1999

Neural computation of all eigenpairs of a matrix with real eigenvalues

Serafim Theodore Perlepes

Follow this and additional works at: https://scholarworks.lib.csusb.edu/etd-project

Part of the Computer Sciences Commons

Recommended Citation

https://scholarworks.lib.csusb.edu/etd-project/1525
NEURAL COMPUTATION OF ALL EIGENPAIRS OF A MATRIX WITH REAL EIGENVALUES

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Serafim Theodore Perlepes
March 1999
NEURAL COMPUTATION OF ALL EIGENPAIRS OF A MATRIX WITH REAL EIGENVALUES

A Thesis
Presented to the
Faculty of
California State University,
San Bernardino

by
Serafim Theodore Perlepes

March 1999

Approved by:

Dr. George M. Georgiou, Chair
Computer Science

Dr. Owen Murphy

Dr. Kerstin Voigt
ABSTRACT

In this thesis, new artificial neural network methods that compute all eigenpairs of a matrix with real eigenvalues are introduced and evaluated. The basic learning rule presented is used to find eigenpairs associated with both positive and negative eigenvalues. The above rule is extended to finding all eigenpairs employing as much parallelism as possible. The algorithms presented are: Serial Deflation, Serial-pipelined deflation and Parallel-pipeline. The three algorithms extract all eigenpairs in order, and Parallel pipeline performs better than the other two. It computes results faster and has the highest degree of parallelism.
ACKNOWLEDGEMENTS

I would like to take the opportunity to acknowledge the direct and indirect help of many people who made this thesis possible. First, I wish to express my appreciation and gratitude to my advisor, Dr. George M. Georgiou whose constant encouragement and support were always present. His expertise and guidance reinforced my knowledge and made me a better researcher. Appreciation is also expressed to the rest of the examining committee members Dr. Owen Murphy, and Dr. Kerstin Voigt for their support and guidance. It was a pleasure and a privilege to collaborate with these special people who contributed so much to the successful completion of this thesis.

I am also grateful to many others for help in one form or another during the course of this work. My sincere and deepest appreciation goes to my father Theodore, my mother Zaffie, and my brother Tom for their love and support throughout the past few years. Also, I am thankful to the rest of my family in Greece for their enduring support and understanding during the course of this research and my program of study.
# TABLE OF CONTENTS

ABSTRACT ........................................ iii

ACKNOWLEDGEMENTS ............................... iv

LIST OF TABLES ............................... vii

LIST OF FIGURES ......................... viii

LIST OF GRAPHS ............................. ix

CHAPTER ONE Introduction ................. 1
  Computing eigenpairs: background ........ 2
  Using neural networks to compute eigenpairs ... 7
  Review of previous work .................. 11
  Thesis preview ............................ 16

CHAPTER TWO The new learning rules and algorithms 18
  The modified learning rule .............. 19
    Derivation ................................ 20
  Finding all eigenpairs .................. 22
    Serial deflation ...................... 23
    Serial-pipelined deflation ........... 24
  Parallel-pipelining rule .............. 26
    Derivation of parallel-pipelining rule ... 27
    Relating parallel-pipelining and Sanger's rules ... 30

CHAPTER THREE Implementation ........... 31
  Finding extreme eigenvalues and eigenvectors ... 32
LIST OF TABLES

Table 1. The serial-pipelined deflation algorithm . . . 25
Table 2. Results A . . . . . . . . . . . . . . . . . . . 66
Table 3. Results B . . . . . . . . . . . . . . . . . . . 67
Table 4. Explanation of symbols for table 5 . . . . . 74
Table 5. Early results . . . . . . . . . . . . . . . . . 75
Table 6. Convergence data . . . . . . . . . . . . . . . 76
LIST OF FIGURES

Figure 1. Simple feedforward neural network . . . . . . . 8
Figure 2. Architecture for simple hebbian learning . . . 12
Figure 3. Simplified hardware implementation of serial-
  pipelined deflation . . . . . . . . . . . . . . . . . . . . . 37
Figure 4. Simplified hardware implementation of parallel-
  pipelined method . . . . . . . . . . . . . . . . . . . . . . 39
LIST OF GRAPHS

Graph 1. The square of the norm of $\mathbf{x}$ vs. epochs ........ 41
Graph 2. The square of the norm of $\mathbf{x}$ vs. epochs ........ 42
Graph 3. The square of the norm of $\mathbf{x}$ vs. epochs ........ 43
Graph 4. The square of the norm of $\mathbf{x}$ vs. epochs ........ 44
Graph 5. Distance between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs .... 46
Graph 6. The square of the norm of $\mathbf{x}$ vs. epochs ........ 47
Graph 7. Distance between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs .... 48
Graph 8. The square of the norm of $\mathbf{x}$ vs. epochs ........ 49
Graph 9. The square of the norm of $\mathbf{x}$ vs. epochs ........ 50
Graph 10. The square of the norm of $\mathbf{x}$ vs. epochs ......... 51
Graph 11. The square of the norm of $\mathbf{x}$ vs. epochs ......... 52
Graph 12. The $\cos(\theta)$ vs. epochs ............................ 53
Graph 13. The $\cos(\theta)$ vs. epochs ............................ 54
Graph 14. The $\cos(\theta)$ vs. epochs ............................ 55
Graph 15. Distances between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs .... 56
Graph 16. Distances between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs .... 57
Graph 17. Distances between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs .... 58
Graph 18. The square of the norm of $\mathbf{x}$ vs. epochs ......... 59
Graph 19. The square of the norm of $\mathbf{x}$ vs. epochs ......... 61
Graph 20. The square of the norm of $\mathbf{x}$ vs. epochs ......... 62
Graph 21. The $\cos(\theta)$ vs. epochs ............................ 63
Graph 22. The $\cos(\theta)$ vs. epochs .......................... 64
Graph 23. The $\cos(\theta)$ vs. epochs .......................... 65
CHAPTER ONE Introduction

Computing the eigenvalues and associated eigenvectors of a given real matrix is necessary in many scientific disciplines. This computation is important for scientific and engineering problems such as signal processing, control theory, and geophysics [21]. The general solutions of differential equation systems often require knowledge of the spectral quantities, i.e. the eigenvectors and eigenvalues. Also, the meaning of the covariance matrix in statistics is most clear when the eigenpairs are known. Besides the standard methods for computing eigenvalues and their eigenvectors, there is a great interest in computing eigenpairs using neural techniques [9]-[10], [17]-[21].

The word eigenvalue derives from the German word eigenwert; eigen means peculiar, characteristic and wert means value. An eigenvalue is one of those special values of a parameter in a particular equation for which the equation has a solution. Specifically, the nontrivial solutions of the equation $A\mathbf{x} = \lambda \mathbf{x}$ were introduced by Lagrange in 1762 to solve systems of differential equations with constant coefficients. The nonzero solutions are the eigenvalues, and the term was introduced by Hilbert in 1904.
to denote a property of integral equations. Later on, eigenvalues became attached to matrices [11]. In the case of a differential equation, a single-valued, finite, and continuous solution is found only for particular values of a parameter and these are the proper-values or eigenvalues of the differential equation. Detailed mathematical definitions are given in section 1.1.

1.1 Computing eigenpairs: background

Finding the eigenvalues of a square matrix is a difficult problem that arises in a wide variety of scientific applications. The solution of many physical problems requires the calculation, or at least estimation of the eigenvalues and corresponding eigenvectors of a matrix associated with a linear system of equations. A few definitions are necessary to better understand the problem.

**Definition 1** A nonzero vector $x \in \mathbb{R}^n$ is an eigenvector (or characteristic vector) of a square matrix $A \in \mathbb{R}^{n \times n}$ if there exists a scalar $\lambda$ such that $Ax = \lambda x$. Then $\lambda$ is an eigenvalue (or characteristic value) of $A$ [1].

In other words, a number $\lambda$ is an eigenvalue of the $n \times n$ matrix $A$ if and only if the homogeneous system
\[(A - \lambda I)x = 0\]

has nontrivial solutions. Furthermore, the nontrivial solutions of the above equation are the eigenvectors of \(A\) associated with eigenvalue \(\lambda\). So, in order to compute the eigenvalues and eigenvectors of a given \(n \times n\) matrix \(A\), we must solve the system \(Ax - \lambda x = 0\). The matrix form of this equation is in Definition 1.

**Definition 2** If \(A\) is a real \(n \times n\) matrix, the polynomial defined by

\[p(\lambda) = \det(A - \lambda I)\]

is called the characteristic polynomial of \(A\) [8].

**Definition 3** If \(A\) is a real \(n \times n\) matrix, the equation defined by

\[\det(\lambda I - A) = 0\]

is called the characteristic equation of \(A\) [3].

It is known that \(p\) is an nth-degree polynomial with real coefficients and, consequently, has at most \(n\) distinct roots; some of these roots may be complex [8].
Definition 4 An eigenvalue $\lambda_i$ and the associated non-zero eigenvector $v_i = [v_{i1}, v_{i2}, ..., v_{in}]^T$ are referred to as an eigenpair.

Definition 5 The magnitude of a vector $v = [v_1, v_2, ..., v_n]$ is $\|v\| = \sqrt{v \cdot v} = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$. It is also called the norm or length of a vector, where $\cdot$ denotes the inner product operator.

Definition 6 The distance between vectors $u$ and $v$ is defined to be

$$d(u, v) = \|u - v\| = \sqrt{(u_1 - v_1)^2 + (u_2 - v_2)^2 + ... + (u_n - v_n)^2}$$

The distance will be used as an error measure between the computed eigenvector and the ideal eigenvector.

Definition 7 The largest in magnitude eigenvalue of a matrix $A$ is called the dominant eigenvalue [8].

Definition 8 For two vectors $x$ and $y$, the cosine of the angle between them is defined as

$$\cos(\theta) = \frac{x \cdot y}{\|x\| \|y\|} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i^2}}.$$
If \( \cos(\theta) \) is close to 1, then \( \mathbf{x} \) and \( \mathbf{y} \) are close to having the same direction. If \( \cos(\theta) \) is close to -1, then \( \mathbf{x} \) is approximately \( -\mathbf{y} \).

**Example 1** Find the eigenvalues and eigenvectors of

\[
\mathbf{A} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}.
\]

Solution.

\[
\mathbf{A} - \lambda \mathbf{I} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} - \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} = \begin{bmatrix} 3 - \lambda & 1 \\ 1 & 3 - \lambda \end{bmatrix}
\]

and \( \det(\mathbf{A} - \lambda \mathbf{I}) = (3-\lambda)(3-\lambda) - 1 \). Setting \( \det(\mathbf{A} - \lambda \mathbf{I}) = 0 \) and solving for \( \lambda \) gives \( \lambda = 4 \) and \( \lambda = 2 \). To find the eigenvalue for \( \lambda = 4 \) we must find a nonzero solution to

\[
\begin{align*}
(3 - 4)x + y &= 0 \\
x + (3 - 4)y &= 0
\end{align*}
\]

This system just demands that \( y = x \). So an eigenvector for the eigenvalue 4 is the vector \( [1 \ 1]^T \) or any nonzero multiple of it.

Similarly, to find an eigenvector for \( \lambda = 2 \) we solve
\[ x + y = 0 \]
\[ x + y = 0 \]

This gives the relation \( y = -x \) which in turn shows that \([1 \ -1]^T\) is an eigenvector for \( \lambda = 2 \).

We can summarize our findings by writing that \( A = CDC^{-1} \)

\[
D = \begin{bmatrix} 4 & 0 \\ 0 & 2 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}
\]

where the diagonal entries of \( D \) are the eigenvalues of \( A \), and the column vectors of \( C \) are their corresponding eigenvectors. This example was taken from [8].

The three types of matrices are mentioned or used in this thesis are symmetric, positive-definite, and positive semidefinite.

**Definition 9:** A square matrix is said to be symmetric if its elements are symmetric about the diagonal. That is to say \( A_{ij} = A_{ji} \) for all \( i \) and \( j \).

**Definition 10:** A matrix \( A \) is positive definite if \( (Av) \cdot v > 0 \) for all vectors \( v \neq 0 \). All Eigenvalues of a positive definite matrix are positive.

**Definition 11:** A matrix \( A \) is positive semidefinite if
\[(\mathbf{A}v) \cdot \mathbf{v} \geq 0\]

for all \( \mathbf{v} \neq 0 \).

The eigenvalues from these three kinds of matrices are real numbers.

1.2 Using neural networks to compute eigenpairs

Artificial neural networks (ANN) are a growing part of the study of artificial intelligence and are intended to be a link to true biological machines [16]. In order to build intelligent machines, the naturally occurring model is the human brain. For that purpose, one of the first things that comes to mind is simulating the function of the brain directly on a computer. Computers today have remarkable abilities including the ability to store vast quantities of information and perform extensive arithmetic calculations without error. Their circuits operate very fast, and humans cannot approach such capabilities [16]. On the other hand, computers cannot efficiently perform simple everyday tasks like walking, talking, natural language processing, and common sense reasoning. Current artificial intelligence systems cannot do any of these tasks better than humans.
The need for a processor that has the functionality of the human brain and the speed of a computer attracted and still attracts many researchers to ANNs [19]. An artificial neural network is a machine or algorithm modeled after the design and function of the brain. For the most part, neural network architectures are not meant to duplicate the operation of the human brain, but to receive inspiration from known facts about how the brain works [16].

**Figure 1.** Simple feedforward neural network
In general, a network consists of many simple processors, also known as nodes or neurons, that are linked together in layers. There are input and output layers, each containing any number of nodes. As illustrated in Figure 1, there can be a number of hidden layers separating the input from the output, also containing an arbitrary number of nodes. Each node contains some small amount of data and each link between the nodes has a value (weight) associated with it, as shown in Figure 1. The concept of the biological machine stems from the idea that the input nodes are equivalent to neurons, and the links are equivalent to the synapses plus axons.

The network is trained in a way that the weights are modified until the ANN, for a given input, produces the correct or most correct output. This training can be done using either supervised or unsupervised learning. An ANN undergoes supervised learning when the input vectors and the corresponding output vectors are used. In a way, there is a teacher to guide the network to the correct output.

Learning in supervised networks is often times achieved by a method called back propagation. The difference between the desired and actual network outputs is observed, then the network is modified, and the
procedure repeats until correct results are obtained. So, the neural network minimizes an error function of the output. Unfortunately, back propagation has problems. First it is slow, secondly it is difficult to analyze the actions of the hidden layers, and finally results are not always produced due to weaknesses of the gradient descent method, (i.e. local minima can distract from gradient descent) [10].

In unsupervised learning there is no teacher; rather, the neural network incorporates local information and internal rules to associate the different inputs with the different outputs. This makes it more similar to the workings of the brain, which does not have an internal teacher. Unsupervised learning is best suited for situations where there is a great deal of redundancy in the input. By repetition, the network organizes itself to distinguish patterns or features in the data [20].

It is interesting to research and study how parallel structures, like neural networks, can solve problems like the computation of eigenvalues and their corresponding eigenvectors. According to many researchers, neural computing defined by dynamic systems is a very promising
approach for solving real time computational problems [9]-
[21].

1.3 Review of previous work

In unsupervised learning, a neural network must
discover for itself patterns, regularities, features,
correlations, or categories of the input data and code for
them in the output [10]. While discovering these, the
network changes its parameters, a process called self-
organization [10].

Assuming we have an input vector with components $\xi_i$, and
each component has a weight $w_i$ associated with it. If we
consider the simplest case of a single linear unit, its
scalar output $V$ is

$$V = \sum_i w_i \xi_i = w^T \xi = \xi^T w$$  \hspace{1cm} (1)

where $w$ is the weight vector. The network architecture is
shown in Figure 2. Hebbian learning, a fundamental learning
mechanism [10], is represented by this learning rule:

$$\Delta w_i = \eta V \xi_i$$ \hspace{1cm} (2)
where $\eta$ is the learning rate, a small positive constant.

The product $V\xi_i$ is the standard Hebb rule and is present one form or another in many learning rules, including the one presented in this thesis (section 2.1).

**Figure 2.** Architecture for simple hebbian learning

The problem here is that the weights keep on growing without bound and learning never stops [10]. To avoid this, Oja [13] added weight decay proportional to the square of the output $V^2$ to the plain Hebbian rule

$$\Delta w_i = \eta V(\xi_i - Vw_i). \quad (3)$$

Oja's rule above causes its weight vector to converge to the eigenvector that corresponds to the largest eigenvalue $\lambda_{\text{max}}$ of matrix $C$, the covariance matrix of the data set [13].
Several researchers have extended Oja's rule to multineuron networks that extract all eigenvectors of the covariance matrix $C$ of a given input set of vectors [10],[18]-[19]. Sanger's rule [18], for example, projects the outputs of an input vector $\xi$ onto the space of the first $M$ principal components. The updated rule is

$$
\Delta w_{ij} = \eta V_i \left( \xi_j - \sum_{k=1}^{j} V_k w_{kj} \right)
$$

This rule is most often used in applications since it is robust and also extracts the principal components individually in order [10].

Georgiou and Tsai approached the problem of finding the eigenvectors of a symmetric positive definite matrix (with neural networks) in a novel way [9]. Data having approximately a specific covariance matrix (the given matrix) is randomly generated, and then the APEX [12] neural architecture and algorithm is used to extract the eigenvectors [20].

In the above studies, learning rules are applied to the covariance matrix of the data (input) vectors, and the eigenvalues and eigenvectors extracted are those of the
covariance matrix. In this thesis, the direct problem is investigated: given a matrix $A$, find all eigenvalues and associated eigenvectors of $A$.

In [17], a dynamical method that produces estimates of real eigenvectors and eigenvalues was presented. The technique proposed is applied to estimate eigenspectra of real $n$-dimensional $k$-forms. Their approach was based on a spectral splicing property of the line manifolds often found in solutions of polynomial differential equations [17].

In [21], a dynamical system for computing the eigenvectors associated with the $\lambda_{\text{max}}$ of a positive definite matrix $A$ is described. They used the rule:

$$\frac{dx}{dt} = Ax - f(x)x \quad (5)$$

where $x = (x_1, x_2, \ldots, x_n)^T \in \mathbb{R}^n$ and function $f(x)$ satisfies certain assumptions [21]. As it is mentioned in the same paper, the first term on the right-hand side in equation 5 can be considered as the standard Hebb rule term (equation 1), and the second term acts to bound the length of vector $x$ [21].
Also, in [21] is mentioned that researchers have looked at the cases where \( f(x) = x^\top Ax \) and \( f(x) = x^\top x \), using positive definite matrices as input.

Samardizija and Waterland in [17] propose sign reversal to obtain negative eigenvalues: use

\[
\frac{dx}{dt} = Ax - (x^\top x)x \quad \text{for positive eigenvalues and}
\]

\[
\frac{dx}{dt} = -Ax - (x^\top x)x \quad \text{for negative.}
\]

In this thesis, we find negative eigenvalues and their associated eigenvectors without sign reversal.

**Statement of the problem:** Use a new neural network algorithm to compute all eigenpairs of a symmetric matrix (i.e., with real eigenvalues).

1. Introduce a new learning rule to find eigenpairs associated with both positive and negative eigenvalues.

2. Introduce algorithms that extend the new rule above to be able to find all eigenpairs employing as much parallelism as possible. The algorithms to be explored are:

   a. Serial Deflation

   b. Serial-pipelined deflation
c. Parallel-pipeline

1.4 Thesis preview

Chapter Two of the thesis presents the theory of the new rules and the new neural algorithms that solve the eigenvalue-eigenvector problem given a matrix $A$. The mathematical foundations, theorems, and proofs are presented and discussed.

To be more specific, equation (5) is used in [21] to compute the eigenvector corresponding to the largest eigenvalue of a positive definite matrix $A$, i.e. all eigenvalues of $A$ are positive. In this thesis, equation (5) is modified to compute eigenpairs of a real symmetric matrix. The only limitation now is that matrix $A$ should have real eigenvalues. Also, besides computing the eigenvector corresponding to the largest eigenvalue, the modified rule can extract the eigenvector associated with the smallest negative eigenvalue of $A$. Depending on the initial value of eigenvector $x$, convergence can be directed to find the eigenpair that belongs to the largest positive or smallest negative eigenvalue. In addition, a serial deflation technique is used to extract the remaining eigenpairs [4],[6]. A serial-pipelined deflation algorithm
is introduced to extract all eigenpairs in parallel-like fashion. Lastly, a third, even more efficient algorithm (Parallel-pipeline) is used to extract all eigenpairs in parallel fashion.

In Chapter Three the specifics of the implementation method and the software simulation aspects are presented. In Chapter Four the computer simulation results are presented and discussed. In Chapter Five conclusions are drawn, and in Chapter Six future studies possibilities are outlined.
CHAPTER TWO The new learning rules and algorithms

This chapter contains the new learning rules and algorithms of this thesis. The proposed learning rule and its derivation are presented in section 2.1. In the derivation, Lagrange multipliers are used [2]. This method is suitable for solving optimization problems like the one in section 2.1.

Next, in section 2.2, the three new methods (Serial Deflation, Serial-Pipelined deflation, and Parallel-Pipeline) for extracting all eigenpairs and their derivations are presented and discussed. The deflation theorem from numerical analysis in 2.2.1 was taken from [6].

In the Parallel-pipelined pipeline section (2.2.3), we extend $\Delta \mathbf{x} = \eta (\mathbf{A} \mathbf{x} - (\mathbf{x}^T \mathbf{A} \mathbf{x}) \mathbf{x})$ to a rule that extracts all eigenpairs. Sanger [18] extended Oja's rule (equation 4) to extract all eigenpairs of the covariance matrix (which is always positive semi-definite) of the given data vectors, whereas we extend $\Delta \mathbf{x} = \eta (\mathbf{A} \mathbf{x} - (\mathbf{x}^T \mathbf{A} \mathbf{x}) \mathbf{x})$ to compute all eigenpairs of a symmetric matrix.

Sanger's rule (equation 5) uses the Gram-Schmidt orthogonalization procedure (well known in linear algebra
to expand Oja's rule. It is important in that it uses only local computations, a characteristic that makes it attractive for neural networks applications. Also, it computes all eigenvectors at the same time: during each iteration a correction to the eigenvectors is made until all converge to their true values.

Sanger's rule although related to the deflation technique [6] of finding successive eigenvectors in that each eigenvector depends on the previous one, it differs in that computation is not done in the serial manner of deflation, but in a more parallel one.

### 2.1 The modified learning rule

A square matrix $A$ is the input to the new learning rule. The only restriction on $A$ for this rule is that $A$ is a square matrix with real eigenvalues. The new rule computes the largest positive or the smallest negative eigenvalue and associated eigenvector according to the initial value of the product $x^T Ax$.

Let $A \in \mathbb{R}^{n \times n}$ be a square matrix with real eigenvalues. The scalars $\lambda_{\text{min neg}}$ and $\lambda_{\text{max pos}}$ denote the smallest negative and the largest positive eigenvalue of $A$, respectively (if such
values exist). In the case that $A$ does not have any negative eigenvalues then $\lambda_{\text{min neg}}$ does not exist since it is defined as the smallest negative eigenvalue. Conversely, when $A$ has only negative eigenvalues, $\lambda_{\text{max pos}}$ does not exist.

Define the product

$$\kappa_i = x_i^T A x_i$$

(6)

and the learning rule

$$x_{i+1} = x_i + (\eta \kappa_i) (A x_i - \kappa_i x_i)$$

(7)

that can also be written as

$$\Delta x = \eta (x^T A x) (A x - (x^T A x)x)$$

(7b)

where $\eta$ is the learning rate (in this case, $\eta$ is a small positive real number) and $x = (x_1, \ldots, x_n)^T \in \mathbb{R}^n$. As the square of the magnitude of $x$ ($\|x\|^2$) converges to 1, $\kappa$ converges to eigenvalue $\lambda_{\text{max pos}}$ of $A$ if $\kappa_0$ is positive or to $\lambda_{\text{min neg}}$ if $\kappa_0$ is negative. At the same time, $x$ converges to the eigenvector associated with the eigenvalue that $\kappa$ converges to (either $\lambda_{\text{max pos}}$ or $\lambda_{\text{min neg}}$).

2.1.1 Derivation

Let $A \in \mathbb{R}^{n \times n}$ be a square matrix with real eigenvalues, e.g. $A$ can be symmetric. Then the field of values of $A$ is
the set \( \{ x^T A x : x \in \mathbb{R}^n, \|x\| = 1 \} \) which is an interval on the real line whose endpoints are eigenvalues. The endpoint furthest from the origin maximizes the expression \( (x^T A x)^2 \) under the constraint \( \|x\| = 1 \). Hence we can obtain an extreme eigenvalue by solving the constraint optimization problem

\[
\max (x^T A x)^2, \ x^T x = 1.
\]

We can solve such optimization problems using the Lagrange multiplier method. Let \( \lambda \) be a Lagrange multiplier. Then the problem is equivalent to maximizing \( E(x) \):

\[
E(x) = \frac{1}{2} (x^T A x)^2 - \lambda (x^T x - 1)
\]

We can use gradient descent to minimize \(-E(x)\). The gradient of \( E(x) \) with respect to \( x \) is

\[
\nabla_x E = -2(x^T A x)Ax + 2\lambda x.
\]

At equilibrium, \( \nabla_x E = 0 \), so

\[
-(x^T A x)Ax + \lambda x = 0.
\]

Right multiplying by \( x^T \),

\[
-(x^T A x)(x^T A x) + \lambda x^T x = 0
\]

or

\[
\lambda = (x^T A x)^2
\]

hence, we write
\[ \nabla_x E = -2x^T Ax (Ax - \lambda x) \]

The gradient above can be written in dynamical system form as:
\[ \nabla_x x = x^T Ax (Ax - (x^T Ax) x) \]
or as the learning rule:
\[ \Delta x = \eta (x^T Ax) (Ax - (x^T Ax) x) . \]

2.2 Finding all eigenpairs

An \( n \times n \) matrix \( A \) has precisely \( n \), not necessarily distinct, eigenvalues that are roots of the polynomial \( p(\lambda) = \det(A - \lambda I) \). In theory the eigenvalues are obtained by finding the \( n \) roots of the characteristic polynomial \( p(\lambda) \). After this, the associated linear system must be solved to find the corresponding eigenvectors. In practice, finding eigenpairs is not that simple. The characteristic polynomial is difficult to obtain, and finding the roots of an \( n \)-th degree polynomial can be difficult unless we deal with small values of \( n \). This leads to the necessity of constructing approximation techniques and algorithms to find eigenvalues and the associated with them eigenvectors. Many such matrix algebra iterative methods exist. One of the approximation
techniques that will be used here is the deflation technique.

2.2.1 Serial deflation

In general, deflation techniques involve forming a new matrix $B$ from the original matrix $A$ whose eigenvalues are the same as those of $A$ with the exception that the dominant eigenvalue of $A$ is replaced by the eigenvalue 0 in matrix $B$.

Deflation theorem from numerical analysis: Suppose that $\lambda_1, \lambda_2, \ldots, \lambda_n$ are eigenvalues of $A$ with associated eigenvectors $v_1, v_2, \ldots, v_n$, and that $\lambda_1$ has multiplicity one. If $x$ is any vector with the property that $x^T v_1$, then

$$B = A - \lambda_1 v_1 x^T$$  \hspace{1cm} (12)

is the matrix with eigenvalues 0, $\lambda_2, \lambda_3, \ldots, \lambda_n$ and associated eigenvectors $v_1, w_2, w_3, \ldots, w_n$ where $v_1$ and $w_1$ are related by the equation

$$v_1 = (\lambda - \lambda_1) w_1 + \lambda_1 (x^T w_1) v_1$$  \hspace{1cm} (13)
for each $i = 2, 3, \ldots, n$.

The idea is to first find $\lambda_1$ and its associated eigenvector $v_1$ using the learning rule of equation 7. Then, deflate matrix $A$ using equation 12 store the result back to $A$, and iterate the rule again to find the $\lambda_2 - v_2$ eigenpair and continue like that until we extract all eigenpairs. If the matrix has negative eigenvalues ($\lambda_{\text{minneg}}$ exists), we can also work backwards starting from $\lambda_n = \lambda_{\text{minneg}}$ and by deflating $A$ and iterating the rule extract the eigenpairs in reverse order, from $\lambda_n$ to $\lambda_1$.

### 2.2.2 Serial-pipelined deflation

Next step of this research is to cast the serial deflation process as a neural network. To do that, we need to construct an algorithm that extracts all eigenpairs in parallel fashion.
Table 1. The Serial-pipelined deflation algorithm

Declare
\[ A_0, A_1, \ldots, A_n : n \times n \text{ matrices} \]
\[ x_0, x_1, \ldots, x_n : n \text{ size vectors randomly initialized} \]
\[ \eta_0, \eta_1, \ldots, \eta_n : \text{real learning rates} \]

While not all have converge

Begin
\[
\Delta x_{A_0} = \eta_0 (x_{A_0}^T A_0 x_{A_0}) (A_0 x_{A_0} - (x_{A_0}^T A_0 x_{A_0}) x_{A_0})
\]
\[ A_1 = A_0 - (x_{A_0}^T A_0 x_{A_0}) x_{A_0} x_{A_0}^T \]
\[
\Delta x_{A_1} = \eta_1 (x_{A_1}^T A_1 x_{A_1}) (A_1 x_{A_1} - (x_{A_1}^T A_1 x_{A_1}) x_{A_1})
\]
\[ A_2 = A_1 - (x_{A_1}^T A_1 x_{A_1}) x_{A_1} x_{A_1}^T \]
\[
\Delta x_{A_{n-1}} = \eta_{n-1} (x_{A_{n-1}}^T A_{n-1} x_{A_{n-1}}) (A_{n-1} x_{A_{n-1}} - (x_{A_{n-1}}^T A_{n-1} x_{A_{n-1}}) x_{A_{n-1}})
\]
\[ A_n = A_{n-1} - (x_{A_{n-1}}^T A_{n-1} x_{A_{n-1}}) x_{A_{n-1}} x_{A_{n-1}}^T \]
\[
\Delta x_{A_n} = \eta_n (x_{A_n}^T A_n x_{A_n}) (A_n x_{A_n} - (x_{A_n}^T A_n x_{A_n}) x_{A_n})
\]
End

To introduce parallelism to serial deflation, instead of deflating matrix \( A \) when one of the eigenpairs has been completely computed, we deflate by a small quantity after each iteration. We need to iterate as many learning rules as the number of eigenpairs (n) that we are extracting.
For each iteration: after a rule has been updated, "partial" deflation takes place. Table 1 contains the algorithm needed to implement serial-pipelined deflation in pseudo code. According to the size of the matrix used, the corresponding number of learning rules is used to extract the eigenvalues and eigenvectors.

2.2.3 Parallel-pipeline rule

In this section we propose a new rule that extends the basic rule:

$$\Delta \mathbf{x} = \eta (\mathbf{A}\mathbf{x} - (\mathbf{x}^\mathsf{T}\mathbf{A}\mathbf{x})\mathbf{x})$$  \hspace{1cm} (13),

that is used for extraction of only one, the dominant, eigenpair.

The new rule is:

$$\Delta \mathbf{x}_i = \eta (\mathbf{A}\mathbf{x}_i - \sum_{k=0}^{i} (\mathbf{x}_i^\mathsf{T}\mathbf{A}\mathbf{x}_k)\mathbf{x}_k)$$ \hspace{1cm} (14)

where $\eta > 0$ is the learning rate (a small positive constant), $\mathbf{A}$ is a given $n \times n$ positive semi-definite matrix, and $\mathbf{x}_i$, $1 \leq i \leq n$, are the eigenvectors, as column vectors, ordered by decreasing importance. Notice that for $i = 1$ the new rule collapses to the basic rule of equation (13).
The parallel pipeline rule uses only local computations, a characteristic that makes it attractive for neural networks applications. Another characteristic is that computes all eigenvectors at roughly the same time. A correction to the eigenvectors is made at each iteration until all converge to their true values.

Still each eigenvector depends on the previous one, but now the computation is done in a way more parallel than serial-pipelined deflation.

2.2.3.1 Derivation of parallel-pipeline rule

The new rule is derived using Lagrange multipliers and mathematical induction. This rule and the idea to use Lagrange multipliers in the derivation are due to Dr. Georgiou. Supposed that the first \( i-1 \) (most dominant) eigenvectors of \( A \) have been obtained and are normalized: \( \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{i-1} \). The problem now is to find the next normalized eigenvector \( \mathbf{x}_i, i \leq n \). We cast the problem as an optimization one and solve it with the method of Lagrange multipliers. The objective function we would like to maximize is

\[
\kappa_i = \mathbf{x}_i^T \mathbf{A} \mathbf{x}_i
\]  

(15)
and the constraints are:

\[ x^T_i x_i = 1 \quad (16) \]

\[ x^T_j x_k = 0, \ 1 \leq k \leq i \quad (17) \]

Equation (16) ensures that \( x \) is normalized and equation (17) that \( x \) is orthogonal to all previous eigenvectors.

Using Lagrange multipliers \( \lambda_k, 1 \leq k \leq i \) for the constraints, we form a new function that we would like to maximize:

\[
G(x_i, \lambda_1, \lambda_2, \ldots, \lambda_i) = x_i^T A x_i + 2 \sum_{k=1}^{i-1} \lambda_k x_i^T x_k + \lambda_i (x_i^T x_i - 1) \quad (18)
\]

The gradient of function \( G \) with respect to all variables must equal zero at the extremum:

\[
\nabla_{x_i} G = A x_i + \sum_{k=1}^{i-1} \lambda_k x_k + \lambda_i x_i = 0 \quad (19)
\]

Left multiplying Equation (19) with \( x_i^T \), and using constrains (16) and (17), we obtain

\[
\lambda_i = -x_i^T A x_i \quad (20)
\]

Left multiplying Equation (19) successively by \( x_i^T, x_2^T, \ldots, x_{i-1}^T \), and again using constraints (16), (17) the following results:

\[
\lambda_i = -x_i^T A x_i, \ 1 \leq k \leq i \quad (21)
\]
Substituting the λ's back to equation (19), the gradient now becomes:

\[
\nabla_{x_i} G = A x_i + \sum_{k=1}^{i-1} (x_k^T A x_i) x_k - (x_i^T A x_i) x_i,
\]

(22)

Which can be written more compactly as

\[
\nabla_{x_i} G = A x_i + \sum_{k=1}^{i} (x_k^T A x_i) x_k.
\]

(23)

Thus, using gradient ascent, we write the new learning rule:

\[
\Delta x_i = \eta (A x_i - \sum_{k=1}^{i} (x_i^T A x_k) x_k),
\]

(24)

which the same as Equation (14).

We note that for \( i = 1 \) equation (24) reduces to the basic rule of equation (13), which will converge to the most significant eigenvector, and thus the mathematical induction is complete.

Since less significant eigenvectors depend on the more significant ones, it is expected that the more significant ones will converge faster. In practice we noticed that the more significant ones converge almost at the same time for square symmetric matrices of dimension three and four and faster for higher dimension matrices.
2.2.3.2 Relating parallel-pipeline and Sanger's rules

The new rule is analogous the one proposed by Sanger: Sanger's rule works with data vectors, whereas the new rule works with a given symmetric matrix.

By applying the expectation operator on Sanger's rule its relationship to the new rule is illustrated:

\[
\langle \Delta x_{i,j} \rangle = \sum_p x_{i,p} A_{p,j} - \sum_{k=1}^q \left( \sum_{pq} x_{i,q} A_{p,q} x_{i,q} \right) x_{j,j}
\]

or

\[
\langle \Delta x_i \rangle = \eta (A x_i - \sum_{k=1}^q (x_{i,k} A x_k) x_k)
\]

It can be seen from the above equation, the right hand side is identical to Equation (24). Although this is not a rigorous argument, since one left hand side has the expectation operator and the other does not, still the similarity of the two equations is striking, and the two rules can be considered analogous. Sanger's rule can be used for finding the eigenvectors of the covariance matrix of given data vectors and the new rule for finding the eigenvectors of a given symmetric matrix.
CHAPTER THREE Implementation

Testing the proposed learning rule under different conditions was very important during the first stages of the research. The simulation programs use a C++ class library developed by Laurent Deniau in CERN, Switzerland. It was downloaded from http://wwwinfo.cern.ch/~ldeniau/, and the library was build with g++ compiler version 2.7.2 under the UNIX (System V Release 4.0) operating system. The matrix class of the library offers the member function eig() which is used to calculate the eigenpairs of symmetric matrices. This function was used in the program to compare the computed results with ideal ones. The Maple mathematical package was used to compare results also. To generate the graphs associated with the simulation results, gnuplot was used. It should be noted that implementations of neural algorithms do have many free variables that usually are randomly initialized. When I (via email) asked Dr. Terry Sanger why a particular implementation of Sanger's rule did not converge, he replied "if it's not converging, the usual problem is a rate that is too high ... try using the rule with just 1 output eigenvector, find the fastest rate that gives good convergence."
Depending on the variables used, initial conditions should be adjusted so the algorithm used produces results. In that fashion, the learning rate that performs best for the three algorithms used in this thesis was chosen. The matrices of which the eigenpairs should be computed are random symmetric to avoid cases of matrices with complex eigenvalues.

3.1 Finding extreme eigenvalues and eigenvectors

To find the extreme eigenvalues $\lambda_{\text{min}^{neg}}$ and $\lambda_{\text{max}^{pos}}$, the initial value for $\kappa_0$ is checked and when the desired for our computation $\kappa_0$ is obtained (section 2.1), the learning rule is applied to the matrix. Since the matrix is constant, what makes $\kappa_0 = x_0^T A x_0$ positive or negative is the initialization value of vector $x_0$. If we want to find $\lambda_{\text{max}^{pos}}$ then $\kappa_0 = x_0^T A x_0$ should be positive. The value of $\kappa_0 = x_0^T A x_0$ must be negative if we want the rule to converge to $\lambda_{\text{min}^{neg}}$. If $\lambda_{\text{min}^{neg}}$ does not exist, then the rule automatically finds $\lambda_{\text{max}^{pos}}$. Also, if $\lambda_{\text{max}^{pos}}$ does not exist then we find $\lambda_{\text{min}^{neg}}$ instead.
The implementation has three steps. First the
declaration of all needed variables and constants (vectors,
matrices, learning rate), second the initialization of the
variables, and third the iteration of the learning rule
until convergence is achieved, i.e., until it converges to
vector $\mathbf{x}$ with the square of its length $\|\mathbf{x}\|^2 \approx 1$. For step
two the random number generator that comes with C language
was used to initialize $\mathbf{A}$ and $\mathbf{x}_0$. The learning rate was set
to 0.01. After a certain number of iterations, the
learning rate is divided by a constant; the default is 500
iterations, and that way the learning rate becomes smaller
and smaller but not less than 0.001. This technique is
often used in Neural Networks to make similar rules
converge faster [10]. When iteration of the rules starts,
division on predefined intervals gradually decreases the
relatively large learning rate (0.01), i.e., the rate is
divided by 1.01 after every 500 iterations. The rate
should not be decreased too much because learning is slowed
down proportionally to the decrease of the learning rate.
For that reason, the smallest rate used is very close to
0.001.
When trying to find $\lambda_{\text{maxpos}}$, $x_0$ must be initialized to a value that makes $\kappa_0$ positive. Function getposinitval() was implemented for that reason. On the other hand, $x_0$ has to be initialized to a value that results to a negative $\kappa_0$ for convergence to $\lambda_{\text{minneg}}$. Function getneginitval() was implemented to do that. Both functions initialize $x_0$ with random values and then compute $\kappa_0$. If the result is the desired one, the $x_0$ is returned. Otherwise, $\kappa_0$ is computed again by trying a new random initialization of $x_0$. If there is no $x_0$ that makes $\kappa_0$ positive then $A$ does not have a positive eigenvalue. Likewise, if there is no $x_0$ that makes $\kappa_0$ negative then $A$ does not have a negative eigenvalue. Accordingly, both functions have a limit to how many times they initialize $x_0$. If the appropriate value has not been found after a hundred iterations, then the current value of $x_0$ is returned.

3.2 Serial deflation implementation

Again, to obtain a value for $x_0$ that will converge to either $\lambda_{\text{maxpos}}$ or $\lambda_{\text{minneg}}$, function getposinitval() or getneginitval() must be used during initialization. As
mentioned earlier, if $\kappa_0$ is negative then the learning rule converges to the eigenvector associated with the smallest negative eigenvalue, whereas if $\kappa_0$ is positive it finds $\lambda_{\text{maxpos}}$. It should be noted at this point that $\lambda_{\text{maxpos}}$ and $\lambda_{\text{minneg}}$ do not have to be a dominant eigenvalue to make serial deflation work.

The actual eigenpairs are calculated using the included with the C++ library member function eig() of the matrix class. Because this function works only with symmetric matrices, Maple was used to compute the ideal eigenpairs in some early experiments (Appendix A).

The segment of the program that implements serial deflation extracts the $n$ eigenpairs of an $n \times n$ matrix serially and in order.

We can either start from $\lambda_{\text{maxpos}}$ and continue deflating $A$ and iterating equation (7) until we find $\lambda_{\text{min}}$ and its associated eigenvector, or we can start from $\lambda_{\text{minneg}}$ and continue until all eigenpairs are found (in reverse order). If no $\lambda_{\text{maxpos}}$ exists then we find $\lambda_{\text{minneg}}$ first and vice-versa.
3.3 Serial-pipelined deflation implementation

Since the eigenpairs in this case are computed after each iteration, we have to initialize all eigenvalue and eigenvector variables before the iterations of the rules start. For example, if we choose $A$ to be a $4 \times 4$ matrix, four eigenpairs should be extracted. For each eigenpair to be computed, iterating rule (7) is used. The four rules will be iterated, each is depending on the previous one, until all converge. Thus, all $k_i$s ($k_i = x_i^T A x_i$) must be initialized before we start iterating.

As it was mentioned earlier, if the eigenvalue of an eigenpair to be computed is positive, the initial value for that eigenvalue ($k_i$) before we start iterating should also be positive. Conversely, when the eigenvalue of the eigenpair to be extracted is negative, its starting value should also be negative. If a random symmetric matrix is used, it is impossible to know beforehand how many eigenvalues will be negative and how many will be positive, in order to initialize them accordingly. To overcome this initialization problem, matrices with positive eigenvalues are used for the serial-pipelined deflation algorithm. For
the rest of the implementation, the serial-pipelined deflation algorithm of table 1 (section 2.2.2) is used.

The pipeline nature of the algorithm is illustrated in figure 3. At each stage, we deflate the matrix and pass it to the next stage. For example, in the second pipeline stage matrix $A_1$ is needed, so we deflate $A_0$ using $x_0$ (equation 12 in 2.2.1) and then we iterate the learning rule.

Figure 3. Simplified hardware implementation of serial-pipelined deflation
3.4 Parallel-pipeline implementation

For the same reason as with the serial-pipelined deflation algorithm, $x_i$s are initialized to values that make $K_i$s positive, i.e. the symmetric matrices used have positive eigenvalues.

For an $n \times n$ size matrix, $n$ learning rules are used, to compute $n$ eigenpairs. The rules are iterated until they all converge. Equation (14) on section 2.2.3 is used to compute each eigenvector. The first rule extracts the largest eigenvector, the second computes the second largest, and so on. Thus, the eigenpairs are extracted in parallel and in order.

Figure 4 illustrates what we get if we view the parallel-pipeline algorithm as a pipeline. Matrix $A$ is the same for all stages since no deflation takes place. Each stage is an iterating rule. So, all preceding vectors ($x_1$ to $x_{i-1}$) are needed to update the rule that computes $x_i$. For example, in the third stage we need $x_1$ and $x_2$ to iterate the rule associated with $x_3$. 
Figure 4. Simplified hardware implementation of parallel-pipelined method
CHAPTER FOUR Computer simulation results and discussion

Symmetric matrices of different dimensions were used as input to the simulation programs. When testing the proposed rule and the three different algorithms, eigenpairs from $2 \times 2$ to $10 \times 10$ size symmetric matrices were successfully computed. For the results presented in this chapter, symmetric and symmetric positive-definite matrices of size $3 \times 3$ and $4 \times 4$ were used. To calculate the actual eigenvalues and eigenvectors, the build-in to the C++ library member function eig() is used since all matrices are symmetric (function eig() works only with symmetric matrices using the Jacobi algorithm to find eigenpairs). The computed eigenpairs are almost equal to the actual eigenpairs (calculated by function eig()) within a tolerance of 0.0000001. There exist cases where the eigenvector with opposite sign is computed. This is acceptable since a vector with opposite sign is simply an eigenvector in the opposite direction.

4.1 Sample runs

Graph 1 shows how the rule converges for matrix
\[
A = \begin{bmatrix}
-8.4 & 1.4 & -1.8 \\
1.4 & -6 & -4.8 \\
-1.8 & -4.8 & 5
\end{bmatrix}
\]

**Graph 1.** The square of the norm of $x$ vs. epochs

For this particular run, the computed $\lambda_{\text{maxpos}}$ was 7.10624 with associated eigenvector $x^T = [-0.139299, -0.353633, 0.924954]$. The value of $\kappa_0$ was positive, so the learning rule converged to $\lambda_{\text{maxpos}}$. The number of iterations needed for convergence was 3616.

The computed $x$ was equal to the actual ($x_a$) returned from function `eig()` within a tolerance of 0.0000001.
Graph 2 depicts how the rule converged for the same matrix $A$ but with different initial $x$.

The learning rate is the same for both runs. The only parameter that changed was the initial $x_0$. The result of this was to need 37,289 iterations to converge, almost ten times more than the number required during the first run. Also, $x_0 = -x_a$ which is the eigenvector with opposite direction.
Graphs 3 and 4 show how the rule converged when finding $\lambda_{\text{min}}$ and its associated eigenvector first.

**Graph 3.** The square of the norm of $x$ vs. epochs

The same $A$ and learning rate were used for graphs 3 and 4. As before, the number of iterations required for the rule to converge is different. This indicates that the rule is sensitive to initial conditions even if the only variable that changes in this case is the initial value of $x$. 

43
4.2 Comparing results

When the $\|x\|^2$ converges to 1, that does not necessarily imply that $x$ converged to an eigenvector. The Euclidean distance of two vectors is a measure of how close they are in space. The distance between the computed $x$ and the actual (or ideal) $x_a$ provides a good measure of the quality of the result.
As mentioned earlier, the learning rule sometimes converges to $\mathbf{x}_a$ and other times to $-\mathbf{x}_a$. In the first case the distance goes to zero, and in the second case it goes to 2 since
\[
d(\mathbf{x}_a - \mathbf{x}) = \\
= \|\mathbf{x} \cdot (-\mathbf{x})\| = \\
= \sqrt{(x_1 - (-x_1))^2 + (x_2 - (-x_2))^2 + \ldots + (x_n - (-x_n))^2} = \\
= \sqrt{2x_1^2 + 2x_2^2 + \ldots + 2x_n^2} = \\
= \sqrt{4} \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} = \\
= \sqrt{4} \|\mathbf{x}\| = \\
= 2
\]

Another way to evaluate results is to look at the cosine of the angle between the computed eigenvector and the actual. The value of $\cos(\theta)$ is used as a measure of how close the two vectors are.

For matrix $\mathbf{A} = \begin{bmatrix} 4.2 & -0.4 & 8.6 \\ -0.4 & 2.2 & -9.4 \\ 8.6 & -9.4 & 5.4 \end{bmatrix}$, the learning rule converged to $\lambda_{\text{max pos}} = 16.8988$ and its associated eigenvector $\mathbf{x}^\top = [0.486797 \ 0.461649 \ -0.741555]$. The actual eigenvector in this case is

$\mathbf{x}_a^\top = [-0.486797 \ -0.461649 \ 0.741555] = -\mathbf{x}^\top$.

Thus, the distance converges to 2. Graph 5 demonstrates exactly that.
Graph 5. Distance between $x$ and $x_0$ vs. epochs

On the other hand, graph 6 shows how the square of the norm of $x$ converges to 1.
For the same $A$ as above when the program found the extreme negative eigenvalue and associated eigenvector. The results were, $\lambda_{\text{min neg}} = -8.37147,$

$$x^T = [0.444881, -0.585649, -0.677566]$$

where the corresponding actual eigenpair was $\lambda = -8.37147$ and

$$x_a^T = [0.444881, -0.585649, -0.677566] = x^T$$
Since we have sign agreement between the actual and computed eigenvectors, this time the distance converged to zero. Graph 7 demonstrates exactly that.

Graph 7. Distance between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs

Graph 8 again shows how the $||\mathbf{x}||^2$ converges to 1, in the same experiment as above.
4.3 Simulation runs of the three algorithms (3 x 3 matrix)

The next run provides a representative collection of graphs that shows how the three algorithms perform. The symmetric matrix used for the all algorithms was

$$\mathbf{A} = \begin{bmatrix} 3.351098 & 0.288294 & 0.746157 \\ 0.288294 & 3.037798 & 0.356264 \\ 0.746157 & 0.356264 & 4.911105 \end{bmatrix},$$

and graphs 9, 10, 11 show how the squares of the norms of the eigenvectors
converged for the Parallel-pipeline, Serial-pipelined deflation, and Serial Deflation algorithms, respectively.

Graph 9 shows how the square of the norms of the three rules converged, when the Parallel-pipeline algorithm was used.

Graph 9. The square of the norm of $x$ vs. epochs

As it can be seen from the graph the rule associated with the largest eigenvalue converged first (curve N 0), the rule computing the eigenpair of second the second largest eigenvalue converged second (curve N 1), the rule
associated with the smallest eigenvalue converged third (curve N 2).

**Graph 10.** The square of the norm of \( \mathbf{x} \) vs. epochs

Graph 10 shows the convergence of \( ||\mathbf{x}||^2 \) of the calculated eigenvectors when the Serial-pipelined deflation algorithm was used.

We can readily see that in this case serial-pipelined deflation was 5,000 iterations slower than Parallel-pipeline. Also, the rules associated with the largest and
second largest eigenvalues (curves N 0 and N 1) converged during the first one thousand iterations, but it took another 6,000 iterations for the rule associated with the smallest eigenvalue (curve N 2) to converge.

The next graph corresponds to the same matrix with serial deflation calculating the eigenpairs.

**Graph 11.** The square of the norm of $\mathbf{x}$ vs. epochs

This algorithm is serial, so first it extracts the dominant eigenpair and deflates the matrix. Then the deflated matrix is used to get the second dominant
eigenpair, and the matrix is deflated again to extract the last eigenpair. The number of iterations for this method for this particular run approximately was 12,000, i.e., 4,000 more than serial-pipelined deflation and 9,000 more than parallel-pipeline.

The next 3 graphs show the cosine of angle theta between the ideal and computed eigenvectors converges to 1 or -1.

Graph 12. The $\cos(\theta)$ vs. epochs
If \( \cos(\theta) \) approaches 1, \( \mathbf{x} \) has the same sign as the ideal eigenvector; on the other hand, when \( \cos(\theta) \) converges to -1 then the sign of the computed \( \mathbf{x} \) is opposite to the sign of the ideal.

As it is shown from the graph 12 (Parallel Pipeline algorithm), the cosine associated with the largest eigenvalue converged to -1. The cosines of the other two rules converged to 1. Also, since we have convergence of the cosine to 1 or -1 that implies that the computed eigenvectors are correct.

**Graph 13.** \( \cos(\theta) \) vs. epochs

![Graph 13](image)
Graph 13, shows how cosine theta for the 3 rules converged to 1 when serial-pipelined deflation was used. Again, the cosines for the first and second rule (Cos 0, Cos 1) converged much earlier than the number of iterations the last rule needed to produce results.

Graph 14. \( \cos(\theta) \) vs. epochs

Graph 14 is displays how the three cosines converged when the Serial Deflation algorithm was used. It is interesting to note that in this case the cosine of the second rule (Cos 1) was the one that required the most iterations to
converge. This happens because the proposed rule that is used the Serial Deflation algorithm is very sensitive to initial conditions. The next three graphs show the calculation of the distance between computed and ideal eigenvectors for the three algorithms tested.

Graph 15 shows how the distance converged to 0 or 2 depending on which eigenvector is calculated. In the same order as before, graph 15 shows the distance between $x$ and $x_a$ when parallel-pipeline was used.

**Graph 15.** Distances between $x$ and $x_a$ vs. epochs
It is interesting in this case to note to that all rules start to converge to 0 or 2 roughly the same time. The same was witnessed in most runs with the Parallel-pipeline rule. On the other hand, in Parallel and Serial Deflation the first two rules converge faster, and they have to "wait" for the last one to converge. Graph 16 illustrates just that. Serial-pipelined deflation was used, and rules one and two (D 0 and D 1, respectively) converged much sooner than rule 3 (D 2).

**Graph 16.** Distances between \( x \) and \( x_a \) vs. epochs
In the next graph, 17, Serial Deflation is used and again one of the rules (the second one, D 1) took longer than the other 2 rules. Overall, this algorithm takes the biggest number of iterations.

**Graph 17.** Distances between $\mathbf{x}$ and $\mathbf{x}_a$ vs. epochs

The distance calculation for the three learning rules when serial deflation was used converged to 0 or 2 in the same way the $\cos(\theta)$ converged to 1 and -1.
4.4 Simulation runs of the three algorithms (4 × 4 matrix)

The algorithms perform the same way for higher dimension matrices, but it takes longer to produce results. There exist cases where the eigenvalues closer to zero take more iterations to converge because the learning rate favors the convergence of the larger eigenvalues.

The next example run uses a 4 × 4 matrix and as before six graphs are used to demonstrate how the three algorithms carried out the computation this time.

Graph 18. The square of the norm of \( x \) vs. epochs
The symmetric matrix used was

\[
A = \begin{bmatrix}
3.23268 & -0.293662 & -0.411963 & -0.480726 \\
-0.293662 & 2.4143 & -0.0757437 & -0.380533 \\
-0.411963 & -0.0757437 & 4.56274 & 1.59223 \\
-0.480726 & -0.380533 & 1.59223 & 3.29027
\end{bmatrix}
\]

Besides using the same matrix for all three algorithms, the same initial \( \mathbf{x}_0 = [0.005 -0.002 -0.039 0.011] \) was used for all also.

Starting from the Parallel-pipeline algorithm, graph 18 presents how \( \|x\|^2 \) (one for each of the four rules) converges to 1. The graph shows that the eigenvector associated with the largest eigenvalue (line N 0) took less number of iterations to converge, and then the eigenvector of the second largest eigenvalue, and so on.

Graph 19 shows \( \|x\|^2 \) of the four eigenvectors when the Serial-pipelined deflation algorithm was used with the same \( \mathbf{A} \) and \( \mathbf{x}_0 \). The four learning rules start to converge approximately at the same time at about 1700 iterations. The squared norm of \( \mathbf{x} \) for the rule extracting the smallest eigenvalue and associated eigenvector (curve N 3) remained below 0.4 for almost 5500 iterations (out of 6500), and then started to converge faster.
Serial-pipelined deflation in graph 19 produced results similar to Parallel-pipeline, but with almost twice as many iterations needed for convergence.

Graph 20 draws the norms of the computed eigenvectors when serial deflation was used. We can easily see the four different serial computations taking place.
Similar to the result we got when the $3 \times 3$ matrix was used with serial deflation, one of the rules (in this case the last) took longer to compute its corresponding eigenvector. The third rule (line N 2) took close to 5,000 iterations to produce results whereas the last took almost 40,000 iterations.

For the same three runs, now we take a look at how the computation of the eigenvectors progresses when observing the cosine theta between the calculated eigenvector and the
ideal one. Graph 21 is the cosine calculation for parallel-pipeline.

Graph 21. The $\cos(\theta)$ vs. epochs

It is noted that all rules converge at almost the same time, three out of the four converged to -1 or one approximately after two thousand iterations (lines Cos 0, Cos 1, Cos 2).

Graph 22 shows how serial-pipelined deflation behaves.
Again, the computation takes a little longer, but still it performs better than the serial deflation algorithm $\cos(\theta)$ computation that follows (Graph 23). As expected, serial deflation took longer (more than five times longer), approximately 7000 iterations for serial-pipelined deflation compared to the 40000 iterations of serial deflation in graph 23.
Group results in the next section portray the characteristics or the three algorithms using a sample of 250 different matrices.

**Graph 23.** The $\cos(\theta)$ vs. epochs

4.5 Simulation results using 250 different matrices

To better understand how the three algorithms behave, 250 different random symmetric positive definite matrices (dimensions $3 \times 3$ and $4 \times 4$) were used, and table 2 summarizes the results:
Rows and columns, horizontally and vertically add up to our sample size, i.e. 250. Each entry shows how many times the corresponding algorithm converged: first (first row), second (second row), or third (third row). "First" means the algorithm needed the least number of iterations for convergence (section 3.1), "second" is used for the second smaller and third for the algorithm that takes the most iterations to converge and produce results. For example after a certain run, serial deflation requires 2000 iterations to produce results when for the same run Serial-pipelined deflation takes 1000 and Parallel-pipeline requires 500 to produce results. In this case, we say that Parallel-pipeline is first for this particular run, Serial-pipelined deflation second, and Serial Deflation third.

The column number indicates which algorithm was used. The first column is for serial deflation (SD), the second for serial-pipelined deflation (SPD) and the third for the
Parallel-pipeline algorithm (PP). Matrix R1 below displays the results on matrix instead of tabular format

\[
R1 = \begin{bmatrix}
0 & 52 & 198 \\
16 & 185 & 49 \\
234 & 13 & 3
\end{bmatrix}.
\]

As we can see from above, Parallel-pipeline (PP) came first (took the least iterations to converge) 198 out of 250 times whereas serial deflation never came first, as expected.

If each number in R1 is translated to a percentage then we obtain a doubly stochastic matrix [5]

\[
R2 = \begin{bmatrix}
0 & 20.8 & 79.2 \\
6.4 & 74 & 19.6 \\
93.6 & 5.2 & 1.2
\end{bmatrix}
\]

and table 3 below:

<table>
<thead>
<tr>
<th></th>
<th>SD</th>
<th>SPD</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>0</td>
<td>20.8</td>
<td>79.2</td>
</tr>
<tr>
<td>2nd</td>
<td>6.4</td>
<td>74</td>
<td>19.6</td>
</tr>
<tr>
<td>3rd</td>
<td>93.6</td>
<td>5.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Serial Deflation gets its highest percentage on the third place, i.e. it came last (took the most iterations to converge) 93.6% of the times. Serial-pipelined deflation receives its highest percentage (74%) in second place, and
Parallel-pipeline gets its highest percentage (79.2%) in first place. We also note that for the 250 matrices used in this experiment, serial deflation never came first.
CHAPTER FIVE Conclusions

The original $\frac{dx}{dt} = Ax - f(x)x$ was extended to a new one $\Delta x = \eta(x^\top Ax)(Ax - (x^\top Ax)x)$ that finds eigenpairs associated with both positive and negative eigenvalues.

As mentioned in chapter three, the learning rate was originally set to 0.01 and division on predefined intervals gradually decreased it to a number not lower than 0.001. The exit condition was that we iterate the rule until the square of the length of the extracted eigenvector $x$ converges to one ($\|x\|^2 \approx 1$). The computer simulation showed that the new rule computed the desired eigenpairs.

The original rule was extended to find all eigenpairs. The first algorithm explored was a linear, serial deflation algorithm. Using the same learning rate and exit condition as above, the simulations showed that the algorithm successfully extracted all. The algorithm was equally successful in first computing the smallest negative eigenvalue and associated eigenvector or in computing first the largest positive eigenvalue and associated eigenvector.

The first attempt to introduce parallelism via extension of serial deflation was successful. A new
serial-pipelined deflation algorithm was introduced to extract all eigenpairs. With serial deflation, in order to extract an eigenpair we needed the previous one. Serial-pipelined deflation deflates the matrix and calculates partial results after each iteration of the rules. The simulation results showed an improvement over serial deflation. With serial-pipelined deflation the rules converged much faster, due to the pipelined nature of the algorithm (Figure 3 shows the hardware implementation).

Even though this algorithm converged faster, it was still taking more time to extract the smaller eigenvalues and associated eigenvectors, than the time needed to extract the eigenpairs associated with the larger eigenvalues.

A new Parallel-Pipelined rule was derived using gradient descent and the Lagrange multipliers method. That was the final attempt to achieve a higher level of parallelism, and as the results show this method was the best of the three presented in this thesis. Figure 4 shows a simplified figure of the Parallel-Pipelined method.

As we conclude from all results and especially from table 3 of the previous section, Parallel-pipeline rule performs the best. The reason for that is its pipeline
structure and the way each term is updated. Just as pipelining is the key technique used to make faster CPUs [15], pipelining the iterating rules of the Parallel-pipeline algorithm speeded up the computation considerably. In this case, partial results for a rule extracting a specific eigenpair are computed by using partial results of all previous eigenpairs. Another advantage of the Parallel-Pipelined method is that the eigenvectors converged almost at the same time. In other words, during each iteration a correction to the eigenvectors is made until all converge to their true values. In this case, we did not witness what happened with the serial-deflation algorithm, i.e. the last eigenvector requiring a larger number of iterations to converge witch slowed down the whole process.
CHAPTER SIX Future work

The derivation in Section 2.1 states that matrix $A$ must be symmetric for the proposed learning rule (equation 7) to work. Some early experiments show cases where the rule worked even when no restrictions were imposed to $A$. For Table 5 in Appendix A, ten random $4 \times 4$ matrices were used for $A$, and $x_0$ was also random. In other words, there were no restrictions to the value of $\kappa_0$. If $\kappa_0$ was negative but $A$ did not have any negative eigenvalues then the rule diverged. Also, if $\lambda_{\text{minneg}}$ or $\lambda_{\text{maxpos}}$ were complex, the rule also diverged. To avoid infinite loops, a limit to the number of iterations was imposed. If after that number of iterations, we still do not have convergence of the square of the norm of $x$ to 1, within 0.000001, then the program initialized the variables again to values that produce $\kappa_0$ with an opposite to the initial sign. After initializing, iteration of the rule started again. When convergence was not achieved, the extreme eigenvalues of $A$ were complex. Table 5 in Appendix 1 contains some runs that computed an extreme eigenvalue of the given $A$ successfully. Again, the number of iterations needed for convergence varied in each case. The problem here as mentioned above is the unpredictability
of the existence of complex eigenvalues for matrix \( \mathbf{A} \) because no restrictions are imposed when initializing \( \mathbf{A} \). The actual eigenpairs in this case are computed using the Maple mathematical package.

Since there exist cases where the basic rule worked even if the matrix was not symmetric, maybe there exists another class of matrices that we can apply the rules and algorithms presented here to compute eigenpairs. Researchers in future studies should look into how Parallel-pipeline can be expanded to work with different kinds of matrices, and in the complex domain.

Also, researchers in the future should look how that can overcome the initialization problem. When this is solved, matrices with negative eigenvalues can be used as input for Serial-pipelined deflation and Parallel-pipeline.
APPENDIX A The First experiments

Table 4. Explanation of symbols for table 5

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>The matrix of which we try to compute particular eigenpairs using equation (7)</td>
</tr>
<tr>
<td>x₀</td>
<td>The random initial value for vector x</td>
</tr>
<tr>
<td>K₀₀</td>
<td>The initial sign of K₀</td>
</tr>
<tr>
<td>k</td>
<td>The eigenvalue of A that was computed by the iteration of equation (7)</td>
</tr>
<tr>
<td>λ</td>
<td>The corresponding to k actual eigenvalue of A</td>
</tr>
<tr>
<td>x</td>
<td>The computed by our dynamic system eigenvector of A corresponding to eigenvalue k</td>
</tr>
<tr>
<td>xₐ</td>
<td>The corresponding to x actual eigenvector of A</td>
</tr>
<tr>
<td>i</td>
<td>The number of time the rule was iterated in order to converge</td>
</tr>
</tbody>
</table>

λ: computed with the build-in functions of the matrix library or with the Maple mathematical software package
## Table 5. Early results

<table>
<thead>
<tr>
<th>A</th>
<th>x₀</th>
<th>x₀₈</th>
<th>x₈</th>
<th>²</th>
<th>x₉</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8 -8</td>
<td>2 4</td>
<td>-0.037</td>
<td>+ 9.92293</td>
<td>9.92293</td>
<td>-.92164</td>
<td>-.918775</td>
</tr>
<tr>
<td>1</td>
<td>-4 -7 2</td>
<td>0.026</td>
<td>-0.09030</td>
<td>0.12474</td>
<td>-.90905</td>
<td>0.213986</td>
</tr>
<tr>
<td>1</td>
<td>-5 -9 9</td>
<td>0.01</td>
<td>-0.35344</td>
<td>0.21890</td>
<td>-.912583</td>
<td>1.120435</td>
</tr>
<tr>
<td>-3.6</td>
<td>-8 3 7 2.7</td>
<td>-0.014</td>
<td>-0.836281</td>
<td>0.31685</td>
<td>-.818576</td>
<td>1.120435</td>
</tr>
<tr>
<td>8.2</td>
<td>-3.6 7.4 9.8</td>
<td>-0.023</td>
<td>-8.24791</td>
<td>8.24790</td>
<td>0.02951</td>
<td>-.028339</td>
</tr>
<tr>
<td>9</td>
<td>-9.8 5.2 7.6</td>
<td>-0.042</td>
<td>-0.336815</td>
<td>0.32964</td>
<td>-.42205</td>
<td>0.149168</td>
</tr>
<tr>
<td>9.6</td>
<td>2 1.2 9.4</td>
<td>-0.027</td>
<td>-0.428017</td>
<td>0.449168</td>
<td>-.353539</td>
<td>0.353539</td>
</tr>
<tr>
<td>7</td>
<td>-3 -7 11</td>
<td>-0.025</td>
<td>-14.8039</td>
<td>14.8039</td>
<td>0.02895</td>
<td>1.116297</td>
</tr>
<tr>
<td>-3 8</td>
<td>-1.4 7.9</td>
<td>-0.00903</td>
<td>-0.009095</td>
<td>-0.00912</td>
<td>-.912583</td>
<td>1.120435</td>
</tr>
<tr>
<td>4 3.2</td>
<td>4.11 7.11</td>
<td>0.006</td>
<td>0.0116297</td>
<td>0.120435</td>
<td>-.918775</td>
<td>1.120435</td>
</tr>
<tr>
<td>11 7.9</td>
<td>.711 6.03</td>
<td>0.01</td>
<td>0.83053</td>
<td>0.871971</td>
<td>-.905620</td>
<td>0.905620</td>
</tr>
<tr>
<td>9.6</td>
<td>-8.9 2.9</td>
<td>0.047</td>
<td>0.910939</td>
<td>0.914236</td>
<td>0.40037</td>
<td>0.092190</td>
</tr>
<tr>
<td>-1</td>
<td>-9.6 -6.6 -4.4</td>
<td>0.03</td>
<td>0.081901</td>
<td>0.081423</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-9.2</td>
<td>-2.4 -7.6 -2.8</td>
<td>0.045</td>
<td>-0.403091</td>
<td>0.40037</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-6</td>
<td>-8.8 1 -4.8</td>
<td>0.013</td>
<td>-0.031615</td>
<td>0.031430</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-4</td>
<td>-7.8 1 -3.6</td>
<td>-0.023</td>
<td>-0.274427</td>
<td>0.265986</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>6.8</td>
<td>8.7 6.2 2.2</td>
<td>0.042</td>
<td>-0.303267</td>
<td>-0.293839</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-4</td>
<td>1.8 -7.2 -0.4</td>
<td>-0.046</td>
<td>-0.659299</td>
<td>0.638803</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>7.6</td>
<td>-4.4 1 -6.6</td>
<td>0.013</td>
<td>-0.630907</td>
<td>0.611293</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>6</td>
<td>5.6 6.6 -1.6</td>
<td>-0.033</td>
<td>-0.437648</td>
<td>-0.433472</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-3</td>
<td>1.6 2.8 9.8</td>
<td>0.045</td>
<td>-0.093078</td>
<td>0.092190</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-3</td>
<td>5.6 9.4 4</td>
<td>-0.027</td>
<td>-0.786702</td>
<td>0.779186</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-8</td>
<td>8.6 -4 -3</td>
<td>-0.019</td>
<td>0.425324</td>
<td>0.421266</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>5.6</td>
<td>-8.4 4.4 8.2</td>
<td>-0.02</td>
<td>0.89709</td>
<td>0.892221</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-4</td>
<td>-8 9.6 4.6</td>
<td>-0.001</td>
<td>-0.249703</td>
<td>0.248351</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>3.8</td>
<td>5.8 -9.6 -9</td>
<td>-0.001</td>
<td>-0.366059</td>
<td>0.362007</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-2.8</td>
<td>3.2 4.6 2.2</td>
<td>0.006</td>
<td>-0.018952</td>
<td>0.018850</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-4.2</td>
<td>-4.6 2 -6.6</td>
<td>-0.016</td>
<td>-0.133112</td>
<td>0.130853</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-9.2</td>
<td>-9.2 3.6 5.2</td>
<td>0.045</td>
<td>-0.644442</td>
<td>0.644106</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-4.4</td>
<td>2.4 -9.6 -4</td>
<td>-0.02</td>
<td>0.216676</td>
<td>0.215540</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-4</td>
<td>1.6 4.4 1</td>
<td>0.006</td>
<td>-0.330574</td>
<td>0.329729</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-2.4</td>
<td>4.4 -8.6 -6.6</td>
<td>-0.01</td>
<td>-0.65256</td>
<td>-0.650892</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-8</td>
<td>-8 5.2 -7.8</td>
<td>-0.011</td>
<td>-0.619567</td>
<td>0.617983</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>2.4</td>
<td>-2.2 0 -9.2</td>
<td>0.012</td>
<td>-0.284645</td>
<td>0.283917</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-3</td>
<td>-7 6.01 11</td>
<td>-0.027</td>
<td>-0.962885</td>
<td>0.961164</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>-3</td>
<td>8 .03 -1.4</td>
<td>-0.032</td>
<td>-0.186521</td>
<td>-0.190061</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>2.8</td>
<td>1.4 9.1 4.11</td>
<td>-0.021</td>
<td>-0.157811</td>
<td>0.160806</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
<tr>
<td>1.3</td>
<td>5.3 5 6</td>
<td>0.036</td>
<td>0.114711</td>
<td>0.116888</td>
<td>0.00037</td>
<td>0.00037</td>
</tr>
</tbody>
</table>

75
APPENDIX B Convergence data for 250 matrices

The first iteration column for each algorithm shows the number of iterations serial deflation took to converge, the second the number serial-pipelined deflation required, and the third the number that parallel-pipelined took. The rank column demonstrates the same as above, but according to the number of iterations a number is assigned. So, for the method that takes the most iterations a "3" is assigned, the one that takes the least is assigned a "1", and the middle one is assigned a "2".

Table 6. Convergence data

<table>
<thead>
<tr>
<th>parallel-pipeline, serial-pipelined deflation, serial deflation</th>
<th>iterations</th>
<th>rank</th>
<th>iterations</th>
<th>rank</th>
<th>iterations</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>27178 2300 866</td>
<td>3 2 1</td>
<td>16155 1310 1763</td>
<td>3 1 2</td>
<td>3416 2511 1840</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>30869 3048 929</td>
<td>3 2 1</td>
<td>29248 1320 1363</td>
<td>3 1 2</td>
<td>3815 2846 1982</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>15646 3654 944</td>
<td>3 2 1</td>
<td>8097 1428 1716</td>
<td>3 2 1</td>
<td>4059 2552 4213</td>
<td>2 1 3</td>
<td></td>
</tr>
<tr>
<td>13834 1564 951</td>
<td>3 2 1</td>
<td>18132 1429 7912</td>
<td>3 1 2</td>
<td>4068 1996 1610</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>50708 2350 1096</td>
<td>3 2 1</td>
<td>8579 1470 1886</td>
<td>3 1 2</td>
<td>4068 1996 1610</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>37949 2222 1226</td>
<td>1 2 1</td>
<td>17895 1521 1577</td>
<td>3 1 2</td>
<td>4550 3781 1259</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>48231 3960 1028</td>
<td>3 2 1</td>
<td>21191 1533 1495</td>
<td>3 2 1</td>
<td>4821 3960 1028</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>10838 6899 1060</td>
<td>3 2 1</td>
<td>19394 1564 951</td>
<td>3 1 2</td>
<td>4866 4394 316</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>7333 4351 1073</td>
<td>3 2 1</td>
<td>23340 1567 1601</td>
<td>3 2 1</td>
<td>5362 1830 1752</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>11335 1001 1084</td>
<td>3 2 1</td>
<td>17856 1598 1555</td>
<td>3 2 1</td>
<td>5516 2265 1421</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>15986 1736 1127</td>
<td>3 2 1</td>
<td>13045 1655 1476</td>
<td>3 2 1</td>
<td>5875 3862 2138</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>8528 2338 1134</td>
<td>3 2 1</td>
<td>2213 1635 5481</td>
<td>3 2 1</td>
<td>5963 2501 2083</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>8352 2412 1145</td>
<td>3 2 1</td>
<td>14422 1702 1699</td>
<td>3 2 1</td>
<td>5990 3163 1793</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>8352 2413 1145</td>
<td>3 2 1</td>
<td>15986 1736 1127</td>
<td>3 2 1</td>
<td>6165 5169 5377</td>
<td>3 1 2</td>
<td></td>
</tr>
<tr>
<td>17564 9658 1146</td>
<td>3 2 1</td>
<td>37450 1767 1942</td>
<td>3 2 1</td>
<td>6334 5294 4808</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>16265 1857 1196</td>
<td>3 2 1</td>
<td>16442 1772 1573</td>
<td>3 2 1</td>
<td>6417 1993 1341</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>6572 1947 1240</td>
<td>3 2 1</td>
<td>30547 1823 1607</td>
<td>3 2 1</td>
<td>6547 2681 3199</td>
<td>3 1 2</td>
<td></td>
</tr>
<tr>
<td>4550 3781 1259</td>
<td>3 2 1</td>
<td>5362 1830 1752</td>
<td>3 2 1</td>
<td>6572 1967 1240</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>22415 3001 1282</td>
<td>3 2 1</td>
<td>9664 1845 1544</td>
<td>3 2 1</td>
<td>6888 2871 2458</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>6417 1993 1341</td>
<td>3 2 1</td>
<td>16265 1857 1196</td>
<td>3 2 1</td>
<td>6767 5447 1745</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>52397 3276 1346</td>
<td>3 2 1</td>
<td>8366 1894 1512</td>
<td>3 2 1</td>
<td>6949 5694 2664</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>27412 3828 1346</td>
<td>3 2 1</td>
<td>11978 1933 1380</td>
<td>3 2 1</td>
<td>7209 3016 2533</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>33490 8587 1257</td>
<td>3 2 1</td>
<td>33650 1947 2454</td>
<td>3 2 1</td>
<td>7210 3117 3514</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>7538 2775 1358</td>
<td>3 2 1</td>
<td>6572 1947 1240</td>
<td>3 2 1</td>
<td>7278 3252 1699</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>29248 1320 1363</td>
<td>3 2 1</td>
<td>11316 1950 2812</td>
<td>3 2 1</td>
<td>7296 6035 1493</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>13978 1933 1380</td>
<td>3 2 1</td>
<td>20228 1991 11678</td>
<td>3 1 2</td>
<td>7333 4351 1073</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>17598 3153 1382</td>
<td>3 2 1</td>
<td>6417 1993 1341</td>
<td>3 2 1</td>
<td>7505 3250 1461</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>10782 3789 1397</td>
<td>3 2 1</td>
<td>4068 1996 1619</td>
<td>3 2 1</td>
<td>7598 2775 1358</td>
<td>3 2 1</td>
<td></td>
</tr>
<tr>
<td>33998 9525 1393</td>
<td>3 2 1</td>
<td>4068 1996 1619</td>
<td>3 2 1</td>
<td>7561 2724 2216</td>
<td>3 2 1</td>
<td></td>
</tr>
</tbody>
</table>
REFERENCES


