0.7226 \\ -0.6907 \end{pmatrix} .
\end{aligned}
$$

We can then use the Rayleigh quotient to obtain an approximation $\tilde{\lambda}_1$ of λ_1:

$$
\begin{aligned}
\tilde{\lambda}_1 &= \frac{(x^2)^T A x^2}{(x^2)^T x^2} \\
&= \frac{7.1416}{1} \\
&= 7.1416 .
\end{aligned}
$$

After two iterations the Power Method thus has given us approximations x^2 of u and $\tilde{\lambda}_1$ of λ_1 with relative errors:

$$
\begin{aligned}
\frac{||x^2 - u||_2}{||u||_2} &= 0.17 \\
\frac{|7.1416 - 10.93|}{|10.93|} &= 0.35 .
\end{aligned}
$$

4 Numerical simulations

In this final Chapter of the thesis we wish to compare the various methods on real-world tasks. We implemented all the methods in Python 3 making extensive usage of the `numpy` package to operate on vectors and matrices as well as use the built-in `numpy.linalg.eig` function to have a benchmark for our methods.

The structure is as follows: in Section 4.1 we will plot the convergence rate of all the methods on a matrix obtained from randomly generated data. In Section 4.2 we will add Gaussian noise of increasing amplitude to this matrix in order to test the stability of the methods. Finally in Section 4.3 we will use Jacobi's and Oja-Sanger's methods to compute the principal component analysis of a data set composed by of photos of faces - we will obtain the so-called eigenfaces which can be used for the task of face-recognition.

4.1 Convergence of methods

In this section we test the convergence of the described methods which we implemented in Python. We used the `make_blobs` function from the `scikit-sklearn` library to generate 5000 samples in $\mathbb{R}^{100}$ picked from the sum of 3 multi-dimensional Gaussian distributions each with mean 0 and standard deviation 5. These samples are the rows of the data matrix $X \in \mathbb{R}^{5000\times 100}$; we will further call $A = X^T X$ the 100×100 covariance matrix of X.

For all methods we set a tolerance $\varepsilon = 1^{-7}$ and a maximum number of iterations $\mathscr{I} = 100$. We relied on `numpy`'s `eig` function (in the `linalg` module) to compute $\lambda_1, \lambda_2, \ldots, \lambda_n$, the exact eigenvalues of A, which we always reordered in decreasing order. We will instead denote with $\tilde{\lambda}_i^k$ the approximation of eigenvalue i given at iteration k by the considered method.

For Oja's and the Power method we considered the convergence of a single eigenvalue by looking at the value $|\lambda_1 - \tilde{\lambda}_1^k|$ for $k = 1, 2, \ldots, \mathscr{I}$. For Oja-Sanger's and Jacobi's method we further looked at this quantity for the second eigenvalue, for the last and second to last, as well as the cumulative convergence $\mathscr{K}^k = \left|\left|\Lambda - \tilde{\Lambda}^k\right|\right|_2$, where Λ and $\tilde{\Lambda}^k$ are vectors with the exact and approximated eigenvalues at iteration k respectively.

All the plots in this whole Chapter have a logarithmic scale on the y-axis to better distinguish the rapidly decreasing plots.

In Figure 4.1 the convergence of Oja's method for the largest eigenvalue is shown, while in Figure 4.2 the cumulative error for Oja-Sanger's method is shown - note that here at

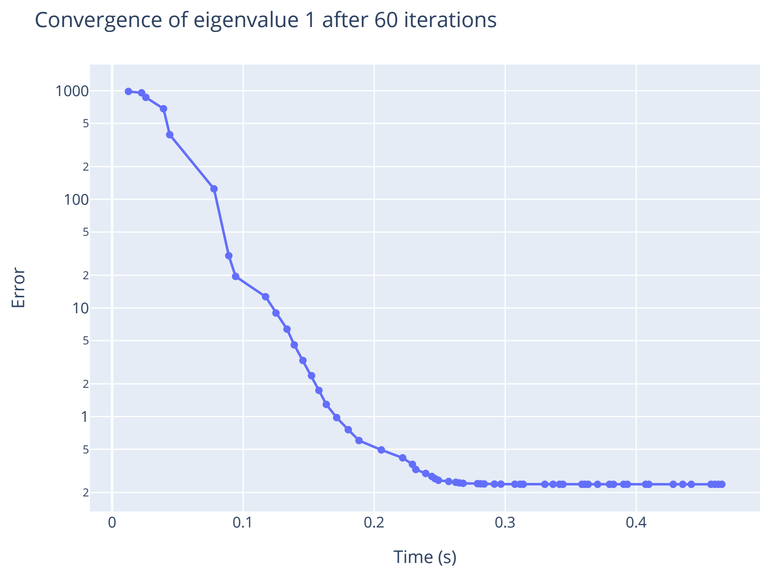


Figure 4.1: Convergence of $|\lambda_1 - \tilde{\lambda}_1^k|$ for Oja's method

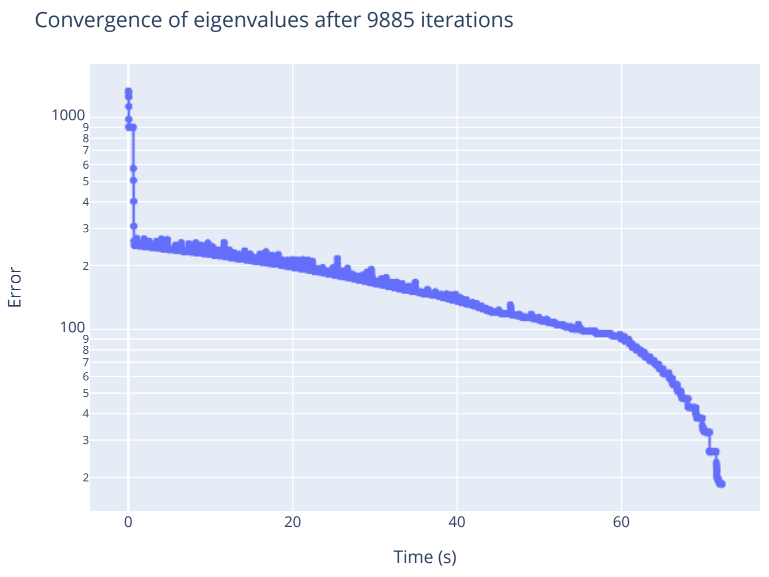


Figure 4.2: Convergence of $\mathscr{K}^k = ||\Lambda - \tilde{\Lambda}^k||_2$ for Oja-Sanger's method

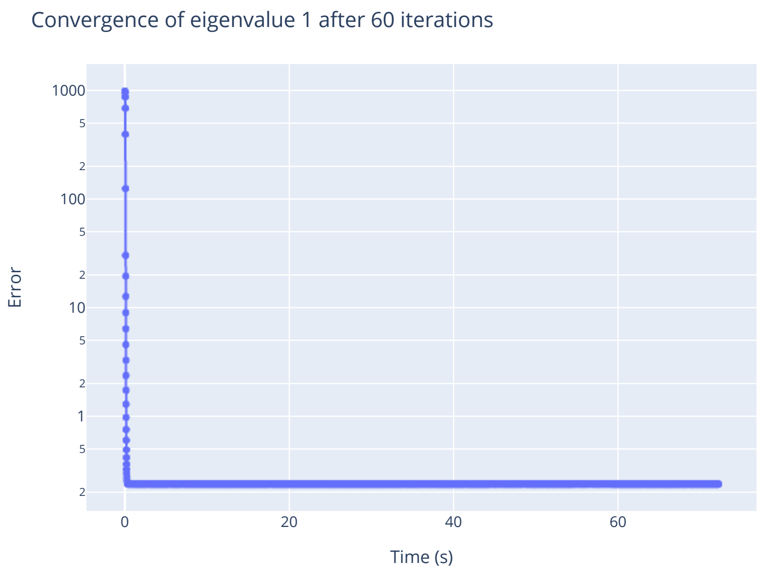


Figure 4.3: Convergence of $|\lambda_1 - \tilde{\lambda}_1^k|$ for Oja-Sanger's method

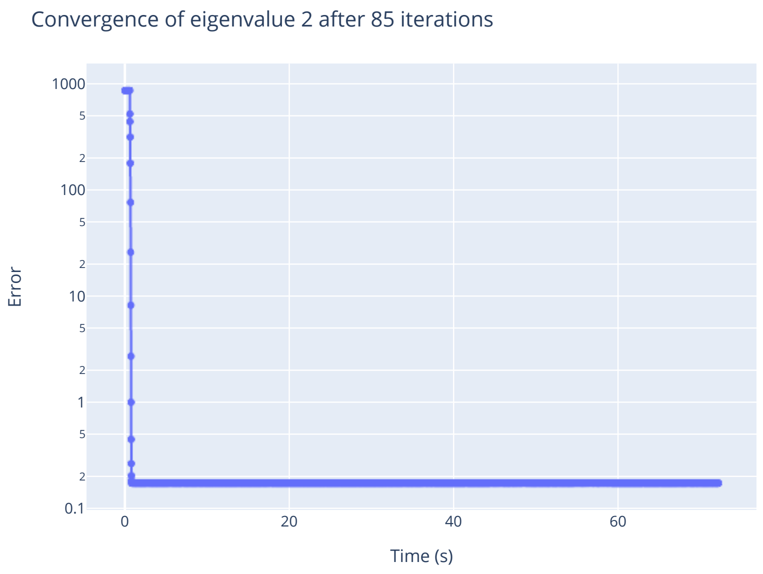


Figure 4.4: Convergence of $|\lambda_2 - \tilde{\lambda}_2^k|$ for Oja-Sanger's method

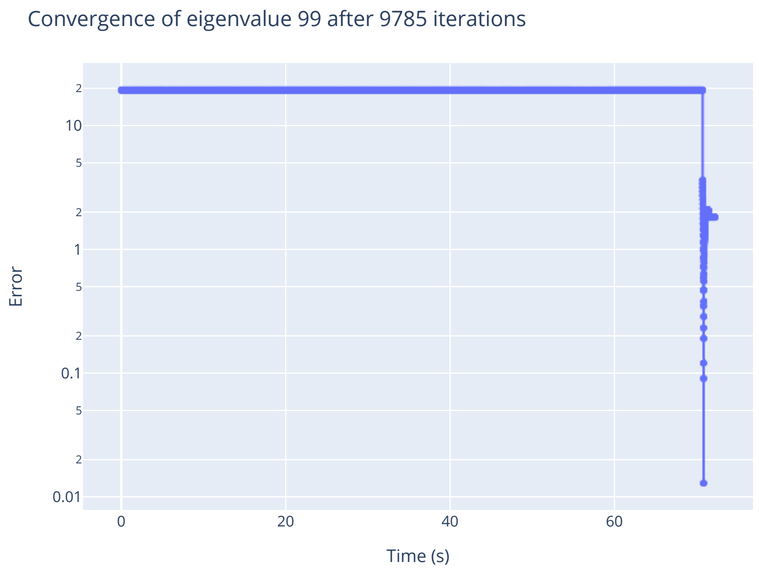


Figure 4.5: Convergence of $|\lambda_{99} - \tilde{\lambda}_{99}^k|$ for Oja-Sanger's method

most 100 iterations are done for each eigenvalue, so the total number of iterations is at most 10000. Because of the structure of Algorithm 2, the eigenvalues here converge one a time. This can be seen by looking at the convergence of individual eigenvalues: eigenvalues 1 and 2 (Figures 4.3 and 4.4 respectively) converge below the tolerance after few iterations, while eigenvalue 99 (Figure 4.5) starts converging only after the method finished with all the previous eigenvalues.

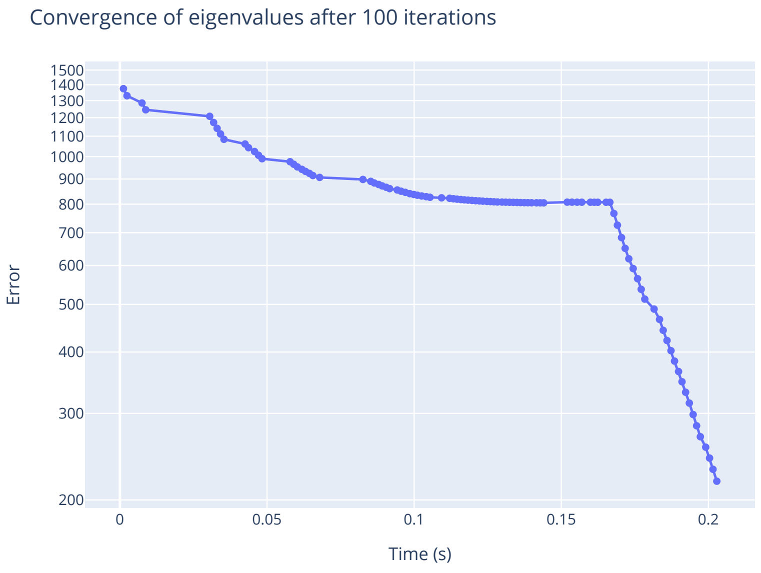


Figure 4.6: Convergence of $\mathscr{C}^k = ||\Lambda - \tilde{\Lambda}^k||_2$ for Jacobi's method

This is contrast with Jacobi's method that improves the approximation for all eigenvalues at each iteration. The convergence of the cumulative error for Jacobi's method can be

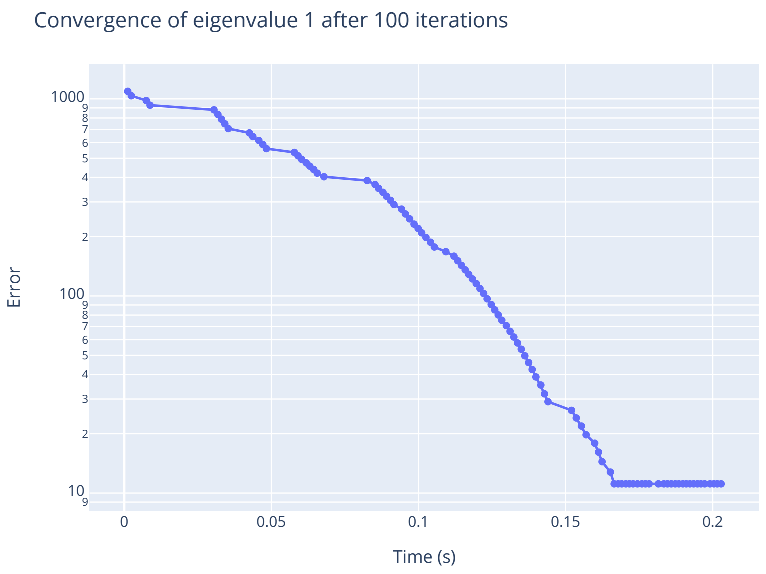


Figure 4.7: Convergence of $|\lambda_1 - \tilde{\lambda}_1^k|$ for Jacobi's method

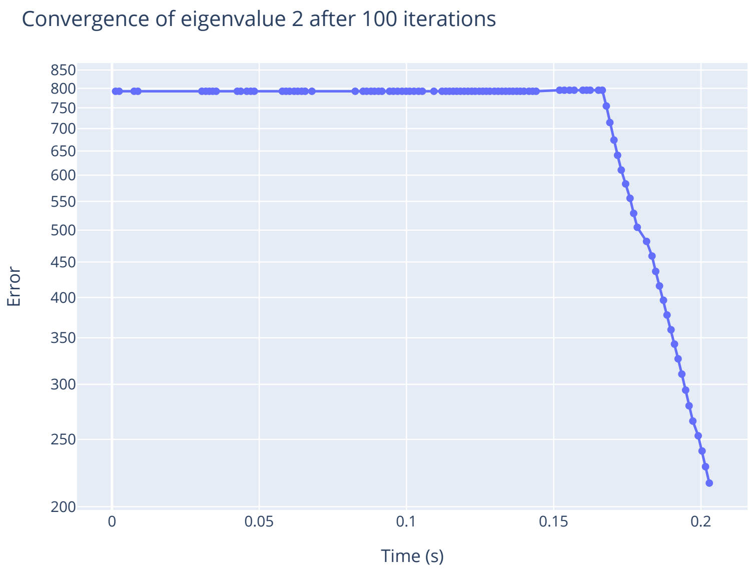


Figure 4.8: Convergence of $|\lambda_2 - \tilde{\lambda}_2^k|$ for Jacobi's method

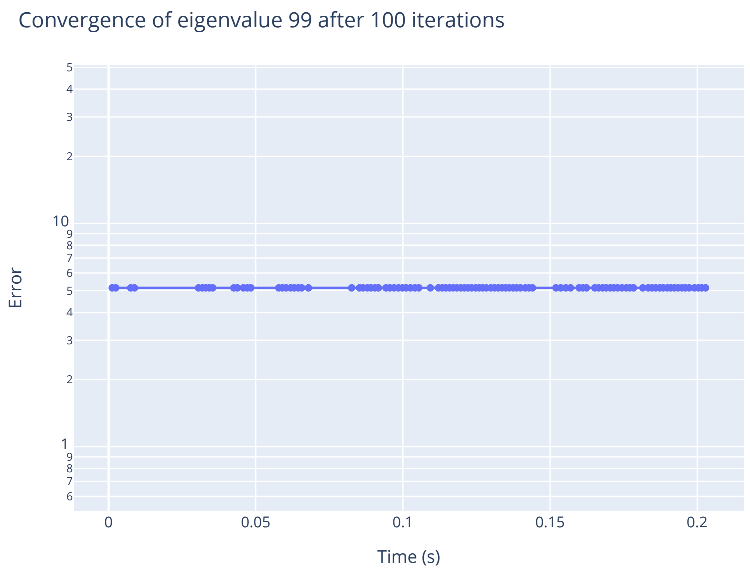


Figure 4.9: Convergence of $|\lambda_{99} - \tilde{\lambda}_{99}^k|$ for Jacobi's method

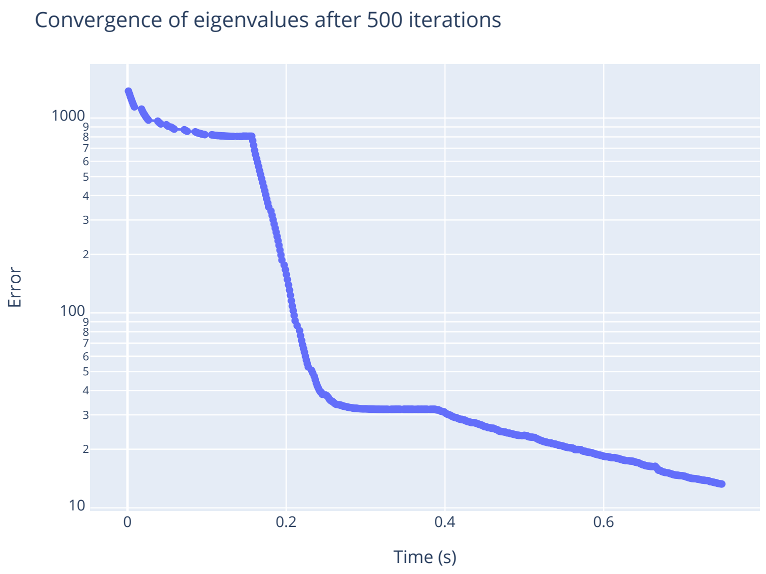


Figure 4.10: Convergence of $\mathscr{K}^k = ||\Lambda - \tilde{\Lambda}^k||_2$ for Jacobi's method with $\mathscr{I} = 500$

seen in Figure 4.6, while the convergence for eigenvalues 1, 2 and 99 can be seen in Figures 4.7, 4.8 and 4.9 respectively. Compared to Oja-Sanger's method, here the cumulative error is much higher but very rapidly decreasing. Out of all the methods, the Jacobi method seems to have the most unpredictable behavior, convergence of different eigenvalues having very different behavior (e.g. eigenvalue 99 does not converge at all, at least in the first 100 iterations) and in general with convergence going through various phases and plateaus. This becomes apparent if we raise the number of iterations $\mathscr{I}$ to 500: we obtain the plot in Figure 4.10, which shows how the method converges goes through 3 different phases with different convergence rates.

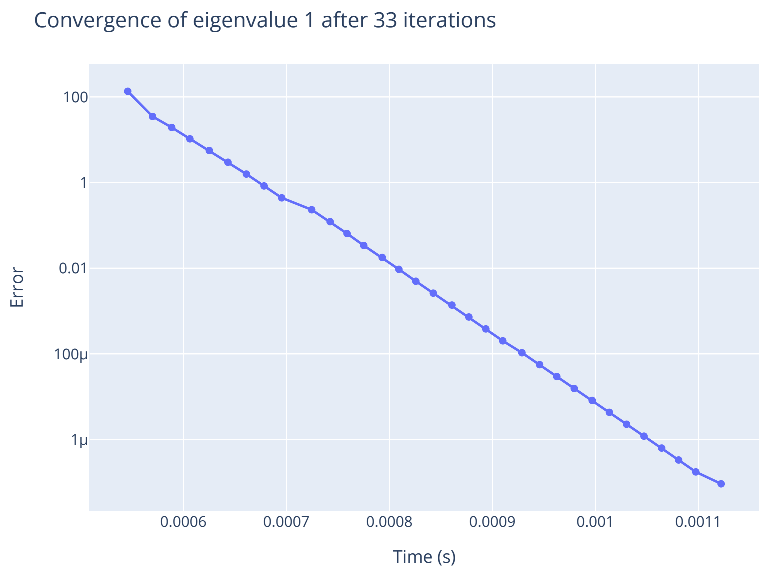


Figure 4.11: Convergence of $|\lambda_1 - \tilde{\lambda}_1^k|$ for the Power method

Finally in figure 4.11 we plot the convergence of the Power method, that deceases steadily and needs only 33 iterations to fall below tolerance.

4.2 Stability

In order to experimentally test the stability of the various methods, we took the same matrix $A \in \mathbb{R}^{100\times 100}$ from the previous section and computed a noisy version $\tilde{A}^\sigma = A + E^\sigma$, where the entries of E^σ are extracted from a normal distribution with mean 0 and variance σ, with $\sigma = 0.001, 0.1, 1, 10$. We then computed $\tilde{\lambda}^{k,\sigma}$ by applying the various methods on $\tilde{A}^\sigma$ and plotted $|\lambda_1 - \tilde{\lambda}_1^{k,\sigma}|$ for Oja's and the Power method and $\left|\left|\Lambda - \Lambda^{\tilde{k},\sigma}\right|\right|_2$ for Oja-Sanger's and Jacobi's method (where $\tilde{\Lambda}^{k,\sigma} = (\tilde{\lambda}_1^{k,\sigma}, \tilde{\lambda}_2^{k,\sigma}, \dots \tilde{\lambda}_n^{k,\sigma})$); see Figures 4.12 - 4.15. If the difference between the exact eigenvalues and the eigenvalues computed on $\tilde{A}$ converges to 0 with the iterations, thanks to equation 2.8 and the considerations in that Section we can conclude the method is stable.

For $\sigma = 0.001$ and less so for 0.1 all methods still have a decreasing error. It is interesting to note that in some cases with a higher value of σ the convergence can be actually better: this is probably due to accumulation of numerical errors and is a phenomenon that would net present itself if one did this test by plotting a statistic (such as the mean) of multiple runs for different versions of the noisy matrix (for the same value of σ). Finally we note that the Power Method is the only one that also for $\sigma = 10$ still has decreasing error, though it does not approach the tolerance, suggesting (together with the predictability of its convergence plot in the previous Section) it is the most numerically stable.

From these plots we can conclude that the most stable methods are the Power Method if one is interested only in the greatest eigenvalue and Oja-Sanger's if one is interested in all the eigenvalues of a covariance matrix.

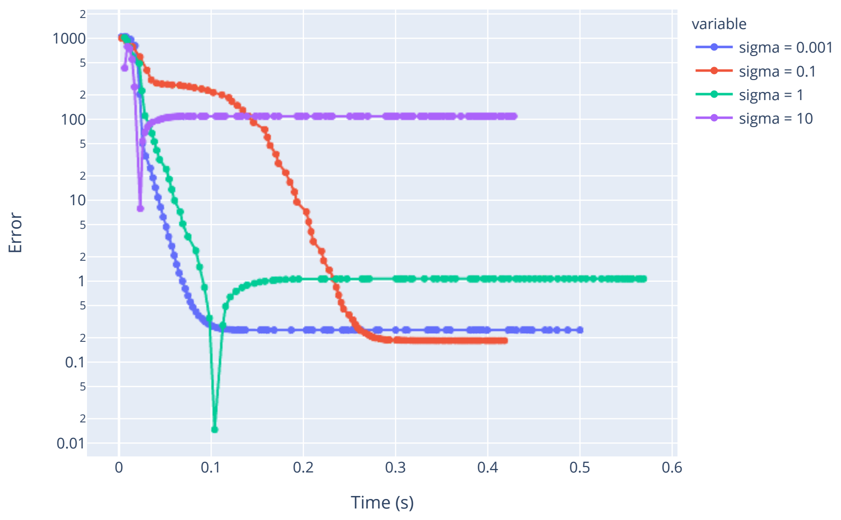


Figure 4.12: Convergence of $|\lambda_1 - \tilde{\lambda}_1^{k,\sigma}|$ for Oja's method.

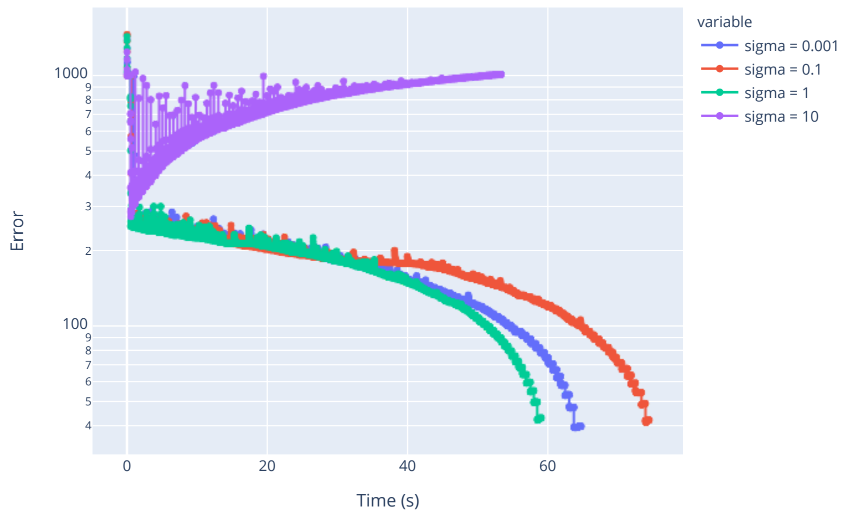


Figure 4.13: Convergence of $\mathscr{K}^k = \left|\left|\Lambda - \tilde{\Lambda}^{k,\sigma}\right|\right|_2$ for Oja-Sanger's method.

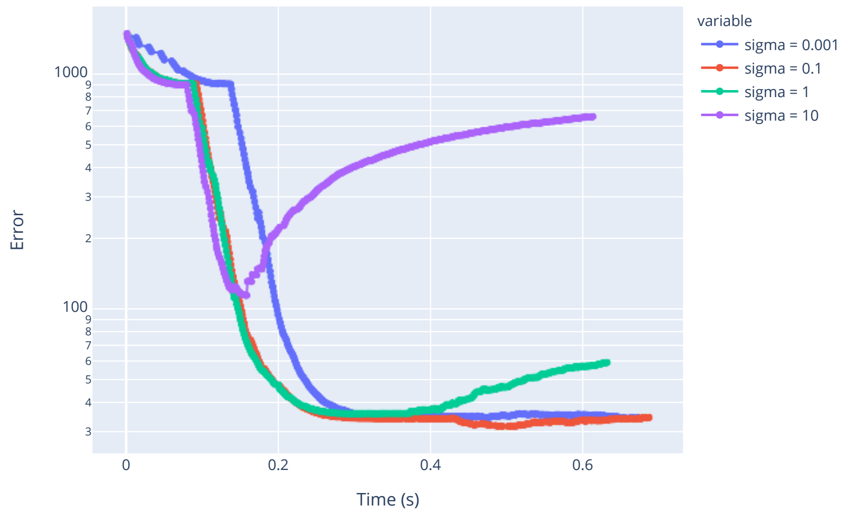


Figure 4.14: Convergence of $\mathscr{K}^k = \left|\left|\Lambda - \tilde{\Lambda}^{k,\sigma}\right|\right|_2$ for Jacobi's method.

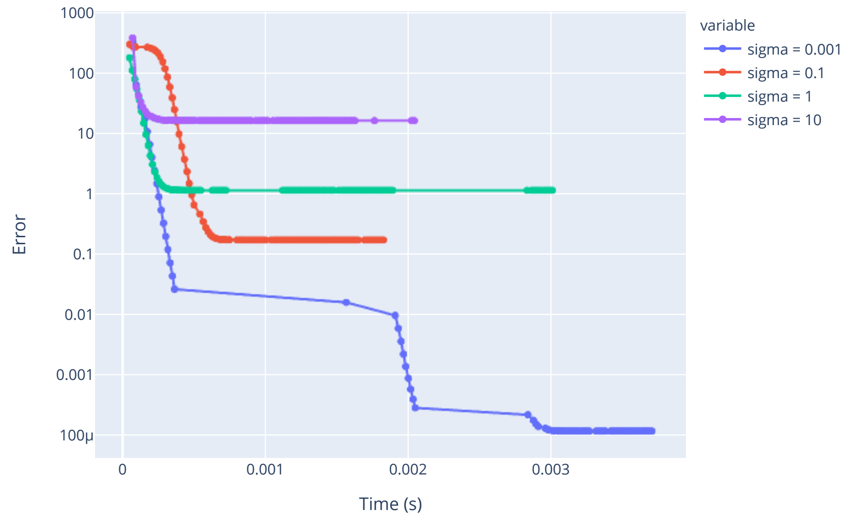


Figure 4.15: Convergence of $|\lambda_1 - \tilde{\lambda}_1^{k,\sigma}|$ for the Power method.

4.3 Computing eigenfaces

In this section we will apply Oja-Sanger's and Jacobi's method to a real-world data set, the ATT (American Telephone and Telegraph) face data set [15][2], consisting of 400 gray scale 112×92 images; see Figure 4.16. We resized all images to 56×46 using `skimage.transform.resize` function from the `scikit-image` library and vectorized them by row stacking, obtaining a $N \times d$ data matrix X with $N = 400$ and $d = 2576 = 56 \cdot 46$, where each row corresponds to an image. We further scaled the matrix X to have zero mean using the `sklearn.preprocessing.StandardScaler` function from the `scikit-learn` library.

Let the sample covariance matrix $C \in \mathbb{R}^{d\times d}$ associated to the data matrix X have eigenvalues $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_d|$ with corresponding eigenvectors $u_1, u_2, \ldots, u_d$. An *eigenface* is simply a 56×46 image that when vectorized by row stacking is equal to one of these eigenvectors, and thus it corresponds to a principal component of X; refer to Section 2.3 for more details. We will interchangeably use the term eigenface, eigenvectors of C or principal component of X in the following.

The eigenfaces can be used to obtain a compressed representation of the original images, simply by projecting an input image $v \in \mathbb{R}^{2576}$ onto the subspace generated by the first $K < 2576$ eigenfaces. In fact, as explained in Section 2.3.1, we can compute the K-dimensional vector $v_{|K}$ and from this $\pi^K(v)$, which is an approximation of v obtained as linear combination of $u_1, u_2, \ldots, u_K$. This can be useful if for example we consider the problem of face recognition, which is the classification problem the authors that in-

[2] available at `https://git-disl.github.io/GTDLBench/datasets/att_face_dataset/`

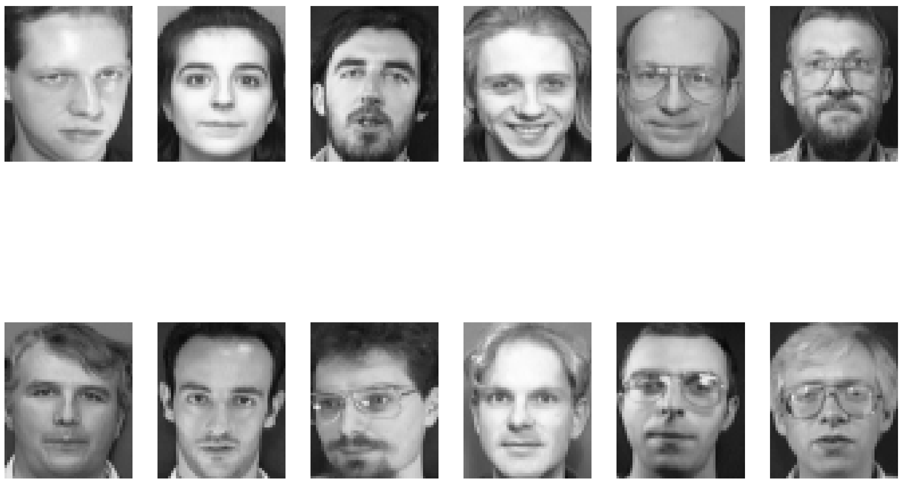


Figure 4.16: Some example images from the ATT faces data set

troduced the concept of eigenfaces had in mind [20]: given a new face image v, we can associate it to one of the 400 faces $f_1, f_2, \dots, f_{400}$ used for training by comparing the K coefficients of the projection $\pi^K(v)$ with those of $\pi^K(f_1), \pi^K(f_2), \dots, \pi^K(f_{400})$ and classifying v as face

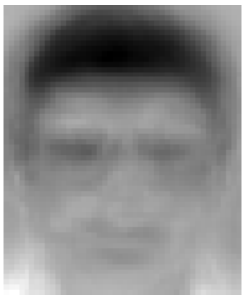
$$\tilde{i} = \arg \min_{i=1,2,\dots,400} \left|\left|\pi^K(v) - \pi^K(f_i)\right|\right|_2 .$$

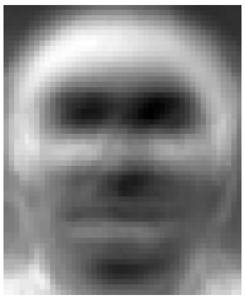
A smaller K means a worse approximation but a faster inference in the classification task.

We set $K = 10$ and computed the eigenfaces with both Oja-Sanger's and Jacobi's method. The advantage of the former is that, by modifying line 2 of Algorithm 2, we can stop the iterations after exactly K eigenvectors are computed, since we don't need the rest.

Oja Sanger took 23 minutes and 52 seconds to compute the first 10 principal components, with a total of 31369 iterations. In Figure 4.17 the first 3 eigenfaces are shown, while in Figure 4.18 3 original faces f_1, f_2 and f_3 are shown next to their projections $\pi^{10}(f_1), \pi^{10}(f_2), \pi^{10}(f_3)$.

Jacobi's method took 26 minutes and 28 seconds to compute the 2000 iterations. In Figure 4.19 the first 3 eigenfaces are shown, while in Figure 4.20 3 original faces f_1, f_2 and f_3 are shown next to their projections $\pi^{10}(f_1), \pi^{10}(f_2), \pi^{10}(f_3)$.


(a) u_1


(b) u_2

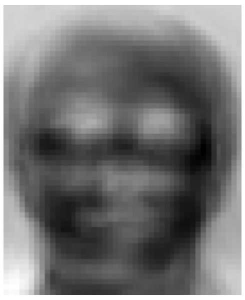

(c) u_3

Figure 4.17: The first 3 eigenfaces as computed by Oja-Sanger's method

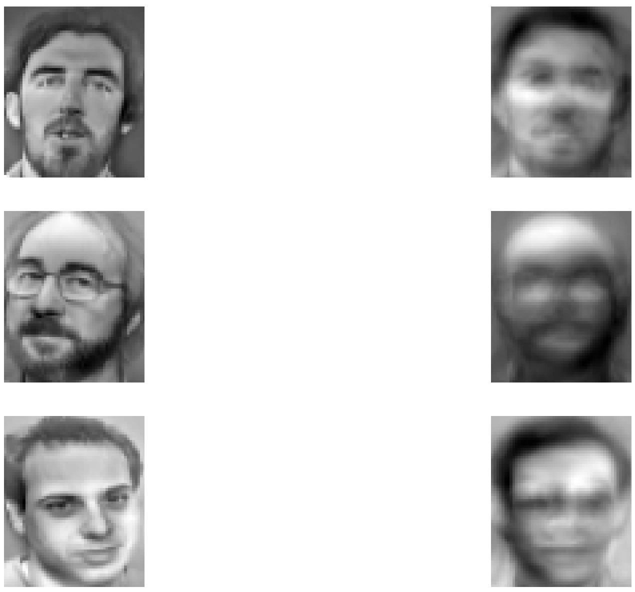
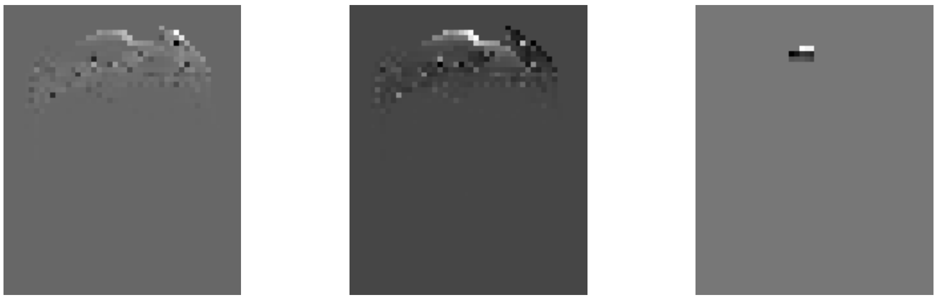


Figure 4.18: Original faces and their projection $\pi^K(f_i)$ on the first $K = 10$ principal components compute with the Oja-Sanger method



(a) u_1 (b) u_2 (c) u_3

Figure 4.19: The first 3 eigenfaces as computed by Jacobi's method

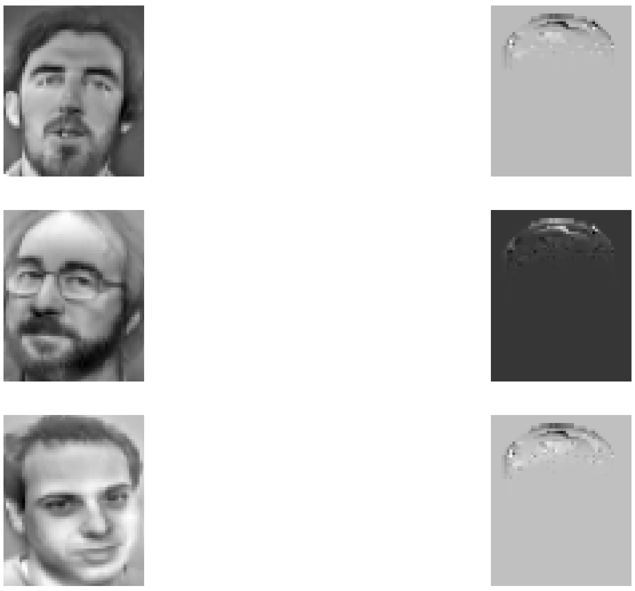


Figure 4.20: Original faces and their projection $\pi^K(f_i)$ on the first $K = 10$ principal components computed with the Jacobi's method

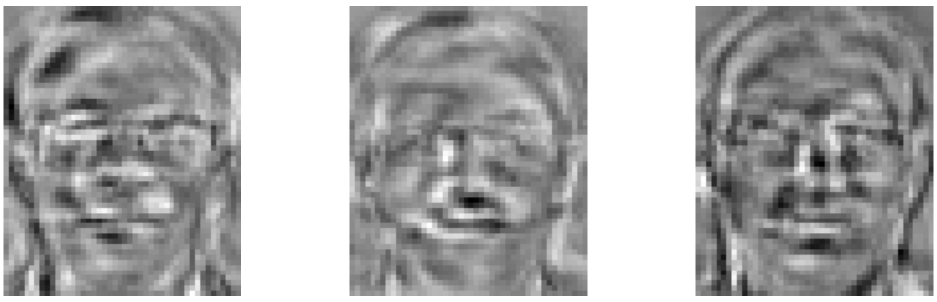

(a) u_1 (b) u_2 (c) u_3

Figure 4.21: The first 3 eigenfaces as computed by Jacobi's method with 100 inverse iterations applied afterwards

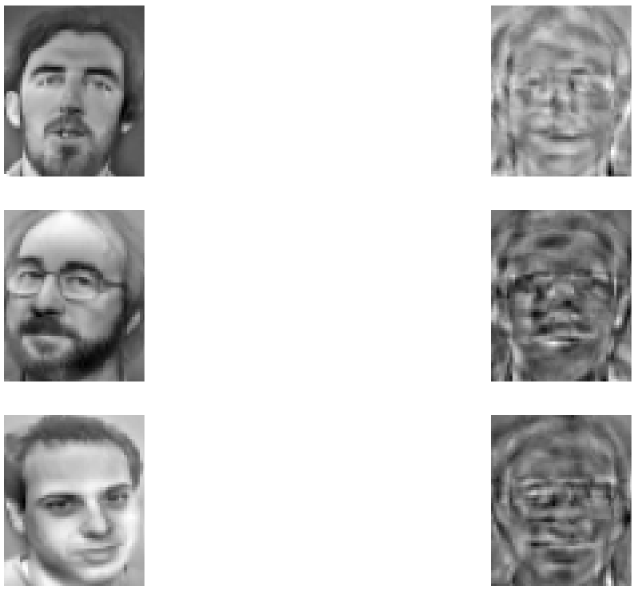


Figure 4.22: Original faces and their projection $\pi^K(f_i)$ on the first $K = 10$ principal components as computed with the Jacobi's method with 100 additional inverse iterations

We can observe that the eigenfaces obtained with Jacobi's method and the corresponding projections are much worse in quality with those obtained with Oja-Sanger's. In an attempt to try and improve the approximation of the first K eigenvectors, we applied 100 inverse iterations: supposing the eigenvalues $\tilde{\lambda}_i$ obtained with Jacobi, for $i = 1, 2, \dots, K$, are good approximations of the exact eigenvalues, we applied the Power method to $(C - \tilde{\lambda}_i)^{-1}$ choosing as initialization value the approximation of the corresponding eigenvector obtained by Jacobi. As seen in Section 3.3, this will converge to the eigenvector of C with eigenvalue closest to $\tilde{\lambda}_i$, which we expected to be an improvement over the eigenvector given by Jacobi's method: results are shown in Figures 4.21 and 4.22. The inverse iterations added a negligible 14 seconds of computation and improved visibly the quality of the eigenfaces, though they still are not as good as those obtained with Oja-Sanger's method. We thus conclude that for this particular application Oja-Sanger's method is better: it allows to focus the computational effort on only the eigenvectors with greatest eigenvalues, which are the only ones needed for projection of the images onto the principal components. This theoretical consideration is reflected in the empirical evidence: in less time it produces higher quality eigenfaces.

4.4 Denoising images

In real-world applications it often happens that one has access only to noisy data (due for example to imprecise measuring instruments) and wishes to *denoise* it, i.e. obtain a transformation of the noisy data that is more closer to the hypothetical original data, which in this scenario is not available. To evaluate a denoising method one usually constructs a synthetic experiment, where some data set is considered the original data, artificial noise is added to it and then the denoising method is applied to this noisy data. We can thus measure (using for example the $||\cdot||_2$ norm) if the denoised data is closer to the original than the noisy data is.

Principal component analysis is sometimes used for denoising: if we suppose the input data to be affected by independent and uncorrelated noise, this will affect equally all the principal components. Then by projecting the data samples onto a small number of components we can get rid of all the noise in the other components. In this Section we will apply this procedure on the first 2000 images of the MNIST (Modified National Institute of Standards and Technology) data set of handwritten digits [9][3]. We used only part of the data set in order to speed up the computation of the covariance matrix which is very long when computed for the whole 70000 images in the data set. Since this is again a case where we need to compute the first $K > 1$ eigenvectors of a covariance matrix, we can test this method using only the Oja-Sanger and Jacobi methods.

We will call the images in the data set $x^1, x^2, \dots, x^N$ with $N = 2000$; each image is 28×28 pixels and each pixel is an integer value from 0 to 255, i.e. $x^i \in \{n \in \mathbb{N} \,|\, n \leq 255\}^{28\times 28}$ for all $i = 1, 2, \dots, N$. With the exact same procedure as in the previous section, we stacked the vectorized images as rows of a data matrix $X \in \{n \in \mathbb{N} \,|\, n \leq 255\}^{N\times d}$ with $d = 784 = 28 \cdot 28$, which we further scaled to have zero mean. We then computed $\tilde{X}$ by adding Gaussian noise with zero mean and standard deviation 50 to X; we will call the rows of $\tilde{X}$ the *noisy images*.

We used first Oja-Sanger's and then Jacobi's method to compute $\tilde{u_1}, \tilde{u_2}, \dots, \tilde{u_N}$, approximations of the eigenvectors of $\tilde{C}$, the sample covariance matrix corresponding to $\tilde{X}$, with corresponding eigenvalues $|\tilde{\lambda}_1| \geq |\tilde{\lambda}_2| \geq \dots \geq |\tilde{\lambda}_N|$. We then computed the projections $\pi^K(\tilde{x}^1), \pi^K(\tilde{x}^2), \dots, \pi^K(\tilde{x}^N)$ with $K = 20$ (see Section 2.3.1) and compared the *noise error* $\left|\left|\tilde{x}^i - x^i\right|\right|_2$ with the *projection error* $\left|\left|\pi^K(\tilde{x}^i) - x^i\right|\right|_2$. If the denoising procedure works, the latter should be smaller than the former.

For Oja-Sanger's method, again we can change line 2 of Algorithm 2 and stop the iterations after exactly K eigenvectors are computed, since we do not need the rest. Setting a maximum number of iterations $\mathscr{I} = 1000$, a tolerance $\varepsilon = 1^{-9}$ and a learning rate $\alpha = 1^{-9}$, Oja-Sanger's method required exactly 11 minutes to compute the first 20 principal components. The first 6 principal components can be seen in Figure 4.23

[3] available at `https://www.openml.org/d/554`

while some original, noisy and denoised images can be seen in Figure 4.24 along with the respective noise and projection errors.

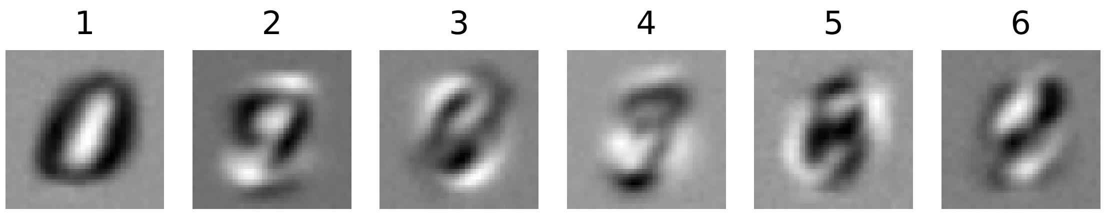


Figure 4.23: The first 6 principal components $u_1, u_2, \dots, u_6$ obtained from the MNIST data set with Oja-Sanger's method.

For Jacobi's method we set a maximum number of iterations $\mathscr{I} = 5000$ and a tolerance $\varepsilon = 1^{-9}$. The method required 18 minutes and 7 seconds to compute. The first 6 principal components can be seen in Figure 4.25 while some original, noisy and denoised images can be seen in Figure 4.26 along with their respective noise and projection errors.

We can see that for Oja-Sanger the denoising procedure worked: the projection errors are lower than the noise errors. For Jacobi's method however, which also required more computation time, the projection error is actually worse, so we can say the denoising procedure failed.

Once again we can then conclude that, for computing the Principal Component Analysis, Oja-Sanger's method is to be preferred over Jacobi's: by concentrating the computation time on only the needed K eigenvectors, it can achieve better results in less time.

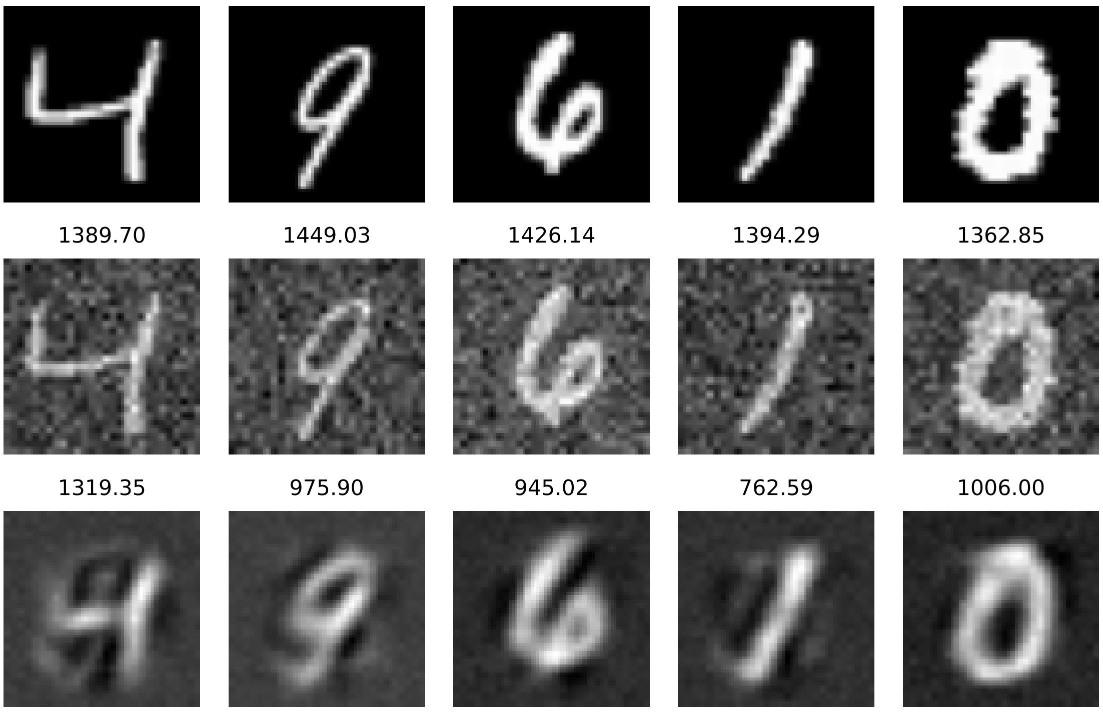


Figure 4.24: For 5 randomly selected indexes $i = 1, 2, \ldots, 2000$, the three rows show x^i, $\tilde{x}^i$ and $\pi^{20}(\tilde{x}^i)$ respectively, when using the Oja-Sanger's method. The numbers above the second and third row are the noise and projection error respectively.

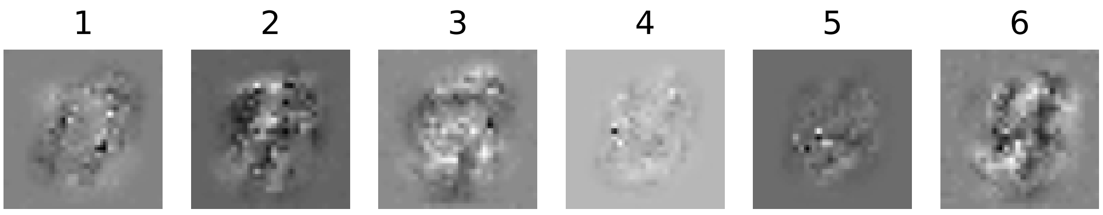


Figure 4.25: The first 6 principal components $u_1, u_2, \ldots, u_6$ obtained from the MNIST data set with Jacobi's method.

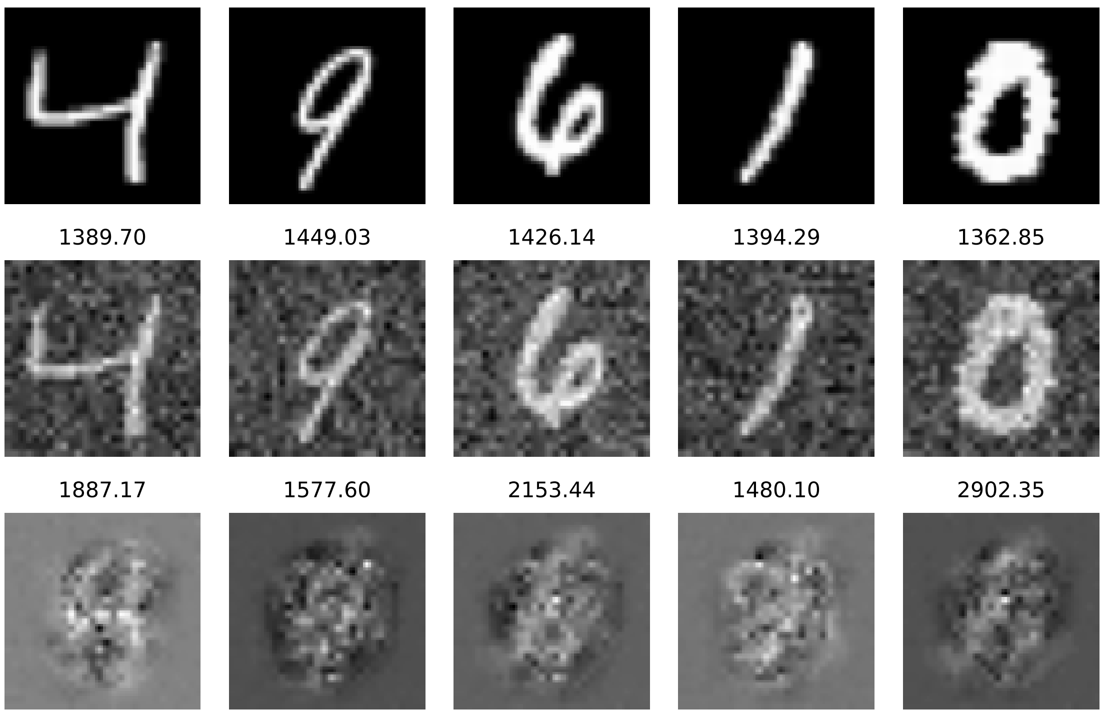


Figure 4.26: For 5 randomly selected indexes $i = 1, 2, \ldots, 2000$, the three rows show x^i, $\tilde{x}^i$ and $\pi^{20}(\tilde{x}^i)$ respectively, when using Jacobi's method. The numbers above the second and third row are the noise and projection error respectively.

4.5 Computing eigenvector centrality

In this section we want to compute the eigenvector centrality measure (see Section 2.4) for a graph generated from internal web links on the popular website Reddit, which is composed of many different communities known as subreddits. Each vertex in the graph corresponds to a vertex and there is an edge from vertex v to w if in some post on subreddit v there is a link to subreddit w. The data set we used was created by the authors of [7][4] by extracting the data from the Reddit website between January 2014 and April 2017; the generated graph consists of 35776 vertices and 124330 edges.

We read the data and created a graph object using the `networkx` Python package; we then used the `networkx.DiGraph.in_degree` method to compute the degree centrality of the vertices. To compute the eigenvector centrality we need to compute the eigenvector corresponding to the largest eigenvalue of the adjacency matrix A; since this is not symmetric and not a covariance matrix, the only method we can use here is the Power method. We therefore set the tolerance $\varepsilon = 1^{-12}$ and run the Power method, which achieved the tolerance in 59 iterations that took 1.3 seconds. The results for the most central 20 vertices for both the degree and eigenvector centrality are visible in Table 4.1.

[4] available at `https://snap.stanford.edu/data/soc-RedditHyperlinks.html`

Subreddit	Degree centrality	Subreddit	Eigenvector centrality
askreddit	2161	subredditdrama	0.2848
iama	1646	drama	0.1811
pics	953	copypasta	0.1632
videos	879	circlejerkcopypasta	0.1610
todayilearned	816	shitliberalssay	0.1516
funny	757	conspiracy	0.1449
writingprompts	717	circlebroke	0.1439
worldnews	661	outoftheloop	0.1411
mhoc	595	bestofoutrageculture	0.1302
outoftheloop	592	justunsubbed	0.1194
gaming	584	self	0.1143
news	581	subredditcancer	0.1127
leagueoflegends	577	nostupidquestions	0.1079
pcmasterrace	478	the_donald	0.1056
explainlikeimfive	457	legaladvice	0.1053
subredditdrama	454	karmacourt	0.1049
technology	442	askreddit	0.1042
science	442	hailcorporate	0.1003
adviceanimals	440	help	0.0995
politics	410	circlebroke2	0.0942

Table 4.1: Degree and eigenvector centrality (computed with the Power method) for the most central 20 vertices in the Reddit data set.

It can be seen that the centrality measures differ substantially, with few subreddits appearing in both lists. The "askreddit" subreddit is the first in degree centrality, but only the 17-th for eigenvector centrality, while "outoftheloop" is 10-th for degree and 8-th for eigenvector centrality. This is indeed one of those cases mentioned in Section 2.4 where the degree centrality is not very useful: it simply ranks the subreddits by their popularity (as in by how many different subreddits they are linked to). In fact "askreddit", "iama", "pics" and "videos" are among the most popular subreddits of the whole website. The eigenvector centrality instead is more nuanced, and selects subreddits that are linked to by other important subreddits, even though they may be overall less popular.

5 Conclusion

In this thesis we introduced the eigenvalue problem for a matrix $A \in \mathbb{R}^{m\times n}$, and in particular 4 different methods to numerically compute an approximation of these. After introducing some fundamental concepts in Chapter 2, we moved on to describe the methods themselves in Chapter 3.

We introduced the Oja and Oja-Sanger's methods given a brief background of the Hebbian learning framework they originated in. These methods can be used in the case of A being the sample covariance matrix of a data matrix $X \in \mathbb{R}^{N\times d}$, which has N d-dimensional samples as rows. In other words, they can be used to compute eigenvalues and eigenvectors of $X^T X$, with Oja computing only the greatest eigenvalue (and respective eigenvector), while Oja-Sanger giving the possibility to compute the first $K \leq d$ eigenvectors and respective eigenvalues.

We then described Jacobi's method, that can be used in the case of A being a symmetric matrix. We introduced the Givens rotations which are at the heart of this method and we studied how they can be used to nullify off-diagonal elements. We observed how, differently than Oja-Sanger's method, Jacobi's method potentially improves the approximation of all eigenvalues and eigenvectors at each iteration, simultaneously.

Then we described the Power method, which is perhaps the simplest of all the methods and can be used to compute the dominant eigenvalue and respective eigenvector of any matrix A

Finally in Chapter 4 we tested Python implementations of the 4 methods in 5 different contexts: we studied the convergence on a randomly generated matrix, we added Gaussian noise to this matrix and tested the stability of the methods, we tested Jacobi's and Oja-Sanger's method to compute the Principal Component Analysis for the so-called eigenfaces and for denoising images and finally we used the Power Method to compute the eigenvector centrality of a graph.

We observed that the Power method is the most stable and has the most predictable convergence, while Jacobi's method has different convergence curves for different eigenvalues (with some not converging at all in the limited number of iterations we tested) and it seems to change convergence behavior throughout its iterations.

For the eigenfaces and denoising applications (both of which rely on Principal Component Analysis) in particular Oja-Sanger clearly performed faster and better than Jacobi's method, even when we helped this with some inverse iterations to improve the quality of the eigenfaces.

Bibliography

[1] Allen-Zhu, Z.; Li, Y.: First Efficient Convergence for Streaming k-PCA:a Global, Gap-Free, and Near-Optimal Rate

[2] Beilina, L., Karchevskii, E., & Karchevskii, M. (2017). Numerical linear algebra: Theory and applications. Springer International Publishing.

[3] Ciarlet, P. G., Ciarlet, P. G., Miara, B., Thomas, J. M. (1989). Introduction to numerical linear algebra and optimisation. Cambridge University Press.

[4] Hertz, J. A. (2018). Introduction to the theory of neural computation. CRC Press.

[5] Higham, N. (2020). What Is a Condition Number? Blog post

[6] Kelleher, J. D. (2019). Deep learning. MIT press.

[7] Kumar, Srijan et al. (2018). Community interaction and conflict on the web. Proceedings of the 2018 World Wide Web Conference on World Wide Web

[8] Lipschutz, S., & Lipson, M. L. (2018). Schaum’s Outline of Linear Algebra. McGraw-Hill Education.

[9] LeCun, Y. & Cortes, C. (2010). MNIST handwritten digit database.

[10] Newman, M. (2018). Networks. Oxford university press.

[11] Olsauhsen, B. (2012). Linear Hebbian learning and PCA. Lecture notes

[12] Oja E.: A simplified neuron model as a principal component analyzer. J Math Biol, 1982, 15: 267 - 273

[13] Pishro-Nik, H. (2016). Introduction to probability, statistics, and random processes.

[14] Roy, R., Chattopadhyay, M. (2013). Iterative methods for eigenvalue problem/

[15] Samaria, Ferdinando S., and Andy C. Harter (1994). "Parameterisation of a stochastic model for human face identification." Proceedings of 1994 IEEE workshop on applications of computer vision. IEEE.

[16] Shilov, G. E. (2012). Linear Algebra. United States: Dover Publications.

[17] Smith, L.I (2002). A tutorial on Principal Components Analysis, lecture notes.

[18] Stoer, J., Bulirsch, R. (2002). Introduction to Numerical Analysis. Springer.

[19] Strang, G. (1998). Introduction to linear algebra. United States: Wellesley-Cambridge Press.

[20] Turk, M. and Pentland, A.P. Face recognition using eigenfaces. Computer Vision and Pattern Recognition, 1991. Proceedings CVPR'91., IEEE Computer Society Conference on 1991

[21] Vlachas, P. (2019). Oja's rule: Derivation, Properties. Lecture notes Project Report

Appendix A: Python code

A.1 Base class

```
import numpy as np
import time
import datetime as dt
import pandas as pd

class BaseMethod():
    def __init__(self, **kargs):
        if 'maxiter' in kargs:
            self.maxiter = int(kargs['maxiter'])
        else:
            self.maxiter = None
        if 'tolerance' in kargs:
            self.tolerance = kargs['tolerance']
        else:
            self.tolerance = None
        self._eigu = []  # list with computed eigenvalues, one per
   iteration
        self._eigv = []  # list with computed eigenvectors, one per
    iteration
        self.eigu = None  # computed eigenvalues
        self.eigv = None  # computed eigenvector matrix
        self.exact_eigu = None
        self.exact_eigv = None
        self._times = []
        self.time = None
        self.iterations = []
        self.tot_iterations = 0
        self.debug = False
        self.reorder = True
        self.normalize_eigv = False
        self.save_data_for_all_iterations = True

        # set this to True to check the found eigenvalues against
   numpy's np.linalg.eig - this can be SLOW
        self.check_result = False

    def fit_transform(self, X):
        self.mat_rows, self.mat_cols = X.shape
        self._fit_transform(X)
        self.n_computed_eigu = len(self.eigu)
        if self.normalize_eigv:
            eigvs = self.eigv
            for idx, eigv in enumerate(eigvs.T):
                norm = np.linalg.norm(eigv)
                self.eigv[:, idx] = eigv/norm
```

```python
        assert self.n_computed_eigu == self.eigv.shape[1]
        if self.n_computed_eigu > 1 and self.reorder:
            eigu_argsort = np.argsort(np.abs(self.eigu))[::-1]
            self.eigu = self.eigu[eigu_argsort]
            self.eigv = self.eigv.T[eigu_argsort].T
            for it in range(len(self._eigu)):
                eigu = self._eigu[it]
                eigv = self._eigv[it]
                self._eigu[it] = eigu[eigu_argsort]
                self._eigv[it] = eigv.T[eigu_argsort].T

    def _compute_exact(self, X):
        self.exact_eigu, self.exact_eigv = np.linalg.eig(X)
        eigu_argsort = np.argsort(self.exact_eigu)[::-1]
        self.exact_eigu = self.exact_eigu[eigu_argsort]
        self.exact_eigv = self.exact_eigv.T[eigu_argsort].T

    def _check_exact_ordering(self, X):
        rows, cols = self.exact_eigv.shape
        for i in range(cols):
            v = self.exact_eigv[:, i]
            l = self.exact_eigu[i]
            print(f"||Av_{i} - lv_{i}|| = {np.linalg.norm(X.dot(v) - l*v)}")

    def pprint_result(self):
        fmtD = "%20s: %5d\n"
        fmtF = "%20s: %5.10f\n"
        print(f"RESULTS - {dt.datetime.utcnow().isoformat()}")
        print(
            fmtD % ('N. Eigv computed', self.n_computed_eigu),
            fmtD % ('Iterations', self.tot_iterations),
            fmtF % ('Time (tot)', self.time),
            fmtF % ('Time (avg)', np.average(self._times))
        )
        if self.exact_eigu is not None:
            toterr = 0
            totverr = 0
            for i in range(self.n_computed_eigu):
                exu = self.exact_eigu[i]
                eu = self.eigu[i]
                toterr += (exu - eu)**2
                exv = self.exact_eigv[:, i].reshape(self.mat_cols, 1)
                ev = self.eigv[i]
                totverr += np.linalg.norm(exv - ev)**2
            toterr /= np.sum(self.exact_eigu**2)
            totverr /= np.sum([np.linalg.norm(v) **
                               2 for v in self.exact_eigv.T])
            print(
                fmtF % ("Eigu rel. error", np.linalg.norm(toterr)),
                fmtF % ("Eigv rel. error", totverr),
            )

    def get_time_error_df(self, eigidx: int = 0):
        timearr = np.cumsum(self._times).reshape(self.tot_iterations, 1)
        eig_list = [alleigs[eigidx] for alleigs in self._eigu]
        errarr = np.abs(np.array(eig_list) -
                        self.exact_eigu[eigidx]).reshape(self.tot_iterations, 1)
        return pd.DataFrame(np.hstack((timearr, errarr)), columns=['Time (s)', 'Error'])

    def get_cum_time_error_df(self):
        timearr = np.cumsum(self._times).reshape(self.tot_iterations, 1)
        cumerrarr = np.vstack(
            [np.linalg.norm(eigs - self.exact_eigu) for eigs in self._eigu])
        return pd.DataFrame(np.hstack((timearr, cumerrarr)), columns=['Time (s)', 'Error'])

    def compute_eigv_with_inverse_iteration(self, X, neigvs=None):
        import utils
        if neigvs is None or neigvs > self.n_computed_eigu:
            neigvs = self.n_computed_eigu
        start = time.time()
        for idx, eigu in enumerate(self.eigu[:neigvs]):
            print(f"inverse iteration: {idx}/{neigvs}")
            mat = X - eigu*np.eye(self.mat_cols)
            arr = utils.inverse_iteration(
                mat, initial_guess=self.eigv[:, idx], tolerance=1e-12)
            self.eigv[:, idx] = arr.reshape(self.mat_cols,)
        end = time.time()
        print(f"Inverse iterations done in {end-start} seconds")
```

A.2 Oja's method

```python
import numpy as np
import math
import time
import base

class Oja(base.BaseMethod):
    def __init__(self, learning_rate=1e-6, maxiter= 1e2, tolerance=1e-2):
        """
        learning_rate: should be lower the more data points htere are
        maxiter: maximum number of iterations, used as stop criteria
        """
        base.BaseMethod.__init__(self, maxiter=maxiter, tolerance=tolerance)
        self.learning_rate = learning_rate

    def _fit_transform(self,X):
        """
        Applies Oja's rule to find 1st principal component of data in X (rows).
        """
        #randomly initialize weights
        weights = np.random.normal(scale=0.25, size=(self.mat_cols, 1))
        if self.debug:
            print("Starting iterations for Oja rule...")
        Ysquare = np.square(X.dot(weights))
        global_start_time = time.time()
        for i in range(self.maxiter):
            start_time = time.time()
            prev_weights = weights.copy()
            Y = np.dot(X, weights)
            prev_ysquaremean = Ysquare.mean()
            Ysquare = np.square(Y)
            weights += self.learning_rate * np.sum(Y*X - Ysquare*weights.T, axis=0).reshape((self.mat_cols, 1))
            residue = np.linalg.norm(weights - prev_weights)
            if self.debug:
                print(f"{i}: {residue}")
            self._eigu.append(np.array(Ysquare.mean()).reshape(1,1))
            self._eigv.append(weights.copy())
            end_time = time.time()
            self._times.append(end_time - start_time)
            if i> 0 and np.abs(Ysquare.mean() - prev_ysquaremean) < self.tolerance:
                break
        self.ystar = np.dot(X,weights)
        global_end_time = time.time()
        self.time = global_end_time - global_start_time
        self.iterations = [i+1]
        self.tot_iterations = i + 1
        self.eigv = weights
        self.eigu = np.array(Ysquare.mean()).reshape(1,1)
```

A.3 Oja-Sanger's method

```python
import numpy as np
import math
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler
import time
from base import BaseMethod
from oja import Oja

class OjaSanger(BaseMethod):
    def __init__(self, learning_rate=1e-6, maxiter=1e5, tolerance=1e-2, compute_n_eigs=None):
        """
        learning_rate: should be lower the more data points htere are
        """
        BaseMethod.__init__(self, maxiter=maxiter, tolerance=tolerance)
        self.learning_rate = learning_rate
        self.reorder = False
        self.compute_n_eigs = compute_n_eigs

    def _fit_transform(self,X):
        """
        Applies Oja-Sanger's rule to find eigenvectors and eigenvalues of covariance matrix of X.
        """
        if self.compute_n_eigs is None:
            self.compute_n_eigs = self.mat_cols
        if self.check_result:
            print(f"Computing correct eigenvectors...")
            correct_eigv, correct_eigu = self._check(X)
        workingX = X.copy()
        eigu_list = []
        eigv_list = []
        global_start_time = time.time()
        for i in range(self.compute_n_eigs): #i represents eigenvalue index
            print(f"Computing eigenvector {i}/{self.mat_cols - 1}...")
            oja = Oja(learning_rate=self.learning_rate,maxiter=self.maxiter,tolerance=self.tolerance)
            oja.fit_transform(workingX)
            start_time = time.time()
            if i == 0:
                self.iterations.append(oja.tot_iterations)
            else:
                self.iterations.append(self.iterations[-1] + oja.tot_iterations)
            ystar = oja.ystar
            eigv = oja.eigv
            eigu = oja.eigu
            if len(eigu_list) == 0:
                prev_eigus = np.zeros(self.compute_n_eigs)
                prev_eigvs = np.zeros((self.mat_cols, self.compute_n_eigs))
            else:
                prev_eigus = eigu_list[-1].copy() #array of prev eigus approx
                prev_eigvs = eigv_list[-1].copy()
            for it in range(oja.tot_iterations):
                prev_eigus[i] = oja._eigu[it]
                eigu_list.append(prev_eigus.copy())
                prev_eigvs[:,i] = oja._eigv[it].reshape(self.mat_cols,)
                eigv_list.append(prev_eigvs.copy())
            for rowIdx, row in enumerate(workingX):
                row = row.reshape((self.mat_cols,1))
                eigvSubspace = float(ystar[rowIdx])*eigv
                x1 = row - eigvSubspace
                workingX[rowIdx] = x1.reshape((1,self.mat_cols))
            end_time = time.time()
            oja_times = oja._times
            oja_times[-1] += end_time - start_time
            self._times += oja_times
        global_end_time = time.time()
        self.tot_iterations = self.iterations[-1]
        self.time = global_end_time - global_start_time
        self.eigu = eigu_list[-1]
        self.eigv = eigv_list[-1]
        self._eigu = eigu_list
        self._eigv = eigv_list
```

A.4 Jacobi's method

```python
import time
import numpy as np
from base import BaseMethod

def get_givens_theta(matrix, p, q):
    x_pq = matrix[p, q]
    x_pp = matrix[p, p]
    x_qq = matrix[q, q]
    theta = np.arctan((2.0*x_pq)/(x_pp - x_qq))/2.0
    return theta

def givens_rotation(matrix, k: int, l: int, theta: float):
    n = matrix.shape[0]
    rotated = matrix
    ct = np.cos(theta)
    st = np.sin(theta)
    for j in range(n):
        if j != k:
            rotated[k, j] = matrix[j, k]*ct + matrix[j, l]*st
            rotated[j, k] = rotated[k, j]
        if j != l:
            rotated[l, j] = - matrix[j, k]*st + matrix[j, l]*ct
            rotated[j, l] = rotated[l, j]
    rotated[k, k] = matrix[k, k] * \
        (ct**2) + matrix[l, l]*(st**2) + 2*matrix[k, l]*st*ct
    rotated[l, l] = matrix[k, k] * \
        (st**2) + matrix[l, l]*(ct**2) - 2*matrix[k, l]*st*ct
    rotated[k, l] = 0
    rotated[l, k]= rotated[k, l]
    return rotated

class Jacobi(BaseMethod):
    def __init__(self, maxiter= 1e2, tolerance=1e-2):
        """
        Jacobi's method.
        """
        BaseMethod.__init__(self, maxiter=maxiter, tolerance =
    tolerance)

    def _fit_transform(self, X):
        """
        Applies Jacobi's method to find eigenvalues and
    eigenvectors
        """
        workingX= X.copy()
        self.pq_tuples= []
        self.theta_list= []
        global_start_time= time.time()
        for i in range(self.maxiter):
            if self.debug and i % 100 == 0:
                print(f"Iteration {i}/{self.maxiter}")
            start_time= time.time()
            max_val= -1
            max_p= -1
            max_q= -1
            for p in range(self.mat_rows):
                for q in range(p+1, self.mat_cols):
                    max_elem= abs(workingX[p, q])
                    if max_elem > max_val:
                        max_val= max_elem
                        max_p= p
                        max_q= q
            assert max_p != max_q
            if max_p > max_q:
                temp= max_p
                max_p= max_q
                max_q= temp
            assert max_p < max_q
            self.pq_tuples.append((max_p, max_q))
            theta= get_givens_theta(workingX, max_p, max_q)
            self.theta_list.append(theta)
            workingX= givens_rotation(workingX, max_p, max_q, theta
   )
            end_time= time.time()
            self._times.append(end_time - start_time)
            if self.save_data_for_all_iterations:
                self._eigu.append(np.diag(workingX).copy())
        global_end_time= time.time()
        self.time= global_end_time - global_start_time
        prevQ= np.eye(self.mat_cols)
        for idx, pqt in enumerate(self.pq_tuples):
            theta= self.theta_list[idx]
            p, q= pqt
            Q= np.eye(self.mat_cols)
            ct= np.cos(theta)
            st= np.sin(theta)
            Q[p, p]= ct
            Q[p, q]= -st
            Q[q, p]= st
            Q[q, q]= ct
            newQ = prevQ.dot(Q)
            if self.save_data_for_all_iterations:
                self._eigv.append(newQ)
            prevQ= newQ.copy()
        self.iterations= [i + 1 for k in range(self.mat_cols)]
        self.tot_iterations= i + 1
        if self.save_data_for_all_iterations:
            self.eigu= self._eigu[-1]
            self.eigv= self._eigv[-1]
        else:
            self.eigu= np.diag(workingX)
            self.eigv= prevQ
```

```

    def _fit_transform_(self, X):  # Jacobi method
        workingX= X.copy()
        def maxElem(a):  # Find largest off-diag. element a[k,l]
            n= len(a)
            aMax= 0.0
```

A.5 Power method

```python
import time
import numpy as np
from base import BaseMethod


class Power(BaseMethod):
    def __init__(self, maxiter= 1e2, tolerance=1e-2):
        """
        Power method.
        """
        BaseMethod.__init__(self, maxiter=maxiter, tolerance=
    tolerance)


    def _fit_transform(self,X):
        """
        Applies Power method to find eigenvalues and eigenvectors
        """
        global_start_time = time.time()
        x = np.random.normal(size=(self.mat_cols)).reshape((self.
    mat_cols, 1))
        ev = x.T.dot(X.dot(x))
        for i in range(self.maxiter):
            start_time = time.time()
            Xv = X.dot(x)
            x_new = Xv / np.linalg.norm(Xv)
            ev_new = x_new.T.dot(X.dot(x_new))
            self._eigu.append(np.array(ev_new).reshape(1,1))
            self._eigv.append(x_new.reshape(self.mat_cols,1))
            end_time = time.time()
            self._times.append(end_time - start_time)
            if np.abs(ev - ev_new) < self.tolerance:
                break
            x = x_new
            ev = ev_new
        global_end_time = time.time()
        self.time = global_end_time - global_start_time
        self.iterations = [i + 1]
        self.tot_iterations = i + 1
        self.eigu = self._eigu[-1]
        self.eigv = self._eigv[-1]
```

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 13/07/2021