A DIVIDE-AND-CONQUER APPROACH TO THE SINGULAR VALUE DECOMPOSITION

By

Jane Elizabeth Bailey Tougas

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF COMPUTER SCIENCE WITH HONOURS
AT
DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA
APRIL 2004

DALHOUSIE UNIVERSITY
FACULTY OF
COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Computer Science for acceptance a thesis entitled "**A Divide-and-Conquer Approach to the Singular Value Decomposition**" by **Jane Elizabeth Bailey Tougas** in partial fulfillment of the requirements for the degree of **Bachelor of Computer Science with Honours**.

Dated: April 2004

Supervisor: _____
Raymond J. Spiteri

Readers: _____

_____

DALHOUSIE UNIVERSITY

Date: **April 2004**

Author: **Jane Elizabeth Bailey Tougas**

Title: **A Divide-and-Conquer Approach to the Singular Value Decomposition**

Faculty: **Computer Science**

Degree: **BCSc** Convocation: **May** Year: **2004**

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions.

Signature of Author

Contents

List of Figures

Acknowledgements

Just as it takes the support of a whole village to raise a child, it takes the support of many people to write a thesis. I extend my heartfelt appreciation to my family, who have been unflagging in their extraordinary patience, support, and encouragement. I also offer special thanks to my supervisor, Dr. Raymond J. Spiteri. His dedication, enthusiasm, and attention to detail are unparalleled, as is his knack of saying just the right thing, at just the right time. Thanks also to Dr. Milios for suggesting a path I hadn't considered and supporting my first steps upon it, and to Dr. Keast for sparking my interest in Numerical Linear Algebra. I would also like to express my appreciation to Dr. Shepherd for the use of his computers, to Sarah Cormier both for the use of her Golub-Kahan SVD implementation and for showing me it could be done, and to Geoff Johnston and Henry Stern for patiently answering my seemingly endless questions. This thesis could not have been completed without the funding of the Natural Sciences and Engineering Research Council of Canada (NSERC) for whose support I am also very grateful.

"I must finish what I've started, even if, inevitably,
what I finish turns out not to be what I began..." —Salman Rushdie

Abstract

The tremendous expansion of the Internet has made efficient searching and information retrieval an area of growing importance. Traditional search methods have drawbacks that may result in documents being returned that are not relevant or in documents that are relevant being overlooked. Latent Semantic Indexing (LSI) is an alternative method of information retrieval that attempts to avoid these problems. This method involves creating a term-by-document matrix to represent the documents and their keywords. Queries are projected into this matrix using the matrix factorization method known as the singular value decomposition (SVD). The SVD provides a means of data compression, allowing reduction of the term space while preserving the most important characteristics of the original matrix. Because calculating the SVD of a matrix is an expensive process that needs to be repeated often in LSI, speeding up this SVD calculation is of particular importance.

Calculating the SVD of a matrix involves two main steps: first the matrix is bidiagonalized, and then the SVD is formed from this bidiagonalized matrix. Traditionally, the SVD calculation is performed using the Golub-Kahan algorithm. Unfortunately, for a bidiagonal $m \times n$ matrix, this algorithm takes $\mathcal{O}(n^3)$ time [8], making it infeasible for the extremely large term-by-document matrices used in LSI. This thesis presents a divide-and-conquer approach to calculating the SVD that is asymptotically faster

than the Golub-Kahan method, resulting in a significant improvement in the cost of computation. Results are presented comparing the CPU times of the two methods, both implemented and tested in the *Matlab* problem solving environment.

Chapter 1

Introduction

Why is the sky blue? Where do butterflies go when it rains? What is the speed of light? Finding the answers to questions such as these would traditionally have involved consulting books or experts; however, in today's fast-paced and technology-oriented society, the most common response to a quest for information is to head for the nearest computer, connect to the Internet, and conduct a search. Online searching has become so much a part of North American culture that *Google*, the name of a popular search engine, is now often used as a verb [16]. Since the inception of the World Wide Web in 1991, the number of existing web pages has increased dramatically from a few thousand to a few billion, with expansion continuing on a daily basis. Although not all web sites contain useful or even accurate information, given the enormous size of the Internet, it is a fairly safe assumption that information pertaining to the subject at hand is available, even though retrieving it may be extremely challenging.

1.1 Information Retrieval

This information retrieval (IR) challenge, much like trying to find the proverbial needle in a haystack, is undertaken by search engines. Search engines build databases, in which available documents are indexed. Then, when a query is made, the database indices can be checked and the most relevant documents returned. Two methods for executing this check are known as *keyword searching* and *content searching*. The most common approach is keyword searching, also known as *literal searching*. It literally matches terms in the query with terms specified, either by the author of a document or by the search engine, as keywords. This sounds like a logical idea, but it does have drawbacks. It is not unusual for a word to have more than one meaning, and this trait, known as *polysemy*, makes it likely that a keyword search will return irrelevant documents [1, 2, 4]. For example, if a query contains the word *mouse*, it may refer to a four-footed furry rodent or to a computer pointing device, and so the documents returned are likely to be a mixture of those about rodents and those about computer hardware. This is known as *precision failure* because the results are not precisely on topic. Whereas precision failure returns irrelevant documents, another type of error, known as *recall failure*, fails to return documents that are relevant. Documents may be about the correct subject, but if they do not contain the exact keywords used in the original query, they will be overlooked in keyword searching [2, 4, 19]. Content searching attempts to avoid recall and precision failure by classifying documents based on a contextual understanding, and returning those that are in the same category as the search query, rather than those that have matching keywords. The problem is that whereas a person can easily determine the focus of a document, it is much more difficult for a computer to do so [10]. Although human experts perform document

classification for some search directories, the vast number of documents now available on the Internet makes this increasingly infeasible [1].

1.2 Latent Semantic Indexing

Latent Semantic Indexing (LSI) is an alternative method of information retrieval that attempts to avoid these problems. LSI uses a mathematical approach to examine the document collection as a whole and determines which documents contain many of the same words. The more words documents have in common, the more closely related they are considered to be. When a query is made, documents containing the keywords are returned, but so are those that are closely related to these documents. This allows the retrieval of documents that do not contain all or even any of the keywords given in the query [1, 19].

LSI involves creating a *term-by-document matrix*, in which there is a vector for each document, with as many entries as there are semantically significant words, or *content words*, in the documents. This is known as a *vector space model* [1, 2, 14]. As with other IR methods, some initial preparation is required. First, the texts are stripped down by removing formatting, punctuation, and words with little semantic meaning such as *the* and *is*. The words that are left are the content words. Next, common word endings are removed in a process known as *stemming*, which produces a more concise list of root words [1, 14]. Finally, these lists of root words from each document are combined into a single list of terms and used to form the term-by-document matrix.

The term-by-document matrix is essentially a table of numbers with, as already noted, a column (vector) for each document, and a row for each term in the combined

list of content root words. This matrix will then be of size $t \times d$, where t denotes the number of terms, and d denotes the number of documents. We will refer to this matrix as $\mathbf{A}$, and to each entry as a_{ij}, where $1 \leq i \leq t$ and $1 \leq j \leq d$. Each entry indicates the presence or absence of a particular term in a particular document. If the term is not present, a 0 is entered in the corresponding matrix position. If the term is contained in a document, a positive number is entered, based on the weight assigned to that particular word in that particular document. Without weighting, a value of 1 can be entered to simply indicate the presence of the word in the document. In practice however, two forms of weighting are commonly used. The first is known as *local weighting*, in which the number of times the word appears in the document is used to calculate the weight, with frequently appearing words receiving a higher weight to indicate their importance [19]. The second type of weighting is *global weighting* [2]. Here the frequency of a word across the entire body of documents is considered. If a word appears in relatively few documents it is considered to be more interesting than a word that appears in many documents, and so it will receive a correspondingly higher weight [19]. Some documents are much larger than others and the size of the document affects the frequency of a term. For this reason, the weights are normalized so that words in smaller documents receive higher weights than those in larger documents with many content words [19].

The following example steps through the process of forming a very small term-by-document matrix.

Example 1.1:

The database consists of a dozen documents that are so short, they each have only one sentence. The terms to be considered content words are underlined:

d_1: Not every mouse is a rodent; not every rodent is a mouse.

d_2: If you move the computer's mouse, it will move the cursor on the computer screen.

d_3: A mouse in your house is better than a rat in your hat.

d_4: The rodent gnawed through the screen to get into the house.

d_5: A computer pointing device is helpful when using a graphics program.

d_6: The computer was difficult to use when the mouse malfunctioned.

d_7: To catch an annoying rat, just use a rodent trap.

d_8: This computer program requires the use of a computer pointing device.

d_9: Whenever I move the mouse, I see squiggly graphics on the computer screen.

d_{10}: Although the story *The Cat in the Hat* is popular, I prefer *The Mouse in the House*.

d_{11}: If you will give me a new mouse, I will move my computer to your house.

d_{12}: When a rodent is as big as a rat, it is time to set a trap.

A list of terms is formed from the document collection by stripping out everything but the underlined content words, and then stemming the word endings and discarding any duplicate terms:

t_1: mouse	t_6: house	t_{11}: graphic
t_2: rodent	t_7: rat	t_{12}: program
t_3: compute	t_8: hat	t_{13}: trap
t_4: cursor	t_9: point	t_{14}: story
t_5: screen	t_{10}:device	t_{15}: cat

These data allow the formation of the term-by-document matrix. For simplicity, global weighting is ignored, and the local weight function is defined as the frequency of a term in a document. The documents are similar in size, so normalization of the weights is not necessary.

Terms	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}
mouse	2	1	1	0	0	1	0	0	1	1	1	0
rodent	2	0	0	1	0	0	1	0	0	0	0	1
compute	0	2	0	0	1	1	0	2	1	0	1	0
cursor	0	1	0	0	0	0	0	0	0	0	0	0
screen	0	1	0	1	0	0	0	0	1	0	0	0
house	0	0	1	1	0	0	0	0	0	1	1	0
rat	0	0	1	0	0	0	1	0	0	0	0	1
hat	0	0	1	0	0	0	0	0	0	1	0	0
point	0	0	0	0	1	0	0	1	0	0	0	0
device	0	0	0	0	1	0	0	1	0	0	0	0
graphic	0	0	0	0	1	0	0	0	1	0	0	0
program	0	0	0	0	1	0	0	1	0	0	0	0
trap	0	0	0	0	0	0	1	0	0	0	0	1
story	0	0	0	0	0	0	0	0	0	1	0	0
cat	0	0	0	0	0	0	0	0	0	1	0	0

Table 1.1: Term-by-Document Matrix $\mathbf{A}$ of size 15×12

A query is represented by a vector formed using a similar process. Suppose the terms *computer*, *pointing*, and *device* are the only terms to be used in a query. After stripping out the insignificant words and stemming the remaining terms, a $t \times 1$ vector is created, where t is, as before, the number of terms in the term-by-document matrix. If a strictly binary approach is used, an entry of 1 is made when the corresponding term is present in the query, and an entry of 0 otherwise. Using the same terms as for the term-by-document matrix thus forms the vector $\mathbf{q} = (0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0)^T$.

The process of searching involves calculating the distances between the query and document vectors; the closer together the vectors are, the more closely related they are considered to be [2]. Each query can be thought of as a probe into the document vector space [1]. A query is projected into the term-by-document matrix using the matrix factorization method known as the *singular value decomposition.*

1.3 Singular Value Decomposition Overview

The singular value decomposition (SVD) is a matrix decomposition whose computation has many applications [17]. It is popular for a number of reasons, including the fact that it can be computed in a stable manner [15]. This means that small perturbations in the data, such as those caused by rounding errors, cause only small perturbations in the resulting factorization. The SVD also gives optimal low-rank approximations to the original matrix, providing a means of data compression.

Because of the tremendous size of modern databases, a term-by-document matrix can potentially be very large, with hundreds of thousands, or even millions, of entries. The SVD is a mathematical method of reducing the size of these large matrices, while maintaining the relationship among the vectors [2]. We can think of the term-by-document matrix as representing a t-dimensional space, with t-dimensional document vectors. Each vector contains the coordinates of that document's location in the t-dimensional space. The vectors of documents with many terms in common will be found close together, whereas the vectors of documents with relatively few terms in common will be located far apart. The SVD projects these vectors into a much

smaller subspace. This results in the loss of some information, but in this case, this is a positive consequence. Looking at the example above, it is easy to see that the matrix is composed mainly of zeroes, since each document tends to contain relatively few of the terms. This is generally the case, with an average 99% of the entries being zero [4]. Such a matrix is referred to as being *sparse*. The SVD can be used to decompose the sparse matrix, discard the information that is least important, and return a much smaller approximation that represents a condensing of the data contained in the original matrix. The data in the condensed, or compressed, matrix contain the latent patterns that indicate the presence of semantically similar terms, whereas the noise in the original matrix, caused by factors such as polysemy and synonymy, has been largely damped or removed [2, 13, 19]. This SVD data compression process is extremely useful, but traditional methods of calculating the SVD are very expensive in terms of computational cost. In fact, most of the processing time in LSI is spent performing this SVD calculation [3, 4]. This thesis describes a recent approach to calculating the SVD that is significantly faster than traditional methods.

1.4 Singular Value Decomposition in More Detail

Given a matrix $\mathbf{A} \in \Re^{m \times n}$, its SVD is written as

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T, \tag{1.1}$$

where $\mathbf{U} \in \Re^{m \times m}$, $\mathbf{V} \in \Re^{n \times n}$, and $\mathbf{\Sigma} \in \Re^{m \times n}$. $\mathbf{U}$ and $\mathbf{V}$ are *orthogonal matrices*.

An orthogonal matrix is a square matrix that has orthonormal columns. Geometrically, this means that the columns are mutually perpendicular, and that each has an Euclidean length of one. The matrix $\mathbf{\Sigma} \in \Re^{m \times n}$ has non-zero entries only on the diagonal. These diagonal entries, denoted σ_j for $j = 1, 2, ..., \min(m, n)$, are the *singular values* of matrix $\mathbf{A}$. The left singular vectors of $\mathbf{A}$ are the columns of $\mathbf{U}$ and the right singular vectors are the columns of $\mathbf{V}$. Every matrix has a singular value decomposition, and the *singular values* $\{\sigma_j\}$ are always uniquely determined [17]. The number of nonzero singular values of a matrix is known as the *rank*, r, of that matrix, and corresponds with the number of linearly independent rows (or columns).

An alternative way to represent the SVD is as the sum of r rank-one matrices

$$\mathbf{A} = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^T,$$

where $\mathbf{u}_j$ and $\mathbf{v}_j$ are the *jth* columns of matrices $\mathbf{U}$ and $\mathbf{V}$, respectively. This representation of the SVD allows the formation of lower-rank approximations of $\mathbf{A}$. Replacing r in this equation by any ν with $0 \leq \nu < r$ gives

$$\mathbf{A} \approx \sum_{j=1}^{\nu} \sigma_j \mathbf{u}_j \mathbf{v}_j^T.$$

Taking the first ν columns of $\mathbf{U}$ and $\mathbf{V}$ and the leading $\nu \times \nu$ submatrix of $\mathbf{\Sigma}$, we obtain the optimal rank-ν approximation of $\mathbf{A}$, which can be expressed as

$$\mathbf{A}_\nu = \mathbf{U}_\nu \mathbf{\Sigma}_\nu \mathbf{V}_\nu^T. \tag{1.2}$$

This approximation is optimal in the sense that there exists no rank-ν approximation that is a better approximation to the original matrix $\mathbf{A}$ [4, 13, 17]. For example, in

the 2-norm

$$\|\mathbf{A} - \mathbf{A}_\nu\|_2 = \min_{\substack{\text{rank}(\mathbf{B}) \leq \nu \\ \mathbf{B} \in \Re^{m \times n}}} \|\mathbf{A} - \mathbf{B}\|_2 = \sigma_{\nu+1}.$$

As described above, the approximation can be used to reduce the dimension of the term-by-document matrix, while eliciting the underlying structure of the data. How many terms to keep in the reduced term-by-document matrix when n is very large is still open to study and debate, but experiments indicate that values of ν between 100 and 300 give the best results [4]. This tremendous dimensional reduction, given the potentially huge size of the term-by-document matrix, demonstrates the power of the SVD as a method of data compression.

A simple example will illustrate the formation of the SVD.

Example 1.2:

Let $\mathbf{A}$ be the 15×12 term-by-document matrix from Example 1.1, and for convenience in plotting, let $\nu = 2$. Recalling (1.1) and (1.2), we write

$$\mathbf{A}_2 = \mathbf{U}_2 \mathbf{\Sigma}_2 \mathbf{V}_2^T,$$

where $\mathbf{A}_2 \in \Re^{15 \times 12}$, $\mathbf{U}_2 \in \Re^{15 \times 2}$, $\mathbf{\Sigma}_2 \in \Re^{2 \times 2}$, and $\mathbf{V}_2^T \in \Re^{2 \times 12}$. Performing this calculation in *Matlab* gives the following results (where only 4 decimal places of accuracy are displayed).

$$\mathbf{U}_2 = \begin{pmatrix} -0.5615 & -0.4186 \\ -0.2162 & -0.5002 \\ -0.6609 & 0.4643 \\ -0.1089 & 0.0449 \\ -0.2175 & -0.0025 \\ -0.1922 & -0.2423 \\ -0.0776 & -0.2266 \\ -0.0917 & -0.1658 \\ -0.1484 & 0.2397 \\ -0.1484 & 0.2397 \\ -0.1391 & 0.1190 \\ -0.1484 & 0.2397 \\ -0.0321 & -0.1410 \\ -0.0462 & -0.0802 \\ -0.0462 & -0.0802 \end{pmatrix}, \mathbf{V}_2 = \begin{pmatrix} -0.3452 & -0.5238 \\ -0.4904 & 0.1575 \\ -0.2049 & -0.3003 \\ -0.1389 & -0.2124 \\ -0.2764 & 0.3713 \\ -0.2713 & 0.0130 \\ -0.0723 & -0.2474 \\ -0.3922 & 0.4697 \\ -0.3505 & 0.0463 \\ -0.2081 & -0.2814 \\ -0.3140 & -0.0560 \\ -0.0723 & -0.2474 \end{pmatrix},$$

$$\mathbf{\Sigma}_2 = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} = \begin{pmatrix} 4.5053 & 0 \\ 0 & 3.5081 \end{pmatrix}.$$

$\mathbf{U}_2$ contains term vectors, and $\mathbf{V}_2$ contains document vectors. The terms and documents can be plotted on a two-dimensional graph with term coordinates (x_{t_i}, y_{t_i}) and document coordinates (x_{d_j}, y_{d_j}), where

$$1 \leq i \leq 15, \quad 1 \leq j \leq 12,$$

and

$$\begin{aligned} x_{t_i} &= \sigma_1 \mathbf{U}_2(i,1), \quad & y_{t_i} &= \sigma_2 \mathbf{U}_2(i,2), \\ x_{d_j} &= \sigma_1 \mathbf{V}_2(j,1), \quad & y_{d_j} &= \sigma_2 \mathbf{V}_2(j,2). \end{aligned}$$

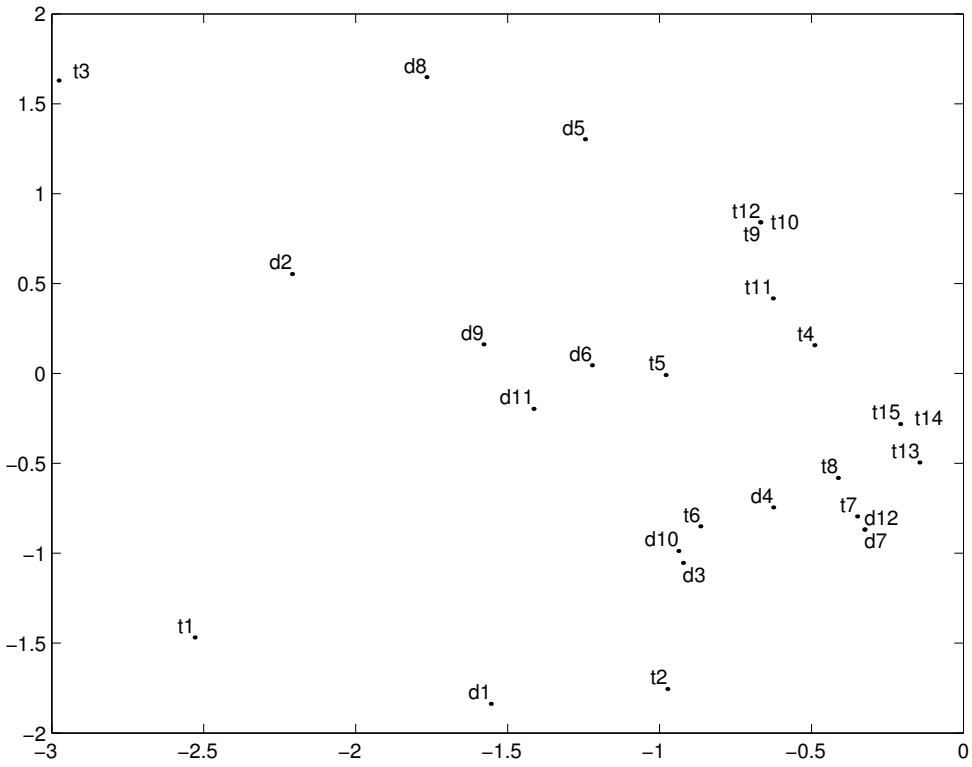

Figure 1.1: Two-Dimensional Plot of 15 Terms and 12 Documents

t_1: mouse	t_6: house	t_{11}: graphic
t_2: rodent	t_7: rat	t_{12}: program
t_3: compute	t_8: hat	t_{13}: trap
t_4: cursor	t_9: point	t_{14}: story
t_5: screen	t_{10}:device	t_{15}: cat

Note that terms such as *point* (t_9), *device* (t_{10}), and *program* (t_{12}) map to the same point because they always occur together. Documents with few terms tend to map approximately equidistant from each term. Document d_6, for example, contains only terms t_3 and t_1; on this plot it is found about an equal distance from each of these terms. Documents that have terms in common, such as d_7 and d_{12}, tend to be found close together. This automatic clustering of related terms and related documents is

one of the great advantages of using LSI and low-rank approximations [1].

Recall, from Example 1.1, that a search using the terms *computer*, *pointing*, and *device* results in the query vector $\mathbf{q} = (0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0)^T$. To project this query into a two-dimensional subspace, it is approximated using the formula [3]

$$\hat{\mathbf{q}} = \mathbf{q}^T \mathbf{U}_2 \mathbf{\Sigma}_2^{-1}.$$

This gives

$$\hat{\mathbf{q}} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} -0.5615 & -0.4186 \\ -0.2162 & -0.5002 \\ -0.6609 & 0.4643 \\ -0.1089 & 0.0449 \\ -0.2175 & -0.0025 \\ -0.1922 & -0.2423 \\ -0.0776 & -0.2266 \\ -0.0917 & -0.1658 \\ -0.1484 & 0.2397 \\ -0.1484 & 0.2397 \\ -0.1391 & 0.1190 \\ -0.1484 & 0.2397 \\ -0.0321 & -0.1410 \\ -0.0462 & -0.0802 \\ -0.0462 & -0.0802 \end{pmatrix} \begin{pmatrix} 0.2220 & 0 \\ 0 & 0.2851 \end{pmatrix} = \begin{pmatrix} -0.2126 & 0.2690 \end{pmatrix}.$$

This compressed query vector is compared with the document vectors contained in $\mathbf{V}_2$, and those documents that are most closely related are considered relevant. One method of determining document relevancy is to measure the cosine of the angle

between the query vector and each of the document vectors. If the cosine of the angle is greater than a chosen threshold, then the document is returned as relevant [3].

The two-dimensional query vector is plotted in Figure 1.2. The shaded portion of the figure represents the area of relevance defined by a cosine threshold of 0.87. Because $\cos^{-1} 0.87 \approx 30°$, this is the area spanning approximately 30° on either side of the query vector.

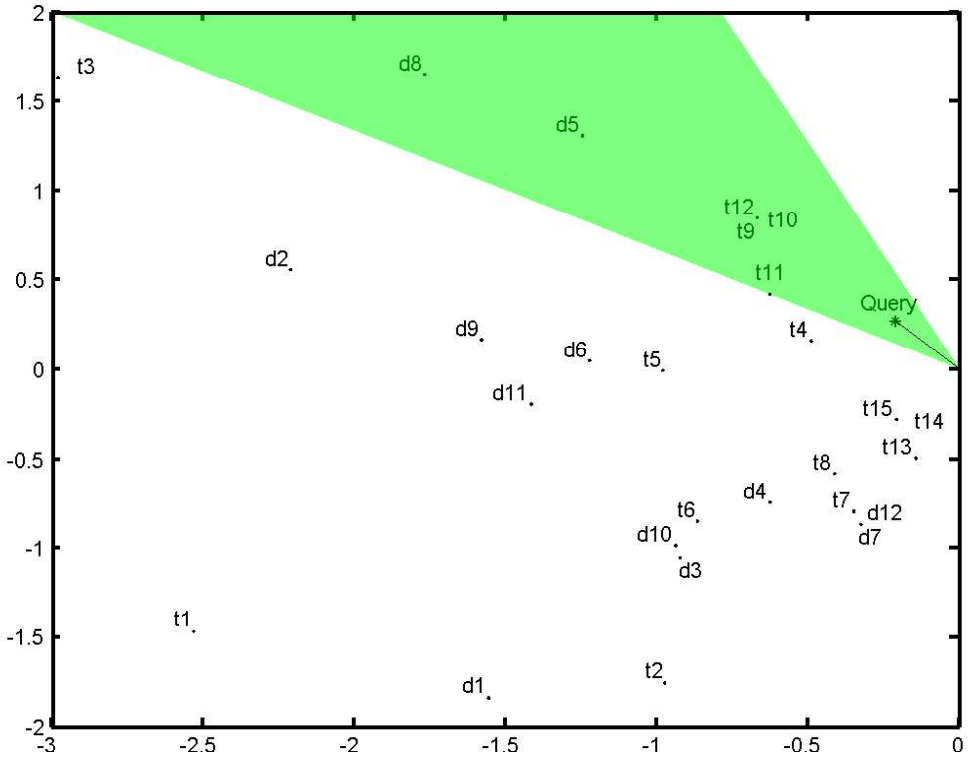

Figure 1.2: Two-Dimensional Plot Using a Cosine Threshold of 0.87

At this threshold, the documents d_5 and d_8, which contain all the query terms, are returned. Using a lower cosine value returns more documents considered to be relevant.

Figure 1.3 uses a cosine threshold of 0.53 to select documents. As before, the shaded portion of the figure indicates the area of relevance: the area spanning approximately 58° on either side of the query vector.

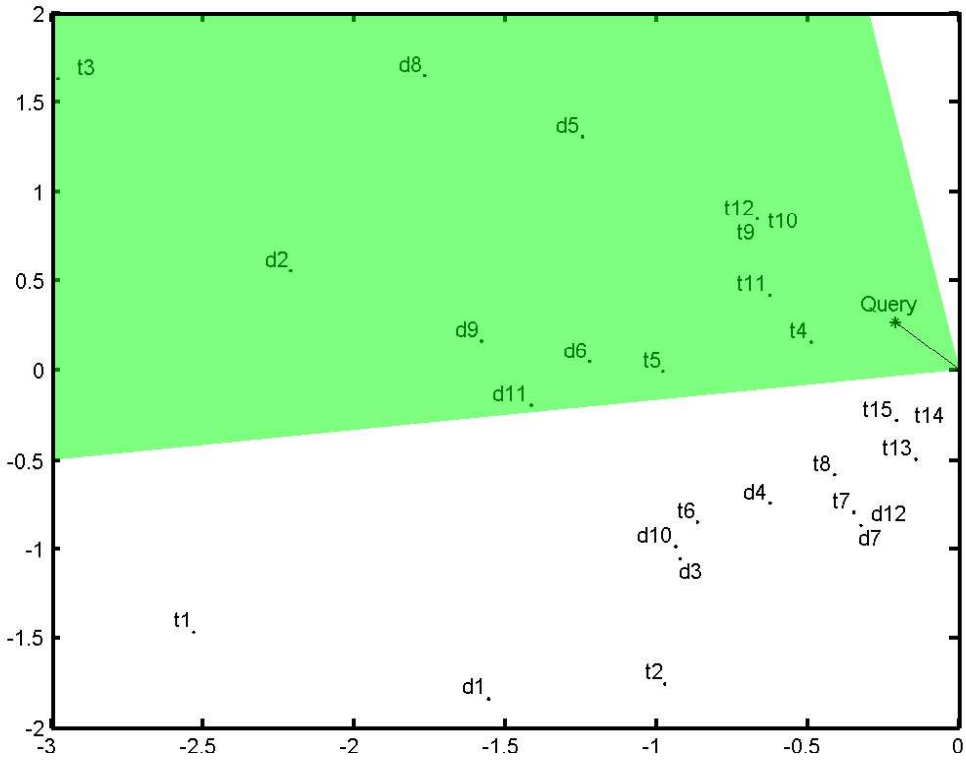

Figure 1.3: Two-Dimensional Plot Using a Cosine Threshold of 0.53

In this example, all the documents relevant to the query are returned, including those that do not contain the query terms *pointing* and *device*. No irrelevant documents are returned. The clustering of documents provides a means to distinguish between the use of the term *mouse* as a computer hardware component, and the use of the term *mouse* as a rodent, thus allowing the selection of the appropriate documents. This is an illustration of the ability of the vector space model to address the problems of recall and precision error.

1.5 Statement of Problem and Goals

The calculation of the SVD in IR vector space models such as LSI constitutes the greatest computational expense incurred by these models [3, 4]. Calculating the SVD of a matrix typically involves two main steps: first the matrix is bidiagonalized, and then the SVD is formed from this bidiagonal matrix. Note that a bidiagonal matrix has nonzero entries only on the diagonal and either on the first subdiagonal, in the case of a lower bidiagonal matrix, or on the first superdiagonal, in the case of an upper bidiagonal matrix. As already illustrated, the term-by-document matrices formed in LSI are sparse; such matrices can be efficiently bidiagonalized by using the Lanczos algorithm [7, 18], which takes advantage of the sparsity. The second step in the SVD calculation is traditionally performed using the Golub-Kahan algorithm [6, 7]. Unfortunately this method of computing the SVD of a size $m \times n$ bidiagonal matrix takes $\mathcal{O}(n^3)$ time when both the singular values and the singular vectors are required [8]. This thesis presents an alternative method of calculating the SVD, known as the *divide-and-conquer* SVD, which is computationally more efficient.

The following chapter gives an overview of the Lanczos and Golub-Kahan algorithms. Chapter 3 provides a detailed description of a divide-and-conquer algorithm for the SVD, including several deflation techniques, proposed by Gu and Eisenstat [8]. Chapter 4 gives an evaluation of the divide-and-conquer SVD implementation, while Chapter 5 discusses conclusions and future work.

Chapter 2

Lanczos Bidiagonalization and Golub-Kahan SVD Algorithms

The efficiency of an algorithm involving matrix computations is affected both by the amount of storage needed and by the number of arithmetic operations performed [7]. Some matrix structures, such as that of the bidiagonal matrix, provide sufficient savings in both storage and arithmetic operations that the cost of performing a change in structure is worthwhile. Thus, the first step in calculating the SVD of a matrix is generally that of bidiagonalization [18], and this is true of the divide-and-conquer approach to be described in this thesis. The divide-and-conquer SVD implementation to be discussed in Chapter 3 uses the Lanczos algorithm [6, 7] for matrix bidiagonalization, which produces an approximate bidiagonal decomposition of the original matrix. An overview of the algorithm is given in the first section of this chapter.

Once a matrix has been bidiagonalized, the traditional method of calculating the SVD is to employ the Golub-Kahan algorithm [6, 7]. As already noted, the running time of this algorithm for an $m \times n$ bidiagonal matrix is $\mathcal{O}(n^3)$ when both the singular values and the singular vectors are required [8]. This is efficient when n is small;

therefore the divide-and-conquer implementation to be discussed uses this method at the base case when computing the SVD of small matrices. An overview of the algorithm is presented in the second section of this chapter.

2.1 Lanczos Bidiagonalization Algorithm

The Lanczos bidiagonalization algorithm was introduced by Golub and Kahan [6], but it is based on the Lanczos algorithm for tridiagonalizing Hermitian matrices [6, 7, 12]. The Lanczos bidiagonalization algorithm produces a matrix that is an bidiagonal approximation of the original matrix. The result is an approximation rather than an exact transformation because the Lanczos algorithm is unstable in the presence of rounding errors [6, 12], resulting in a loss of orthogonality of the vectors it generates (i.e., matrices $\mathbf{U}$ and $\mathbf{V}$ in (2.1) below). In spite of this problem, it maintains a good approximation of the large singular values of $\mathbf{A}$, although the approximation of the smaller singular values is less accurate [7]. For applications such as LSI that use only the ν largest singular values for the rank-ν approximation of $\mathbf{A}$, the Lanczos algorithm is an acceptable bidiagonalization method. This is especially true because, as illustrated in Chapter 1, the term-by-document matrices generated by LSI are sparse, and the Lanczos algorithm is an efficient method for bidiagonalizing large sparse matrices [12]. The algorithm preserves the sparsity of the matrix [7], providing advantages in storage space and computation time [12].

In theory, the Lanczos bidiagonalization algorithm computes orthogonal matrices $\mathbf{U} \in \Re^{m \times n}$ and $\mathbf{V} \in \Re^{n \times n}$ such that

$$\mathbf{U^T AV} = \mathbf{B} = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ & \alpha_2 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{n-1} & \beta_{n-1} \\ & & & & \alpha_n \end{pmatrix}. \tag{2.1}$$

In practice, as already noted, rounding errors due to finite precision cause the columns of $\mathbf{U}$ and $\mathbf{V}$ to lose orthogonality [6, 12], giving

$$\mathbf{U^T AV} = \mathbf{B}_{\text{Lanczos}} \approx \mathbf{B}.$$

The algorithm can be summarized as follows.

Algorithm 2.1 LANCZOS BIDIAGONALIZATION ALGORITHM

1: choose $\mathbf{v}_1 \in \Re^n$ such that $\|\mathbf{v}_1\|_2 = 1$
2: set $\mathbf{p}_0 = \mathbf{v}_1 \in \Re^n$ and $\mathbf{u}_0 = \mathbf{0} \in \Re^m$
3: set $k = 0$ and $\beta_0 = 1$
4: **while** $\beta_k \neq 0$ **do**
5: $\quad \mathbf{v}_{k+1} = \mathbf{p}_k / \beta_k$
6: $\quad k = k + 1$
7: $\quad \mathbf{r}_k = \mathbf{A}\mathbf{v}_k - \beta_{k-1}\mathbf{u}_{k-1}$
8: $\quad \alpha_k = \|\mathbf{r}_k\|_2$
9: $\quad \mathbf{u}_k = \mathbf{r}_k / \alpha_k$
10: $\quad \mathbf{p}_k = \mathbf{A}^T\mathbf{u}_k - \alpha_k\mathbf{v}_k$
11: $\quad \beta_k = \|\mathbf{p}_k\|_2$
12: **end while**

Note that this algorithm assumes matrix $\mathbf{A} \in \Re^{m \times n}$, where $m \geq n$. If this is not the case, $\mathbf{A}$ can simply be transposed before the bidiagonalization takes place. Likewise, the upper bidiagonal matrix produced by the procedure can easily be transposed to a lower bidiagonal matrix if necessary.

2.2 Golub-Kahan SVD Algorithm

The Golub-Kahan algorithm was introduced in 1965 [6] and soon became the standard method for computing the SVD of a matrix. To facilitate the computation of the SVD, the original matrix $\mathbf{A}$ is typically bidiagonalized to $\mathbf{B}$ [6]. In this case, $\mathbf{B}$ must be square, upper bidiagonal, and have no zero entries on either the diagonal or the first super diagonal. The algorithm then calculates the SVD of $\mathbf{B}$ using *Givens transformations*, also known as *Givens rotations* [18], on the left and right of $\mathbf{B}$ to *zero out*, or transform to zero, the elements on the super diagonal of the matrix [7]. This results in the diagonal matrix $\mathbf{\Sigma}$, containing the singular values of $\mathbf{A}$.

A Givens rotation is a multiplication by an orthogonal matrix. For a 2×2 matrix, it takes the form

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix},$$

where $c^2 + s^2 = 1$. The structure of a Givens rotation of arbitrary size $(n \times n)$ to be applied from the left is as follows:

$$G = \begin{pmatrix} 1 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 1 & & & & & & & \\ & & & c & 0 & \cdots & 0 & s & & \\ & & & 0 & 1 & & & 0 & & \\ & & & \vdots & & \ddots & & \vdots & & \\ & & & 0 & & & 1 & 0 & & \\ & & & -s & 0 & \cdots & 0 & c & & \\ & & & & & & & & 1 & \\ & & & & & & & & & \ddots \\ & & & & & & & & & & 1 \end{pmatrix}, \tag{2.2}$$

where $\mathbf{B}_{k,i}$ is the element to be zeroed out, c is in positions $\mathbf{G}_{i,i}$ and $\mathbf{G}_{k,k}$, s is in position $\mathbf{G}_{i,k}$, and $-s$ is in position $\mathbf{G}_{k,i}$, for $i \neq k$ and $0 < i, k \leq n$.

For an arbitrary vector $\mathbf{v}$ in which element v_k is to be zeroed out, the values of c and s are determined using the equations

$$c = \frac{v_i}{\sqrt{v_i^2 + v_k^2}}$$

$$s = \frac{v_k}{\sqrt{v_i^2 + v_k^2}},$$

where $i \neq k$ and $0 < i, k \leq n$.

The Givens rotations acting on $\mathbf{B}$ from the left and right can be expressed as

$$\mathbf{U}^T \mathbf{B} \mathbf{V},$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices composed of the products of the Givens rotations $\mathbf{U}_i^T$ and $\mathbf{V}_i$ such that

$$\mathbf{U}^T = \mathbf{U}_{n-1}^T \mathbf{U}_{n-2}^T \ldots \mathbf{U}_2^T \mathbf{U}_1^T$$

and

$$\mathbf{V} = \mathbf{V}_1 \mathbf{V}_2 \ldots \mathbf{V}_{n-2} \mathbf{V}_{n-1}.$$

This gives

$$\mathbf{\Sigma} = \mathbf{U}^T \mathbf{B} \mathbf{V},$$

where

$$\mathbf{B} = \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T$$

is the SVD of $\mathbf{B}$.

This is a rather simplistic overview of the Golub-Kahan SVD algorithm. Detailed descriptions are given by Golub and Kahan [6] and by Golub and Van Loan [7]. As previously noted, this algorithm takes $\mathcal{O}(n^3)$ time to compute the SVD of an $m \times n$ bidiagonal matrix [8]. This becomes increasingly expensive as n gets large. The divide-and-conquer algorithm described in Chapter 3 can be optimized to compute the singular values in $\mathcal{O}(n \log_2 n)$ time and the singular values and singular vectors

in $\mathcal{O}(n^2)$ [8]. This is of particular importance to applications such as LSI, in which the SVD is calculated many times, and for matrices where n is very large.

Chapter 3

Divide-and-Conquer SVD Algorithm

In the world of computer science, many algorithms use the divide-and-conquer design paradigm [5]. This method works recursively, breaking down the original problem into related subproblems, and then applying the algorithm to each of the subproblems until a trivial base case is reached. The subproblems are solved independently, and then their solutions are combined to produce a solution to the original problem. An advantage of this paradigm is a reduction in the complexity of the algorithm in comparison with non-recursive methods. Consider a problem of size n that takes $\mathcal{O}(n^2)$ time using a traditional method. Combining two solutions of size $\frac{n}{2}$ takes $\mathcal{O}(n)$ time, so the problem can be solved in $\mathcal{O}(n \log_2 n)$ time using a divide-and-conquer approach. Although divide-and-conquer may be slower than traditional methods for small data sets, due to the extra overhead from recursion, solving such small subproblems using a traditional algorithm reduces this liability. This chapter describes a divide-and-conquer SVD algorithm proposed by Gu and Eisenstat [8].

3.1 Dividing into Subproblems

Suppose there is a matrix $\mathbf{A} \in \Re^{m \times n}$ for which the SVD must be computed. Gu and Eisenstat's divide-and-conquer SVD algorithm takes as its argument a lower bidiagonal matrix $\mathbf{B} \in \Re^{(n+1) \times n}$, and therefore the first step in the SVD implementation is to transform $\mathbf{A}$ to this form. As described in Section 2.1, the Lanczos algorithm may be used to bidiagonalize the matrix. If the matrix produced is of size $n \times n$ rather than $(n+1) \times n$, a zero row is appended to the bottom of the matrix.

Given $\mathbf{B} \in \Re^{(n+1) \times n}$, the next step is to divide the matrix into two subproblems, $\mathbf{B}_1 \in \Re^{k \times (k-1)}$, and $\mathbf{B}_2 \in \Re^{(n-k+1) \times (n-k)}$, where $\mathbf{B}_1$, and $\mathbf{B}_2$ are lower bidiagonal matrices, and $1 < k < n$. Note that k is typically assigned the value $\lfloor \frac{n}{2} \rfloor$ [8].

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_1 & \alpha_k \mathbf{e}_k & \mathbf{0} \\ \mathbf{0} & \beta_k \mathbf{e}_1 & \mathbf{B}_2 \end{pmatrix},$$

where $\mathbf{e}_j$ is the jth *unit vector* of the appropriate size [8]. The jth unit vector is the jth column of the *identity matrix*

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{pmatrix},$$

whose dimension is determined by the context in which it is used. Here, $\mathbf{e}_k$ is the kth column of $\mathbf{I} \in \Re^{k \times k}$ and $\mathbf{e}_1$ is the first column of $\mathbf{I} \in \Re^{(n+1-k) \times (n+1-k)}$.

Once $\mathbf{B}$ has been partitioned, the SVD of each of the two subproblems, $\mathbf{B}_1$ and $\mathbf{B}_2$, is computed. The solution of the subproblems is done recursively until they are

reduced to a trivial base case, at which time it is solved using a traditional, non-recursive SVD method. The implementation discussed here applies the Golub-Kahan SVD algorithm at this step. The dimension of the base case is established using empirical evidence. For the particular implementations tested in this thesis, when $n \leq 15$, the traditional Golub-Kahan SVD implementation [11] without the overhead of recursion outperforms the divide-and-conquer method. Figure 3.1 illustrates this.

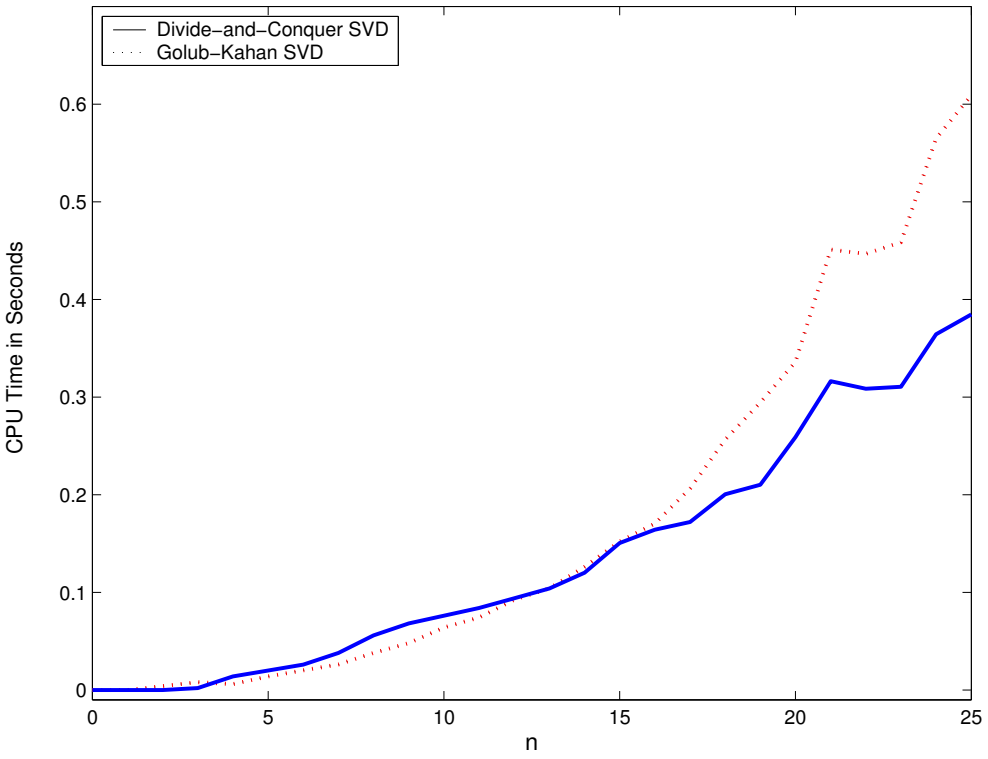

Figure 3.1: An Initial Comparison of CPU Times for Divide-and-Conquer and Golub-Kahan Implementations ($1 \leq n \leq 25$)

The divide-and-conquer implementation discussed here therefore uses $n \leq 15$ as the base case. However, no matter what dimension is used for the base case, there cannot be more than $\mathcal{O}(\log_2 n)$ levels of recursion.

3.2 Combining Subproblem Solutions

The bulk of the divide-and-conquer SVD algorithm consists of computing the SVD of $\mathbf{B}$, utilizing the SVDs of $\mathbf{B}_1$ and $\mathbf{B}_2$. For the purpose of illustration, only these two subproblems will be considered. Let the SVD of each $\mathbf{B}_i$ be

$$\mathbf{B}_i = \begin{pmatrix} \mathbf{Q}_i & \mathbf{q}_i \end{pmatrix} \begin{pmatrix} \mathbf{D}_i \\ \mathbf{0} \end{pmatrix} \mathbf{W}_i^T.$$

The SVDs of $\mathbf{B}_1$ and $\mathbf{B}_2$ can be combined in the following manner to create matrices whose product is $\mathbf{B}$.

$$\mathbf{B} = \begin{pmatrix} \mathbf{q}_1 & \mathbf{Q}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_2 & \mathbf{q}_2 \end{pmatrix} \begin{pmatrix} \alpha_k\lambda_1 & \mathbf{0} & \mathbf{0} \\ \alpha_k\mathbf{l}_1 & \mathbf{D}_1 & \mathbf{0} \\ \beta_k\mathbf{f}_2 & \mathbf{0} & \mathbf{D}_2 \\ \beta_k\varphi_2 & \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{0} & \mathbf{W}_1 & \mathbf{0} \\ 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{W}_2 \end{pmatrix}^T, \qquad (3.1)$$

where $\mathbf{l}_1^T$ is the last row of $\mathbf{Q}_1$, λ_1 is the last element of $\mathbf{q}_1$, $\mathbf{f}_2^T$ is the first row of $\mathbf{Q}_2$, and φ_2 is the first element of $\mathbf{q}_2$. An identity matrix of the form $\mathbf{I} = \mathbf{G}^T\mathbf{G}$ is inserted between the first two matrices on the right side of (3.1), where $\mathbf{G}$ is a Givens rotation (and hence orthogonal) chosen such that it zeros out the element $\beta_k\varphi_2$.

$$\mathbf{I} = \mathbf{G}^T\mathbf{G} \equiv \begin{pmatrix} c_0 & & & & -s_0 \\ & 1 & & & \\ & & \vdots & & \\ & & & 1 & \\ s_0 & & & & c_0 \end{pmatrix} \begin{pmatrix} c_0 & & & & s_0 \\ & 1 & & & \\ & & \vdots & & \\ & & & 1 & \\ -s_0 & & & & c_0 \end{pmatrix}, \tag{3.2}$$

with $c_0 = \frac{\alpha_k\lambda_1}{r_0}$, $s_0 = \frac{\beta_k\varphi_2}{r_0}$, and $r_0 = \sqrt{(\alpha_k\lambda_1)^2 + (\beta_k\varphi_2)^2}$. This gives

$$\begin{aligned} \mathbf{B} &= \left(\left(\begin{pmatrix} \mathbf{q}_1 & \mathbf{Q}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_2 & \mathbf{q}_2 \end{pmatrix} \mathbf{G}^T \right) \left(\mathbf{G} \begin{pmatrix} \alpha_k\lambda_1 & \mathbf{0} & \mathbf{0} \\ \alpha_k\mathbf{l}_1 & \mathbf{D}_1 & \mathbf{0} \\ \beta_k\mathbf{f}_2 & \mathbf{0} & \mathbf{D}_2 \\ \beta_k\varphi_2 & \mathbf{0} & \mathbf{0} \end{pmatrix} \right) \right) \begin{pmatrix} \mathbf{0} & \mathbf{W}_1 & \mathbf{0} \\ 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{W}_2 \end{pmatrix}^T \\ &\equiv \left(\left(\begin{pmatrix} c_0\mathbf{q}_1 & \mathbf{Q}_1 & \mathbf{0} \\ s_0\mathbf{0} & \mathbf{0} & \mathbf{Q}_2 \end{pmatrix} \begin{pmatrix} -s_0\mathbf{q}_1 \\ c_0\mathbf{q}_2 \end{pmatrix} \right) \begin{pmatrix} r_0 & \mathbf{0} & \mathbf{0} \\ \alpha_k\mathbf{l}_1 & \mathbf{D}_i & \mathbf{0} \\ \beta_k\mathbf{f}_2 & \mathbf{0} & \mathbf{D}_2 \\ 0 & \mathbf{0} & \mathbf{0} \end{pmatrix} \right) \begin{pmatrix} \mathbf{0} & \mathbf{W}_1 & \mathbf{0} \\ 1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{W}_2 \end{pmatrix}^T \\ &\equiv \begin{pmatrix} \mathbf{Q} & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{M} \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T. \end{aligned} \tag{3.3}$$

Rearranging (3.3) gives

$$\mathbf{W}\,\mathbf{B}\begin{pmatrix}\mathbf{Q} & \mathbf{q}\end{pmatrix}^T = \begin{pmatrix}\mathbf{M}\\ \mathbf{0}\end{pmatrix}. \tag{3.4}$$

Note that $\mathbf{W}$ and $(\mathbf{Q}\ \ \mathbf{q})^T$ are orthogonal matrices whose application transforms the bidiagonal matrix $\mathbf{B}$ to $\binom{\mathbf{M}}{\mathbf{0}}$, where $\mathbf{M} \in \Re^{n\times n}$ has nonzero elements only in the first column and on the diagonal [8].

3.3 Computing the Singular Values

Because matrix $\mathbf{M}$ is an orthogonal transformation of $\mathbf{B}$, the singular values of $\mathbf{M}$ are also the singular values of $\mathbf{B}$. Now $\mathbf{M}$ is of the form

$$\mathbf{M} = \begin{pmatrix} z_1 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{pmatrix}. \tag{3.5}$$

This unique form facilitates the computation of the singular values using the secular equation (3.9) below, but there are two other conditions that $\mathbf{M}$ must fulfill in order for its singular values to be computed using this method. First some notation is introduced. For a given precision, assumed here to be double precision, let *machine epsilon*, denoted ϵ_{mach}, be the smallest positive number which, when added to 1, gives a number different from 1. Let τ be a small empirically chosen multiple of ϵ_{mach}, and

let $d_1 = 0$ and $\mathbf{D} = \text{diag}(d_1, \ldots, d_n)$. Computing the singular values of $\mathbf{M}$ using the method to be discussed requires that

$$d_{j+1} - d_j \geq \tau \|\mathbf{M}\|_2, \tag{3.6}$$

and

$$|z_j| \geq \tau \|\mathbf{M}\|_2. \tag{3.7}$$

Fortunately, any matrix in the form illustrated by (3.5) can be reduced to a matrix that satisfies these conditions by permuting the matrix and using a deflation method [8] to be discussed in Section 3.5.

Let the SVD of $\mathbf{M}$ be $\mathbf{U}\mathbf{\Omega}\mathbf{V}^T$, where

$$\mathbf{U} = (\mathbf{u}_1, \cdots, \mathbf{u}_n), \quad \Omega = \text{diag}(\omega_1, \cdots, \omega_n), \quad \text{and } \mathbf{V} = (\mathbf{v}_1, \cdots, \mathbf{v}_n).$$

If $\mathbf{M}$ is in the correct form and meets conditions (3.6) and (3.7), then the singular values of $\mathbf{M}$ satisfy an *interlacing property* [8, 9]. This means that the singular values of $\mathbf{M}$ lie between the corresponding diagonal entries of $\mathbf{M}$ such that

$$0 = d_1 < \omega_1 < d_2 < \cdots < d_n < \omega_n < d_n + \|\mathbf{z}\|_2, \tag{3.8}$$

where the singular values ω_i are the exact roots of the *secular equation* [8, 9]:

$$f(\omega) = 1 + \sum_{j=1}^{n} \frac{z_j^2}{d_j^2 - \omega^2} = 0. \tag{3.9}$$

The secular equation is used when computing the singular values of $\mathbf{M}$. The procedure used to find the n roots of (3.9) is now outlined. For ω_i, where $1 \leq i < n$, if $f\left(\frac{d_i+d_{i+1}}{2}\right) > 0$, then $\omega_i \in \left(d_i, \frac{d_i+d_{i+1}}{2}\right)$, else $\omega_i \in \left[\frac{d_i+d_{i+1}}{2}, d_{i+1}\right)$. This is because

$$\begin{aligned} \lim_{\omega \to d_i^+} f(\omega) &= \mathcal{O}(1) + \lim_{\omega \to d_i^+} \frac{z_i^2}{d_i^2 - \omega^2} \\ &= \mathcal{O}(1) + \lim_{\omega \to d_i^+} \frac{z_i^2}{d_i - \omega} = -\infty, \end{aligned}$$

and

$$\begin{aligned} \lim_{\omega \to d_{i+1}^-} f(\omega) &= \mathcal{O}(1) + \lim_{\omega \to d_{i+1}^-} \frac{z_{i+1}^2}{d_{i+1}^2 - \omega^2} \\ &= \mathcal{O}(1) + \lim_{\omega \to d_i^-} \frac{z_{i+1}^2}{d_{i+1} - \omega} = +\infty. \end{aligned}$$

This behaviour is depicted in Figure 3.2.

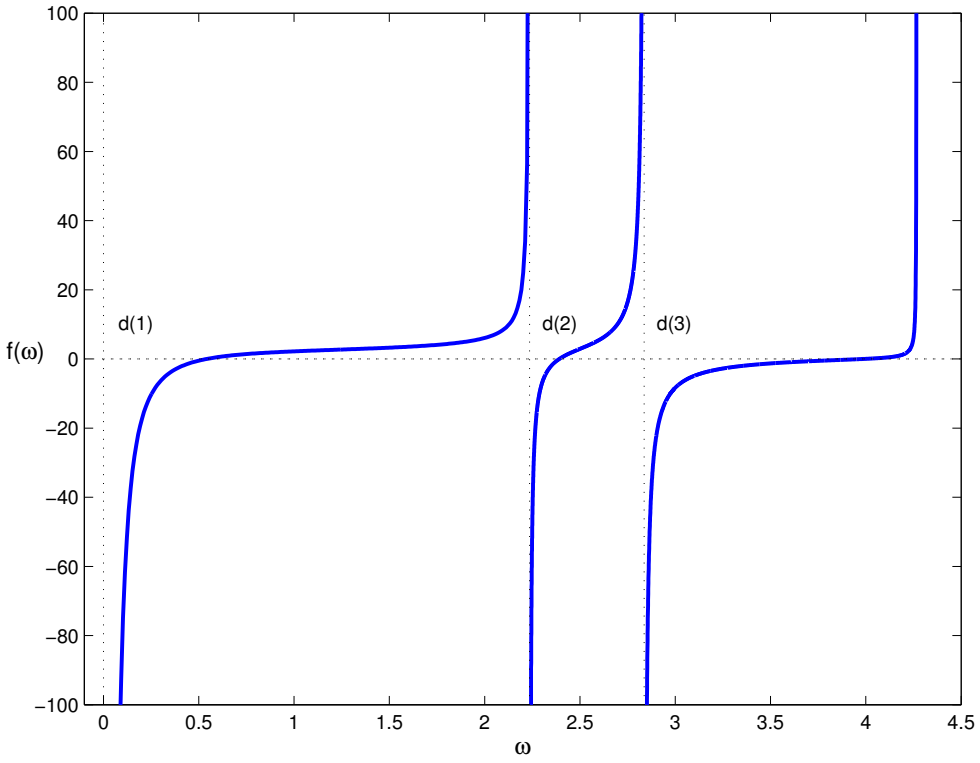

Figure 3.2: A Plot of the Secular Equation (3.9)

Suppose the case where $\omega_i \in \left(d_i, \frac{d_i+d_{i+1}}{2}\right)$ applies. Let $\delta_j = d_j - d_i$, and let

$$\psi_i(\mu) \equiv \sum_{j=1}^{i} \frac{z_j^2}{(\delta_j - \mu)(d_j + d_i + \mu)},$$

and

$$\phi_i(\mu) \equiv \sum_{j=i+1}^{n} \frac{z_j^2}{(\delta_j - \mu)(d_j + d_i + \mu)}.$$

Equation 3.9 can be now be reformulated as

$$g_i(\mu) \equiv f(\mu + d_i) = 1 + \psi_i(\mu) + \phi_i(\mu) = 0, \tag{3.10}$$

where the root in question is $\mu_i = \omega_i - d_i \in \left(0, \frac{\delta_{i+1}}{2}\right)$.

Now suppose the case where $\omega_i \in \left[\frac{d_i + d_{i+1}}{2}, d_{i+1}\right)$ applies. Let $\delta_j = d_j - d_{i+1}$,

$$\psi_i^+(\mu) \equiv \sum_{j=1}^{i} \frac{z_j^2}{(\delta_j - \mu)(d_j + d_{i+1} + \mu)},$$

and

$$\phi_i^+(\mu) \equiv \sum_{j=i+1}^{n} \frac{z_j^2}{(\delta_j - \mu)(d_j + d_{i+1} + \mu)}.$$

Equation 3.9 can now be reformulated as

$$g_i(\mu) \equiv f(\mu + d_{i+1}) = 1 + \psi_i^+(\mu) + \phi_i^+(\mu) = 0, \tag{3.11}$$

where the root in question is $\mu_i = \omega_i - d_{i+1} \in \left[\frac{\delta_i}{2}, 0\right)$.

Finally, for the case when $i = n$, let $\delta_j = d_j - d_n$. Let

$$\psi_n(\mu) \equiv \sum_{j=1}^{n} \frac{z_j^2}{(\delta_j - \mu)(d_j + d_n + \mu)},$$

and

$$\phi_n(\mu) \equiv 0.$$

The secular equation (3.9) can now be reformulated as

$$g_n(\mu) \equiv f(\mu + d_n) = 1 + \psi_n(\mu) + \phi_n(\mu) = 0, \tag{3.12}$$

where the root to be found is $\mu_n = \omega_n - d_n \in (0, \|z\|_2)$.

The roots to the reformulated secular equations (3.10), (3.11), and (3.12) are found using a method such as bisection or rational interpolation. The particular method is less important computationally than the stopping condition that is used [8]. Let η be a small empirically chosen multiple of ϵ_{mach}. The stopping condition proposed by Gu and Eisenstat [8] is

$$|g_i(\mu)| \leq \eta n(1 + |\psi_i(\mu)| + |\phi_i(\mu)|). \tag{3.13}$$

The roots $\omega_1, \cdots, \omega_n$ of the secular equation are the singular values of $\mathbf{M}$, and therefore of $\mathbf{B}$.

3.4 Computing the Singular Vectors

In theory, the singular values of $\mathbf{M}$ are computed to high relative accuracy [8] using (3.10), (3.11), and (3.12). In practice however, factors such as machine precision allow only an approximation of the exact singular values. Let $\hat{\omega}_1, \cdots, \hat{\omega}_n$ be the approximate singular values of $\mathbf{M}$, computed as described in Section 3.3. Then according to Gu and Eisenstat's Lemma 3.2 [8], there exists a matrix $\hat{\mathbf{M}}$ such that $\hat{\omega}_1, \cdots, \hat{\omega}_n$ are its exact singular values. In order to compute $\hat{\mathbf{M}}$, the values $\hat{\omega}_1, \cdots, \hat{\omega}_n$ must satisfy the interlacing property of (3.8). Gu and Eisenstat note that because the exact singular values of $\mathbf{M}$ satisfy (3.8), requiring the approximate singular values to also satisfy it is an accuracy requirement for computing the singular values, rather than a restriction on $\mathbf{M}$ [8]. $\hat{\mathbf{M}}$ has the form

$$\hat{\mathbf{M}} = \begin{pmatrix} \hat{z}_1 & & & \\ \hat{z}_2 & d_2 & & \\ \vdots & & \ddots & \\ \hat{z}_n & & & d_n \end{pmatrix}, \tag{3.14}$$

where the elements $d_2 \ldots d_n$ do not change, and each element of $\hat{\mathbf{z}}^T = (\hat{z}_1, \cdots, \hat{z}_n)$ is computed as proposed by Lemma 3.2 in [8]:

$$|\hat{z}_i| = \sqrt{(\hat{\omega}_n^2 - d_i^2) \prod_{k=1}^{i-1} \frac{\hat{\omega}_k^2 - d_i^2}{d_k^2 - d_i^2} \prod_{k=i}^{n-1} \frac{\hat{\omega}_k^2 - d_i^2}{d_{k+1}^2 - d_i^2}}.$$

The sign of each $\hat{z}_i$ can be chosen arbitrarily to match the sign of the corresponding z_i [8].

The exact singular values of $\hat{\mathbf{M}}$, $\hat{\omega}_1, \cdots, \hat{\omega}_n$, and vector $\hat{z}_i$ are used to compute

the exact singular vectors of $\hat{\mathbf{M}}$. The SVD of $\hat{\mathbf{M}}$ is then used as an approximation of the SVD of $\mathbf{M}$ to compute the singular vectors of $\mathbf{B}$ [8]. Denote the singular vectors of $\hat{\mathbf{M}}$ by $\hat{\mathbf{U}} = \hat{\mathbf{u}}_1, \cdots, \hat{\mathbf{u}}_n$ and $\hat{\mathbf{V}} = \hat{\mathbf{v}}_1, \cdots, \hat{\mathbf{v}}_n$. They are computed to high relative accuracy [8] using (3.15) and (3.16):

$$\hat{\mathbf{u}}_i = \left(\frac{\hat{z}_1}{d_1^2 - \hat{\omega}_i^2}, \cdots, \frac{\hat{z}_n}{d_n^2 - \hat{\omega}_i^2} \right)^T \Bigg/ \sqrt{\sum_{k=1}^{n} \frac{\hat{z}_k^2}{\left(d_k^2 - \hat{\omega}_i^2\right)^2}}, \tag{3.15}$$

$$\hat{\mathbf{v}}_i = \left(-1, \frac{d_2 \hat{z}_2}{d_2^2 - \hat{\omega}_i^2}, \cdots, \frac{d_n \hat{z}_n}{d_n^2 - \hat{\omega}_i^2} \right)^T \Bigg/ \sqrt{1 + \sum_{k=2}^{n} \frac{\left(d_k \hat{z}_k\right)^2}{\left(d_k^2 - \hat{\omega}_i^2\right)^2}}. \tag{3.16}$$

This gives

$$\hat{\mathbf{M}} = \hat{\mathbf{U}} \mathbf{\Omega} \hat{\mathbf{V}}^T \approx \mathbf{U} \mathbf{\Omega} \mathbf{V}^T = \mathbf{M}.$$

Using these components and components from the SVDs of the subproblems $\mathbf{B}_1$ and $\mathbf{B}_2$, the SVD of the bidiagonal matrix $\mathbf{B}$ of the original problem is formed:

$$\begin{aligned} \mathbf{B} &= \left(\mathbf{Q}\hat{\mathbf{U}}, \mathbf{q} \right) \begin{pmatrix} \mathbf{\Omega} \\ \mathbf{0} \end{pmatrix} \left(\mathbf{W}\hat{\mathbf{V}} \right)^T \\ &= \mathbf{U} \begin{pmatrix} \mathbf{\Omega} \\ \mathbf{0} \end{pmatrix} \mathbf{V}^T \\ &= \mathbf{U} \mathbf{\Sigma} \mathbf{V}^T, \end{aligned} \tag{3.17}$$

where $\mathbf{Q}$, $\mathbf{q}$, and $\mathbf{W}$ are as defined in Section 3.2.

3.5 Methods of Deflation

In the divide-and-conquer SVD algorithm, deflation serves three purposes. The first is to reduce $\mathbf{M}$ so that it satisfies conditions (3.6) and (3.7), the second is to maintain the block structure of matrices $\mathbf{Q}$ and $\mathbf{W}$ for efficiency purposes, and the third is to reduce the size of the problem and thus accelerate the computation of the SVD.

3.5.1 Deflation of Matrix M

As noted in Section 3.3, in order to satisfy the interlacing property (3.8) and the secular equation (3.9), it must be the case that $|z_j| \geq \tau\|\mathbf{M}\|_2$ and $|d_i - d_j| \geq \tau\|\mathbf{M}\|_2$ for $i \neq j$, where $\mathbf{z} = (z_1, \cdots, z_n)^T$ is the first column of $\mathbf{M}$, $d_1 = 0$ and $d_2, \ldots, d_n$ are elements $\mathbf{M}_{2,2}, \cdots, \mathbf{M}_{n,n}$, and τ is a small multiple of ϵ_{mach}.

First consider the condition that $|z_j| \geq \tau\|\mathbf{M}\|_2$. Suppose that $|z_1| < \tau\|\mathbf{M}\|_2$. Changing the value of $|z_1|$ to $\tau\|\mathbf{M}\|_2$ allows $|z_1|$ to meet the necessary condition while perturbing the matrix by $\mathcal{O}(\tau\|\mathbf{M}\|_2)$ [8].

$$\mathbf{M} = \begin{pmatrix} z_1 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{pmatrix} = \begin{pmatrix} \tau\|\mathbf{M}\|_2 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2).$$

For any subsequent z_i, where $1 < i \leq n$, if $|z_i| < \tau\|\mathbf{M}\|_2$, changing the value of the

element z_i to 0 also perturbs the matrix by $\mathcal{O}(\tau\|\mathbf{M}\|_2)$ [8].

$$\mathbf{M} = \begin{pmatrix} z_1 & & & & & \\ z_2 & d_2 & & & & \\ \vdots & & \ddots & & & \\ z_i & & & d_i & & \\ \vdots & & & & \ddots & \\ z_n & & & & & d_n \end{pmatrix} = \begin{pmatrix} z_1 & & & & & \\ z_2 & d_2 & & & & \\ \vdots & & \ddots & & & \\ 0 & & & d_i & & \\ \vdots & & & & \ddots & \\ z_n & & & & & d_n \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2).$$

The element d_i is now a singular value of the perturbed matrix. The matrix can be deflated by removing the ith row and the ith column, with d_i being stored as a singular value. The remaining $(n-1)\times(n-1)$ matrix has the same structure as $\mathbf{M}$.

Now consider the condition that $|d_i - d_j| \geq \tau\|\mathbf{M}\|_2$ for $i \neq j$. Suppose it is the case that $|d_i - d_1| = |d_i - 0| < \tau\|\mathbf{M}\|_2$, where $1 < i \leq n$. Changing the value of d_i to 0 and performing a Givens rotation from the left to zero out z_i creates the new matrix $\mathbf{GM}$ which is perturbed by $\mathcal{O}(\tau\|\mathbf{M}\|_2)$ [8].

$$
\begin{aligned}
\mathbf{GM} &= \begin{pmatrix} c & & & s & & \\ & 1 & & & & \\ & & \ddots & & & \\ -s & & & c & & \\ & & & & \ddots & \\ & & & & & 1 \end{pmatrix} \begin{pmatrix} z_1 & & & & & \\ z_2 & d_2 & & & & \\ \vdots & & \ddots & & & \\ z_i & & & 0 & & \\ \vdots & & & & \ddots & \\ z_n & & & & & d_n \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2) \\
&= \begin{pmatrix} r & & & & & \\ z_2 & d_2 & & & & \\ \vdots & & \ddots & & & \\ 0 & & & 0 & & \\ \vdots & & & & \ddots & \\ z_n & & & & & d_n \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2),
\end{aligned}
$$

where $r = \sqrt{z_i^2 + z_1^2}$, $s = z_i/r$, and $c = z_1/r$. In matrix $\mathbf{GM}$, $d_i = 0$ is a singular value [8]. The matrix can be deflated by removing the ith row and the i column, with d_i being stored as a singular value. The remaining $(n-1) \times (n-1)$ matrix has the same structure as $\mathbf{M}$.

Recall from (3.3) that $\mathbf{B} = (\mathbf{Q} \quad \mathbf{q}) \begin{pmatrix} \mathbf{M} \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T$. In order to maintain this equality, the Givens rotation $\mathbf{G}^T$ must be applied to $\mathbf{Q}$ from the right, where $\mathbf{G}^T$ is the transpose of the Givens rotation applied to $\mathbf{M}$ from the left when zeroing out element z_i. This gives

$$\mathbf{B} = \begin{pmatrix} \mathbf{Q} & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{M} \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T = \begin{pmatrix} \mathbf{Q}\mathbf{G}^T & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{G}\mathbf{M} \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T.$$

Now suppose it is the case that $|d_i - d_j| < \tau\|\mathbf{M}\|_2$, where $1 < i, j \leq n$, and $i \neq j$. Changing the value of d_j to the value of d_i and applying a Givens rotation $\mathbf{G}$ from the left and from the right to zero out z_i results in the new matrix $\mathbf{G}\mathbf{M}\mathbf{G}^T$, which is perturbed by $\mathcal{O}(\tau\|\mathbf{M}\|_2)$ [8]. For example, let $i = 3$ and $j = 2$. Then

$$\begin{aligned} \mathbf{G}\mathbf{M}\mathbf{G}^T &= \begin{pmatrix} 1 & & \\ & c & s \\ & -s & c \end{pmatrix} \begin{pmatrix} z_1 & & \\ z_2 & d_3 & \\ z_3 & & d_3 \end{pmatrix} \begin{pmatrix} 1 & & \\ & c & -s \\ & s & c \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2) \\ &= \begin{pmatrix} z_1 & & \\ z_2 & d_3 & \\ 0 & & d_3 \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2), \end{aligned}$$

where $r = \sqrt{z_i^2 + z_j^2}$, $s = z_i/r$, and $c = z_j/r$. In matrix $\mathbf{G}\mathbf{M}$, d_i is a singular value [8]. The matrix can be deflated by removing the ith row and the ith column, with d_i being stored as a singular value. The remaining $(n-1) \times (n-1)$ matrix has the same structure as $\mathbf{M}$.

As in the previous case, the transpose $\mathbf{G}^T$ must be applied to the right of $\mathbf{Q}$, where $\mathbf{G}$ and $\mathbf{G}^T$ are the Givens rotations applied to the left and right of $\binom{\mathbf{M}}{\mathbf{0}}$ respectively when zeroing out element $z_i \in \mathbf{M}$. In addition, $\mathbf{G}$ must be applied to the left of $\mathbf{W}^T$, or for ease of notation, $\mathbf{G}^T$ is applied to the right of $\mathbf{W}$. This gives

$$\mathbf{B} = \begin{pmatrix} \mathbf{Q} & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{M} \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T = \begin{pmatrix} \mathbf{Q}\mathbf{G}^T & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{G}\mathbf{M}\mathbf{G}^T \\ \mathbf{0} \end{pmatrix} \left(\mathbf{W}\mathbf{G}^T\right)^T. \quad (3.18)$$

The deflation of $\mathbf{M}$ is necessary to ensure that the matrix satisfies the interlacing property and the secular equation. Two other forms of deflation, known as *local* and *global deflation* [8], are performed for the sake of efficiency, to accelerate the speed of computation. These methods are discussed in the following two subsections.

3.5.2 Local Deflation

The greatest computational cost in the divide-and-conquer SVD algorithm arises from the multiplication of matrices [8]. When the singular vectors of $\mathbf{B}$ are formed in (3.17), the products $\mathbf{QU}$ and $\mathbf{WV}$ must be computed. This computation can be made more efficient by utilizing the fact that $\mathbf{Q}$ and $\mathbf{W}$ contain *zero blocks*. This simply means that they each have blocks of elements that are zero. The local deflation procedure ensures that the structure of these blocks is maintained, so that efficient multiplication of the matrices can take place. According to Gu and Eisenstat, the local deflation procedure speeds up the divide-and-conquer SVD algorithm by a factor of two [8].

As noted in the previous deflation method, when a Givens rotation is applied to the left of $\mathbf{M}$, its transpose must be applied to the right of $\mathbf{Q}$. Likewise, when a Givens rotation is applied to the right of $\mathbf{M}$, its transpose must be applied to the left of $\mathbf{W}$. In most cases, this does not affect the block structure of $(\mathbf{Q} \quad \mathbf{q})$ and $\mathbf{W}$; however there is one case where the block structure is compromised. Local deflation deals with this case.

As illustrated in Sections 3.2 and 3.3, $\mathbf{M}$ is of the form

$$\mathbf{M} = \begin{pmatrix} z_1 & & & \\ z_2 & d_2 & & \\ \vdots & & \ddots & \\ z_n & & & d_n \end{pmatrix} = \begin{pmatrix} \alpha_k \lambda_1 & \mathbf{0} & \mathbf{0} \\ \alpha_k \mathbf{l}_1 & \mathbf{D}_1 & \mathbf{0} \\ \beta_k \mathbf{f}_2 & \mathbf{0} & \mathbf{D}_2 \end{pmatrix}.$$

If it is the case that an element $d_i \in \mathbf{D}_1$ is close in value to an element $d_j \in \mathbf{D}_2$ such that $|d_i - d_j| < \tau \|\mathbf{M}\|_2$, then when the deflation process takes place, the Givens rotations applied to $\mathbf{Q}$ and $\mathbf{W}$ disturb the block structure of these matrices. This prevents efficient matrix multiplication when the singular vectors of $\mathbf{B}$ are computed. The following, based on an example from Gu and Eisenstat [8], illustrates this case.

Suppose that d_2, the first element of $\mathbf{D}_1$, is close in value to d_n, the last element of $\mathbf{D}_2$; i.e., $|d_n - d_2| < \tau \|\mathbf{M}\|_2$. Then, as described in the previous subsection, d_2 is given the value of d_n, and a Givens rotation $\mathbf{G}$ is applied to $\mathbf{M}$, zeroing out z_n. Let

$$\mathbf{Q}_1 = \begin{pmatrix} \tilde{\mathbf{q}}_1 & \tilde{\mathbf{Q}}_1 \end{pmatrix}, \mathbf{Q}_2 = \begin{pmatrix} \tilde{\mathbf{Q}}_2 & \tilde{\mathbf{q}}_2 \end{pmatrix}, \mathbf{W}_1 = \begin{pmatrix} \tilde{\mathbf{w}}_1 & \tilde{\mathbf{W}}_1 \end{pmatrix}, \text{and} \quad \mathbf{W}_2 = \begin{pmatrix} \tilde{\mathbf{W}}_2 & \tilde{\mathbf{w}}_2 \end{pmatrix}$$

such that

$$\mathbf{B} = \begin{pmatrix} \mathbf{Q} & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{M} \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T = \left(\begin{pmatrix} c_0 \mathbf{q}_1 & \tilde{\mathbf{q}}_1 & \tilde{\mathbf{Q}}_1 & \mathbf{0} & \mathbf{0} \\ s_0 \mathbf{q}_2 & \mathbf{0} & \mathbf{0} & \tilde{\mathbf{Q}}_2 & \tilde{\mathbf{q}}_2 \end{pmatrix} \mathbf{q} \right) \begin{pmatrix} \mathbf{G}\mathbf{M}\mathbf{G}^T \\ \mathbf{0} \end{pmatrix} \mathbf{W}^T,$$

with c_0, s_0 as defined for (3.2) and $\mathbf{q}$ as defined in (3.3) . Also let

$$\mathbf{D}_1 = \text{diag}\left(d_2, \tilde{\mathbf{D}}_1\right), \mathbf{D}_2 = \text{diag}\left(\tilde{\mathbf{D}}_2, d_n\right), \alpha_k l_1 = \begin{pmatrix} z_2 \\ \tilde{z}_1 \end{pmatrix}, \text{and} \quad \beta_k f_2 = \begin{pmatrix} \tilde{z}_2 \\ z_n \end{pmatrix}.$$

Then

$$\mathbf{M} = \begin{pmatrix} \alpha_k\lambda_1 & \mathbf{0} & \mathbf{0} \\ \alpha_k\mathbf{l}_1 & \mathbf{D}_1 & \mathbf{0} \\ \beta_k\mathbf{f}_2 & \mathbf{0} & \mathbf{D}_2 \end{pmatrix} = \begin{pmatrix} z_1 & & & & \\ z_2 & d_2 & & & \\ \tilde{\mathbf{z}}_1 & & \tilde{\mathbf{D}}_1 & & \\ \tilde{\mathbf{z}}_2 & & & \tilde{\mathbf{D}}_2 & \\ z_n & & & & d_n \end{pmatrix}.$$

Assigning the value of d_n to d_2 and applying the Givens rotations $\mathbf{G}$ and $\mathbf{G}^T$ to the left and right of $\mathbf{M}$, respectively, gives

$$\mathbf{G}\mathbf{M}\mathbf{G}^T = \begin{pmatrix} r_0 & & & & \\ r & d_n & & & \\ \tilde{z}_1 & & \tilde{\mathbf{D}}_1 & & \\ \tilde{z}_2 & & & \tilde{\mathbf{D}}_2 & \\ 0 & & & & d_n \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2) \equiv \begin{pmatrix} \tilde{\mathbf{M}}_1 & \mathbf{0} \\ \mathbf{0} & d_n \end{pmatrix} + \mathcal{O}(\tau\|\mathbf{M}\|_2),$$

where $r = \sqrt{z_2^2 + z_n^2}$, $c = z_2/r$, and $s = z_n/r$. Substituting this into (3.18) gives

$$
\begin{aligned}
\mathbf{B} &= \begin{pmatrix} \mathbf{Q}\mathbf{G}^T & \mathbf{q} \end{pmatrix} \begin{pmatrix} \mathbf{G}\mathbf{M}\mathbf{G}^T \\ \mathbf{0} \end{pmatrix} \left(\mathbf{W}\mathbf{G}^T\right)^T \\
&= \begin{pmatrix} \tilde{\mathbf{X}}_1 & \tilde{\mathbf{x}} & \mathbf{q} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{M}}_1 & \mathbf{0} \\ \mathbf{0} & d_n \\ \mathbf{0} & 0 \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{Y}}_1 & \tilde{\mathbf{y}} \end{pmatrix}^T + \mathcal{O}(\tau\|\mathbf{M}\|_2),
\end{aligned}
$$

where

$$
\tilde{\mathbf{X}}_1 = \begin{pmatrix} c_0\mathbf{q}_1 & c\tilde{\mathbf{q}}_1 & \tilde{\mathbf{Q}}_1 & \mathbf{0} \\ s_0\mathbf{q}_2 & s\tilde{\mathbf{q}}_2 & \mathbf{0} & \tilde{\mathbf{Q}}_2 \end{pmatrix} \qquad \text{and} \qquad \tilde{\mathbf{x}} = \begin{pmatrix} -s\tilde{\mathbf{q}}_1 \\ c\tilde{\mathbf{q}}_2 \end{pmatrix}
$$

and

$$
\tilde{\mathbf{Y}}_1 = \begin{pmatrix} \mathbf{0} & c\tilde{\mathbf{w}}_1 & \tilde{\mathbf{W}}_1 & \mathbf{0} \\ 1 & 0 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & s\tilde{\mathbf{w}}_2 & \mathbf{0} & \tilde{\mathbf{W}}_2 \end{pmatrix} \qquad \text{and} \qquad \tilde{\mathbf{y}} = \begin{pmatrix} -s\tilde{\mathbf{w}}_1 \\ 0 \\ c\tilde{\mathbf{w}}_2 \end{pmatrix}.
$$

Thus, d_n is an approximate singular value of $\mathbf{B}$ and can therefore be deflated [8]. The left and right singular vectors of d_n are approximated by $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ respectively. The structure of the matrix $\tilde{\mathbf{M}}$ is the same as that of $\mathbf{M}$, and so the process of local deflation can continue until there is no case left where $|d_i - d_j| < \tau\|\mathbf{M}\|_2$ for $i \neq j$. Completing the process of local deflation gives

$$\mathbf{B} = \begin{pmatrix} \tilde{\mathbf{X}}_1 & \tilde{\mathbf{X}}_2 & \mathbf{q} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{M}}_1 & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{\Omega}}_2 \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{Y}}_1 & \tilde{\mathbf{Y}}_2 \end{pmatrix}^T + \mathcal{O}(\tau \|\mathbf{B}\|_2), \tag{3.19}$$

where permutations of the diagonal entries of the $\mathbf{D}_i$ and of the columns of the $\mathbf{Q}_i$ and $\mathbf{W}_i$ are ignored [8]. $\tilde{\mathbf{X}}_1$ and $\tilde{\mathbf{X}}_2$ have the form

$$\tilde{\mathbf{X}}_1 = \begin{pmatrix} c_0\mathbf{q}_1 & c\tilde{\mathbf{Q}}_{0,1} & \tilde{\mathbf{Q}}_1 & \mathbf{0} \\ s_0\mathbf{q}_2 & s\tilde{\mathbf{Q}}_{0,2} & \mathbf{0} & \tilde{\mathbf{Q}}_2 \end{pmatrix} \quad \text{and} \quad \tilde{\mathbf{Y}}_1 = \begin{pmatrix} \mathbf{0} & c\tilde{\mathbf{W}}_{0,1} & \tilde{\mathbf{W}}_1 & \mathbf{0} \\ 1 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & s\tilde{\mathbf{W}}_{0,2} & \mathbf{0} & \tilde{\mathbf{W}}_2 \end{pmatrix}.$$

The column dimension of $\tilde{\mathbf{Q}}_{0,1}$, $\tilde{\mathbf{Q}}_{0,2}$, $\tilde{\mathbf{W}}_{0,1}$, and $\tilde{\mathbf{W}}_{0,2}$ is the number of local deflations. The columns of $\tilde{\mathbf{Q}}_1$, $\tilde{\mathbf{Q}}_2$, $\tilde{\mathbf{W}}_1$, and $\tilde{\mathbf{W}}_2$ are the columns of $\mathbf{Q}_1$, $\mathbf{Q}_2$, $\mathbf{W}_1$, and $\mathbf{W}_2$ that have not been affected by the local deflation. $\tilde{\mathbf{\Omega}}_2$ is a diagonal matrix containing the deflated singular values on its diagonal, with the columns of $\tilde{\mathbf{Y}}_1$ and $\tilde{\mathbf{Y}}_2$ containing the corresponding left and right singular vectors respectively [8]. $\tilde{\mathbf{M}}_1$ has the form

$$\tilde{\mathbf{M}}_1 = \begin{pmatrix} r_0 & & & \\ \tilde{\mathbf{z}}_0 & \tilde{\mathbf{D}}_0 & & \\ \tilde{\mathbf{z}}_1 & & \tilde{\mathbf{D}}_1 & \\ \tilde{\mathbf{z}}_2 & & & \tilde{\mathbf{D}}_2 \end{pmatrix}.$$

The dimension of $\tilde{\mathbf{D}}_0$ is the number of deflations performed by the local deflation

process. Submatrices $\tilde{\mathbf{D}}_1$ and $\tilde{\mathbf{D}}_2$ contain the elements of $\mathbf{D}_1$ and $\mathbf{D}_2$ that have not been affected by local deflation, with $\tilde{\mathbf{z}}_0$, $\tilde{\mathbf{z}}_1$, and $\tilde{\mathbf{z}}_2$ defined accordingly [8].

Let $\tilde{\mathbf{U}}_1\tilde{\mathbf{\Sigma}}_1\tilde{\mathbf{V}}_1^T$ be the SVD of $\tilde{\mathbf{M}}_1$. Substituting this into (3.19) gives

$$\begin{aligned}
\mathbf{B} &= \begin{pmatrix} \tilde{\mathbf{X}}_1 & \tilde{\mathbf{X}}_2 & \mathbf{q} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{U}}_1\tilde{\mathbf{\Sigma}}_1\tilde{\mathbf{V}}_1^T & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{\Omega}}_2 \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{Y}}_1 & \tilde{\mathbf{Y}}_2 \end{pmatrix}^T + \mathcal{O}(\tau\|\mathbf{B}\|_2) \\
&= \begin{pmatrix} \tilde{\mathbf{X}}_1\tilde{\mathbf{U}}_1 & \tilde{\mathbf{X}}_2 & \mathbf{q} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{\Omega}}_1 & \mathbf{0} \\ \mathbf{0} & \tilde{\mathbf{\Omega}}_2 \\ \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{Y}}_1\tilde{\mathbf{V}}_1 & \tilde{\mathbf{Y}}_2 \end{pmatrix}^T + \mathcal{O}(\tau\|\mathbf{B}\|_2).
\end{aligned}$$

The approximate left and right singular vectors of $\mathbf{B}$ are then $\begin{pmatrix} \tilde{\mathbf{X}}_1\tilde{\mathbf{U}}_1 & \tilde{\mathbf{X}}_2 & \mathbf{q} \end{pmatrix}$ and $\begin{pmatrix} \tilde{\mathbf{Y}}_1\tilde{\mathbf{V}}_1 & \tilde{\mathbf{Y}}_2 \end{pmatrix}$ respectively. Local deflation preserves the block structure of $\tilde{\mathbf{X}}_1$ and $\tilde{\mathbf{Y}}_1$. This allows the efficient computation of the products $\tilde{\mathbf{X}}_1\tilde{\mathbf{U}}_1$ and $\tilde{\mathbf{Y}}_1\tilde{\mathbf{V}}_1$.

3.5.3 Global Deflation

Whereas local deflation is applied to subproblems when their SVDs are being combined for the computation of the SVD of $\mathbf{B}$, global deflation deflates a subproblem before its SVD is even computed. The singular values and corresponding singular vectors that are deflated by the global deflation process are then absent from subsequent subproblems [8], reducing the dimension of the problems.

Recall from Section 3.1 that the matrix $\mathbf{B} \in \Re^{(n+1)\times n}$ is divided into submatrices $\mathbf{B}_1 \in \Re^{k\times(k-1)}$ and $\mathbf{B}_2 \in \Re^{(n-k+1)\times(n-k)}$, and scalars α_k and β_k, where $k = \lfloor \frac{n}{2} \rfloor$.

Global deflation subdivides $\mathbf{B}_1$ in a similar manner.

$$\mathbf{B} = \begin{pmatrix} \mathbf{B}_1 & \alpha_{i+j}\mathbf{e}_{i+j} & \\ & \beta_{i+j}\mathbf{e}_1 & \mathbf{B}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{B}_{1,1} & \alpha_i\mathbf{e}_i & & \\ & \beta_i\mathbf{e}_1 & \mathbf{B}_{1,2} & \alpha_{i+j}\mathbf{e}_j \\ & & & \beta_{i+j}\mathbf{e}_1 & \mathbf{B}_2 \end{pmatrix}.$$

Here, $\mathbf{B}_{1,1} \in \Re^{i\times(i-1)}$ and $\mathbf{B}_{1,2} \in \Re^{j\times(j-1)}$, where $i+j=k$, and $j = \lfloor \frac{k}{2} \rfloor$. As before, $\mathbf{e}_j$ is the jth unit vector of the appropriate size.

Let $\mathbf{X}_{1,2}\begin{pmatrix}\mathbf{D}_{1,2}\\ \mathbf{0}\end{pmatrix}\mathbf{W}_{1,2}^T$ be the SVD of $\mathbf{B}_{1,2}$. Let the first row of $\mathbf{X}_{1,2}$ be $\begin{pmatrix}\mathbf{f}_{1,2}^T & \varphi_{1,2}\end{pmatrix}$, and the last row be $\begin{pmatrix}\mathbf{l}_{1,2}^T & \lambda_{1,2}\end{pmatrix}$. Then

$$\mathbf{B} = \bar{\mathbf{X}} \begin{pmatrix} \mathbf{B}_{1,1} & \alpha_i\mathbf{e}_i & & \\ & \beta_i\mathbf{f}_{1,2} & \mathbf{D}_{1,2} & \alpha_{i+j}\mathbf{l}_{1,2} \\ & \beta_i\varphi_{1,2} & \mathbf{0} & \alpha_{i+j}\lambda_{1,2} \\ & & & \beta_{i+j}\mathbf{e}_1 & \mathbf{B}_2 \end{pmatrix} \bar{\mathbf{Y}}^T, \tag{3.20}$$

where $\bar{\mathbf{X}} = \text{diag}\begin{pmatrix}\mathbf{I}_i & \mathbf{X}_{1,2} & \mathbf{I}_{n-i-j+1}\end{pmatrix}$ and $\bar{\mathbf{Y}} = \text{diag}\begin{pmatrix}\mathbf{I}_{i-1} & \mathbf{W}_{1,2} & 1 & \mathbf{I}_{n-i-j}\end{pmatrix}$.

The global deflation takes place as follows. Let $\bar{d}_s$ be the sth diagonal element of $\mathbf{D}_{1,2}$, let $\bar{f}_s$ be the sth element of $\mathbf{f}_{1,2}$, and let $\bar{l}_s$ be the sth element of $\mathbf{l}_{1,2}$. Ignore the zero elements of the middle matrix of (3.20). The $(i+s)$th column is then $\begin{pmatrix}\bar{\mathbf{d}}_s\end{pmatrix}$ and the $(i+s)$th row is $\begin{pmatrix}\beta_i\bar{f}_s & \bar{\mathbf{d}}_s & \alpha_{i+j}\bar{l}_s\end{pmatrix}$. Suppose $|\beta_i\bar{f}_s|$ and $|\alpha_{i+j}\bar{l}_s|$ are both *small*, for example $|\beta_i\bar{f}_s|, |\alpha_{i+j}\bar{l}_s| < \tau\|\mathbf{B_1}\|_2$. Both $|\beta_i\bar{f}_s|$ and $|\alpha_{i+j}\bar{l}_s|$ can then be perturbed to zero. $\bar{d}_s$ is a singular value of the perturbed matrix, and the $(i+j)$th columns of $\bar{\mathbf{X}}$ and $\bar{\mathbf{Y}}$ are its corresponding left and right singular vectors [8]. $\bar{d}_s$ and these singular vectors can then be deflated from all subsequent subproblems, globally reducing the dimension of the problem to be solved.

Chapter 4

Results

The implementions and tests discussed in this thesis use the *Matlab* problem solving environment. Tests are run using *Matlab* Release 13 on an Ultra3 SunFire V880 (Solaris 8 operating system) with 4 CPUs and a total of 8 GB shared memory.

As described in Section 1.3, the matrices used in LSI are very sparse. Searching the Text Retrieval Conference collection, for example, results in term-by-document matrices with a density of only $0.001-0.002\%$ [3, 4]. This means that $99.998-99.999\%$ of the elements in the matrices are 0. On average, the density of a term-by-document matrix is approximately 1% [1, 4]; therefore the testing described here uses matrices with approximately 1% nonzero elements. These matrices are generated using the built-in *Matlab* function `sprand`. The function call `sprand(m, n, density)` produces a random $m \times n$ sparse matrix with approximately $density * m * n$ nonzero entries uniformly distributed on the interval [0,1].

The sparse test matrices are first bidiagonalized using an implementation of the Lanczos algorithm [6, 7] (Algorithm 2.1) based on code written by Sarah MacKinnon-Cormier [11]. The SVDs of the resulting $(n+1) \times n$ matrices are then computed.

The divide-and-conquer implementation uses a base case of $n = 15$, as discussed in Section 3.1.

The following example illustrates this process. For the purpose of this example, `m` = `n` = 9, `density` = 10% and a base case of `n` = 4 is used for the divide-and-conquer implementation. Note that to simplify this example, matrix elements are shown rounded to 2 decimal places, and the singular vectors are not displayed.

Example 4.1:

$$\mathbf{A} = \texttt{sprand(m, n, density)}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0.96 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.07 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.51 & 0 & 0.7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.62 & 0 & 0 & 0 & 0 \\ 0.23 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.63 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.15 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\mathbf{B} \;=\; \texttt{lanczos}(\mathbf{A})$$

$$= \begin{pmatrix}
0.60 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.74 & 0.34 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.56 & 0.40 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0.56 & 0.66 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.43 & 0.16 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0.25 & 0.17 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.07 & 0.40 * 10^{-12} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.97 & 0.06 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.81 & .40 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .80
\end{pmatrix}$$

The SVD of $\mathbf{B}$ is then computed. For this example only the singular values from the divide-and-conquer implementation and the built-in *Matlab* SVD function are compared. The following commands are used in *Matlab*, where `dc_svd` is the divide-conquer implementation and `svd` is the built-in *Matlab* function:

$[\mathbf{U1}, \mathbf{S1}, \mathbf{V1}] = \texttt{dc\_svd}(\mathbf{B});$

$[\mathbf{U2}, \mathbf{S2}, \mathbf{V2}] = \texttt{svd}(\mathbf{B});$

$\mathbf{s1} = \texttt{diag}(\mathbf{S1});$

$\mathbf{s2} = \texttt{diag}(\mathbf{S2});$

The singular values contained in **s1** and **s2** are as follows:

$$\mathbf{s1} = \begin{pmatrix} 1.02747756077280 \\ 1.02715741993593 \\ 0.96465307783203 \\ 0.96459969118453 \\ 0.63067714200223 \\ 0.62953517851245 \\ 0.34051401537432 \\ 0.22749403687719 \\ 0.06576695330954 \end{pmatrix}, \qquad \mathbf{s2} = \begin{pmatrix} 1.02747756077280 \\ 1.02715741993593 \\ 0.96465307783203 \\ 0.96459969118453 \\ 0.63067714200223 \\ 0.62953517851245 \\ 0.34051401537432 \\ 0.22749403687719 \\ 0.06576695330954 \end{pmatrix}.$$

In this case, the divide-and-conquer SVD implementation and the built-in *Matlab* SVD function produce the same singular values to at least 15 digits of accuracy. This, however, is not always the case. In this example, the singular values range from approximately 0.0658 to 1.0275. If there is a much greater spread between the singular values, then this implementation of the divide-and-conquer algorithm loses accuracy. The following example illustrates this behaviour.

Example 4.2:

In this example, the first two diagonal elements of the bidiagonalized matrix $\mathbf{B}$ from Example 4.1 are increased by a factor of 10^5, giving the perturbed matrix $\hat{\mathbf{B}}$.

$$\hat{\mathbf{B}} = \begin{pmatrix} 0.60 * 10^5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.74 & 0.34 * 10^5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.56 & 0.40 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.56 & 0.66 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.43 & 0.16 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.25 & 0.17 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.07 & 0.40 * 10^{-12} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.97 & 0.06 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.81 & .40 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .80 \end{pmatrix}$$

Computing the singular values of $\hat{\mathbf{B}}$ gives

$$\hat{\mathbf{s}}\mathbf{1} = 10^4 * \begin{pmatrix} 6.03284973308697 \\ 3.35336075134385 \\ 0.00010271574199 \\ 0.00009674182117 \\ 0.00009645996912 \\ 0.00006295351785 \\ 0.00004326121052 \\ 0.00002999189629 \\ 0.00000838082131 \end{pmatrix}, \qquad \hat{\mathbf{s}}\mathbf{2} = 10^4 * \begin{pmatrix} 6.03284973308696 \\ 3.35336075134383 \\ 0.00010271574199 \\ 0.00009674412461 \\ 0.00009645996912 \\ 0.00006295351785 \\ 0.00004326148087 \\ 0.00002999190992 \\ 0.00000838082134 \end{pmatrix},$$

where the elements of $\hat{\mathbf{s}}\mathbf{1}$ and $\hat{\mathbf{s}}\mathbf{2}$ are the singular values produced by the divide-and-conquer implementation and the *Matlab* SVD function respectively. In this case, the accuracy of the singular values produced by the divide-and-conquer implementation range in accuracy from 9 to 15 digits. The introduction of scaling methods could increase the accuracy of this particular divide-and-conquer implementation.

In order to compare the running times of the SVD methods discussed, two sets of trials were conducted. In the first set of trials, CPU times are compared for three SVD methods: a Golub-Kahan implementation coded by Sarah MacKinnon-Cormier [11], the divide-and-conquer implementation previously discussed, and the built-in *Matlab* SVD function. For this first set of trials, $n = 50, 100, 150, \ldots, 750$. The results shown in Figures 4.1 and 4.2 are the fastest CPU times for a minimum of five different trials. Figure 4.1 is a close-up view in which the difference in CPU times between the divide-and-conquer implementation and the *Matlab* SVD is more evident

than in the wider view of Figure 4.2. The Golub-Kahan implementation [11] is not tested on matrices with $n > 750$ because of the huge disparity between its running time and the running times of the other two methods tested. It is important to note that the Golub-Kahan implementation and the divide-and-conquer implementation are both unoptimized code. Thus, the dramatically faster CPU times for the divide-and-conquer implementation illustrated in Figures 4.1 and 4.2 are evidence of the efficiency of the divide-and-conquer SVD algorithm.

In the second set of trials, only the CPU Times for the divide-and-conquer implementation and the built-in *Matlab* SVD function are compared. The trials in this second set are for $n = 750, 1000, 1250 \ldots, 4500, 5000$. The results shown in Figure 4.3 are the fastest CPU times for a minimum of three different trials. The figure illustrates that even though the *Matlab* SVD function is highly optimized code, the divide-and-conquer implementation is faster for $1500 \leq n \leq 3750$. Although the results shown for $n > 3750$ are unexpected, the CPU times for the divide-and-conquer implementation are within a factor of 1.15 of those for the *Matlab* function.

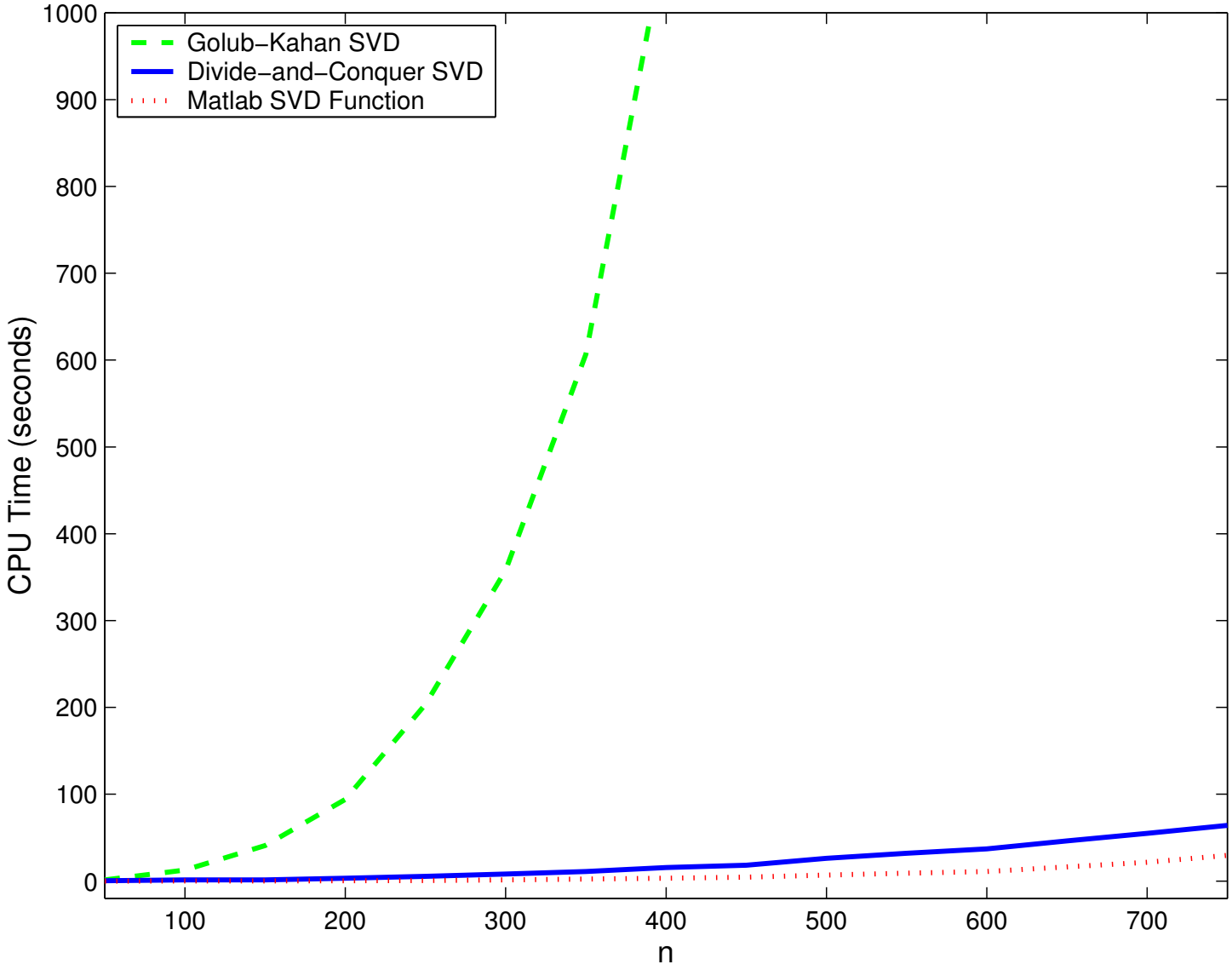

Figure 4.1: A Comparison of CPU Times for Computing an SVD ($50 \leq n \leq 750$), Close-up View

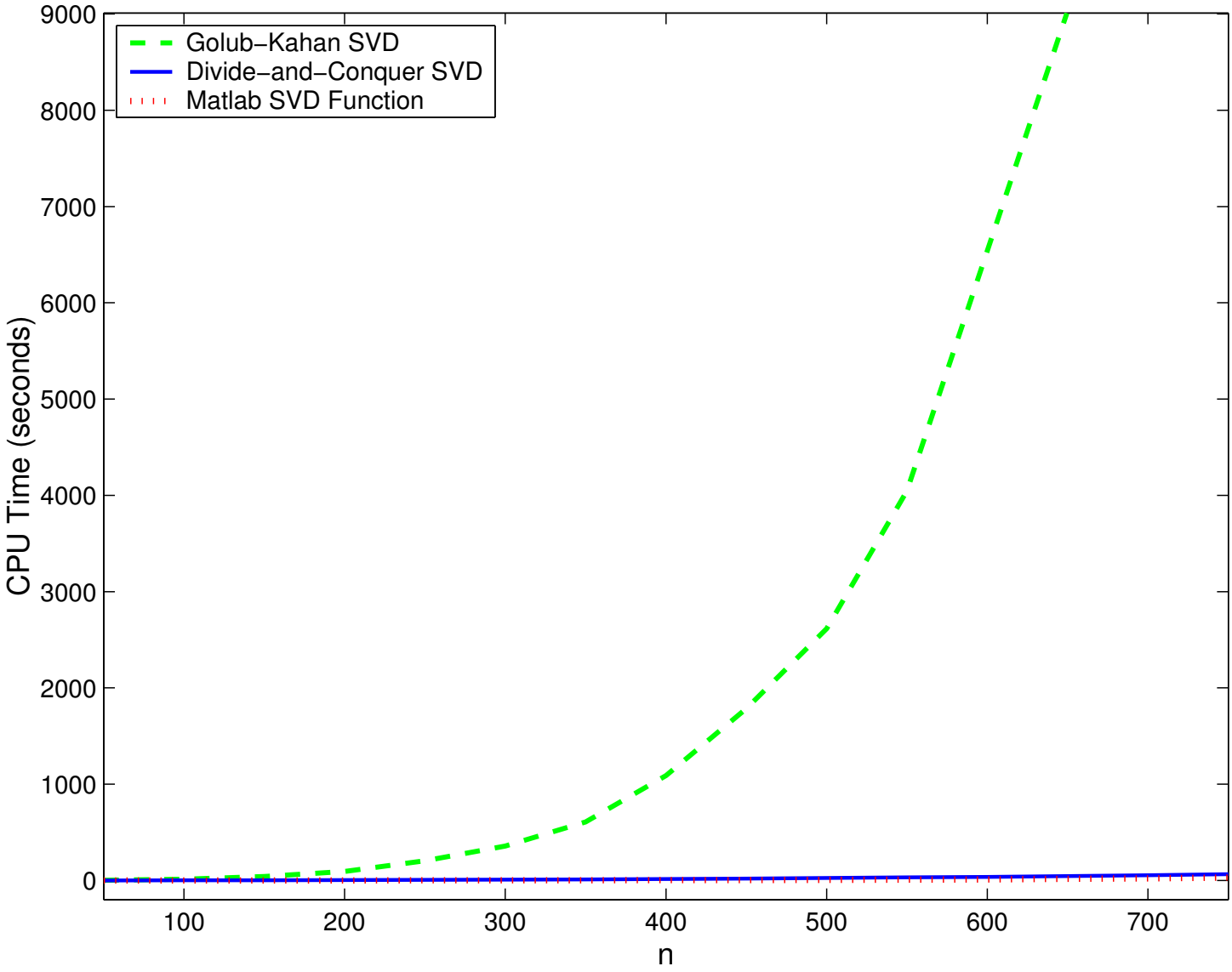

Figure 4.2: A Comparison of CPU Times for Computing an SVD ($50 \leq n \leq 750$), Wide View

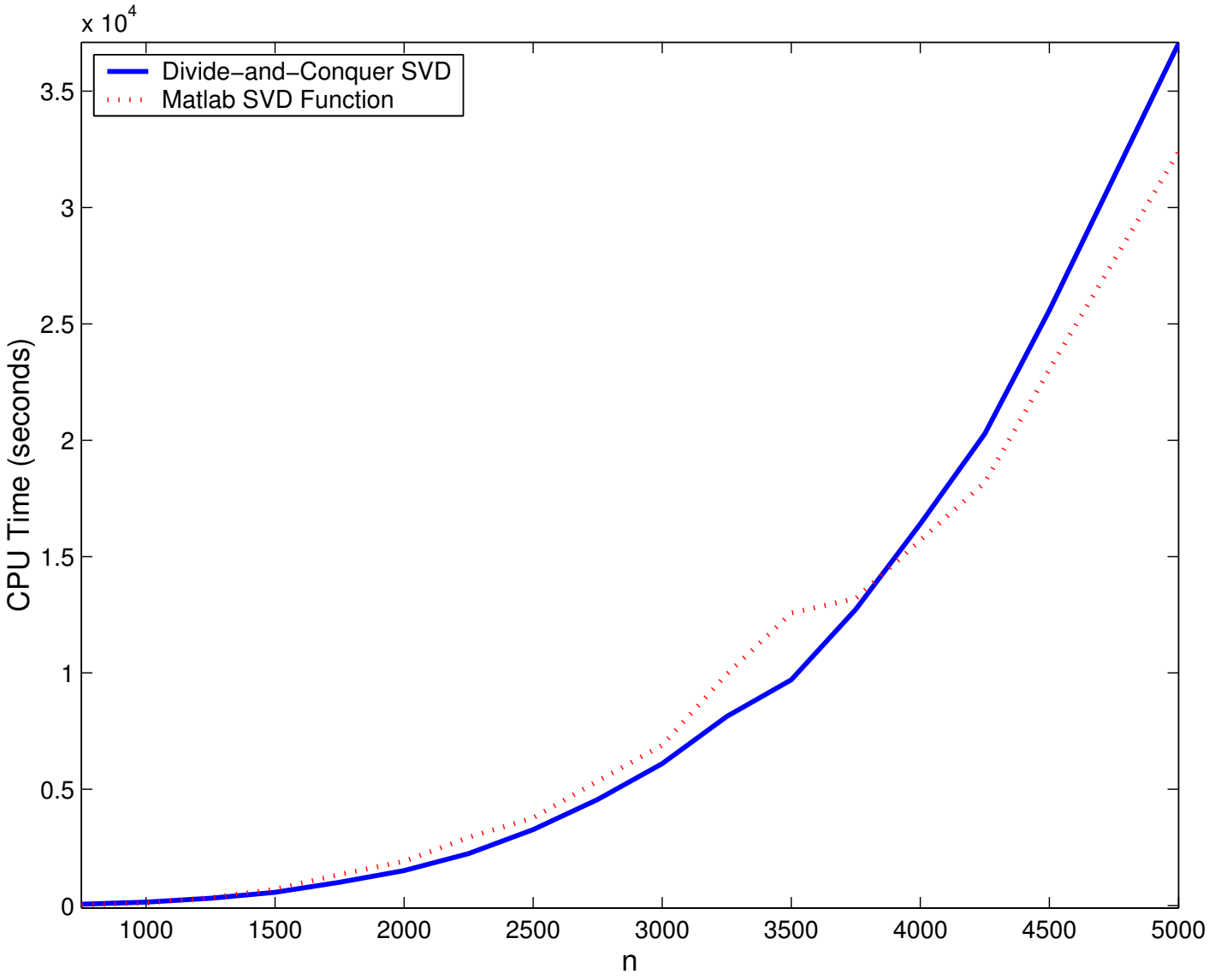

Figure 4.3: A Comparison of CPU Times for Computing an SVD ($750 \leq n \leq 5000$)

Chapter 5

Conclusions and Future Work

The size of the Internet and of the databases associated with it continues to expand, making the efficient retrieval of information an area of continuing interest. It was the goal of this thesis to illustrate that a divide-and-conquer approach is an efficient method of computing the SVD of the term-by-document matrices formed by the LSI search method. Results indicate that when compared with a similarly coded traditional Golub-Kahan method, the divide-and-conquer method is much more efficient. For example, for a 501×500 bidiagonal matrix, the divide-and-conquer implementation is faster than the Golub-Kahan implementation by a factor of 95, and this factor increases as the size of the matrices tested increases (see Figure 4.1). As already noted, the term-by-document matrices formed during the LSI process are very large, and the SVD needs to be computed relatively often. The divide-and-conquer approach thus offers potentially tremendous savings in computational time.

It is interesting to note that the divide-and-conquer implementation outperforms the built-in *Matlab* SVD function for $(n+1) \times n$ bidiagonal matrices with $1500 \leq n \leq 3750$. This is despite the fact that in all tests the divide-and-conquer implementation

is unoptimized and runs as interpreted code, whereas the built-in SVD function is highly optimized and runs as compiled code. The results are thus an indication of the computational efficiency of the divide-and-conquer approach. It is possible that the divide-and-conquer implementation would again outperform the *Matlab* SVD function when the running times are compared for matrices with larger n than is tested here; however it was not feasible to run such trials for this thesis.

Even using the divide-and-conquer algorithm, calculating the SVD is still a relatively expensive process. When terms or documents are added to, or deleted from, a term-by-document matrix, it makes sense to *update* or *downdate* the SVD that has been calculated for that matrix, rather than going to the expense of calculating the SVD of the modified matrix from scratch. Updating and downdating the SVD are areas not addressed by this thesis, but future work could include investigating an efficient means of updating and downdating the SVD of a matrix. These functions can then be added to the existing divide-and-conquer implementation. The running time of the resulting implementation can then be compared with that of traditional implementations, using large data sets representative of those used in LSI.

Bibliography

[1] M. W. Berry and M. Browne. *Understanding Search Engines: Mathematical Modeling and Text Retrieval.* Software, Environments, and Tools. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, first edition, 1999.

[2] M. W. Berry, Z. Drmač, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Rev.*, 41(2):335–362 (electronic), 1999.

[3] M. W. Berry, S. T. Dumais, and T. A. Letsche. Computational methods for intelligent information access, 1995. Presented at the Proceedings of Supercomputing.

[4] M. W. Berry, S. T. Dumais, and G. W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Rev.*, 37(4):573–595, 1995.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT Press, Cambridge, MA, second edition, 2001.

[6] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *J. Soc. Indust. Appl. Math. Ser. B Numer. Anal.*, 2:205–224, 1965.

[7] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[8] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM J. Matrix Anal. Appl.*, 16(1):79–92, 1995.

[9] E. R. Jessup and D. C. Sorensen. A parallel algorithm for computing the singular value decomposition of a matrix. *SIAM J. Matrix Anal. Appl.*, 15(2):530–548, 1994.

[10] G. Kowalski. *Information Retrieval Systems: Theory and Implementation.* The Kluwer International Series on Information Retrieval. Kluwer Academic Publishers, Boston, MA, first edition, 1997.

[11] S. MacKinnon-Cormier. Efficient singular value decomposition calculation for information retrieval, 2002. Honours Thesis, Dalhousie University.

[12] C. C. Paige. Bidiagonalization of matrices and solutions of the linear equations. *SIAM J. Numer. Anal.*, 11:197–209, 1974.

[13] H. Park, M. Jeon, and J. B. Rosen. Lower dimensional representation of text data in vector space based information retrieval. In *Computational information retrieval (Raleigh, NC, 2000)*, pages 3–23. SIAM, Philadelphia, PA, 2001.

[14] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Computer Science Series. McGraw-Hill, New York, NY, first edition, 1983.

[15] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Rev.*, 35(4):551–566, 1993.

[16] D. Teather. US banks pitch for Google flotation. *The Guardian*, October 25 2003.

[17] L. N. Trefethen and D. Bau, III. *Numerical Linear Algebra.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[18] D. S. Watkins. *Fundamentals of Matrix Computations.* Pure and Applied Mathematics. Wiley-Interscience [John Wiley & Sons], New York, 2002. Second edition.

[19] C. Yu, J. Cuadrado, M. Ceglowski, and J. S. Payne. Patterns in unstructured data: Discovery, aggregation, and visualization, 2002. A Presentation to the Andrew W. Mellon Foundation.