Limited-Memory Matrix Methods with Applications[1]

Tamara Gibson Kolda[2]
Applied Mathematics Program
University of Maryland at College Park

Abstract. The focus of this dissertation is on matrix decompositions that use a limited amount of computer memory, thereby allowing problems with a very large number of variables to be solved. Specifically, we will focus on two applications areas: optimization and information retrieval.

We introduce a general algebraic form for the matrix update in limited-memory quasi-Newton methods. Many well-known methods such as limited-memory Broyden Family methods satisfy the general form. We are able to prove several results about methods which satisfy the general form. In particular, we show that the only limited-memory Broyden Family method (using exact line searches) that is guaranteed to terminate within n iterations on an n-dimensional strictly convex quadratic is the limited-memory BFGS method. Furthermore, we are able to introduce several new variations on the limited-memory BFGS method that retain the quadratic termination property. We also have a new result that shows that full-memory Broyden Family methods (using exact line searches) that skip p updates to the quasi-Newton matrix will terminate in no more than $n+p$ steps on an n-dimensional strictly convex quadratic. We propose several new variations on the limited-memory BFGS method and test these on standard test problems.

We also introduce and test a new method for a process known as Latent Semantic Indexing (LSI) for information retrieval. The new method replaces the singular value matrix decomposition (SVD) at the heart of LSI with a semi-discrete matrix decomposition (SDD). We show several convergence results for the SDD and compare some strategies for computing it on general matrices. We also compare the SVD-based LSI to the SDD-based LSI and show that the SDD-based method has a faster query computation time and requires significantly less storage. We also propose and test several SDD-updating strategies for adding new documents to the collection.

---

[1]Dissertation submitted to the Faculty of the Graduate School of the University of Maryland at College Park in partial fulfillment of the requirements for the degree of Doctor of Philosophy. Dianne P. O'Leary of the Department of Computer Science and the Institute for Advanced Computer Studies was the advisor for this dissertation.

[2]This work was supported in part by the National Physical Science Consortium, the National Security Agency, the IDA Center for Computer Sciences, and the University of Maryland.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The focus of this dissertation is on matrix decompositions that use a limited amount of computer memory, thereby allowing problems with a very large number of variables to be solved. Specifically, we will focus on two applications areas: optimization (Chapters 2 – 5) and information retrieval (Chapters 6 – 8).

The goal in optimization is to find the minimum of a real-valued function of n variables. Most methods for solving minimization problems fall into the category of (iterative) quasi-Newton methods. At each iteration of a quasi-Newton method, we generate a search direction using the gradient and an approximation to the Hessian (or its inverse), choose a step length along the search direction, compute the next iterate and gradient, and choose a new Hessian approximation. Different choices for the Hessian approximation yield different methods. We are particularly interested in choices for the approximate Hessian that do not explicitly store a dense matrix. Many methods are able to construct a Hessian approximation using only a few vectors - these methods are loosely grouped under the title of limited-memory quasi-Newton methods. Chapter 2 reviews both full- and limited-memory quasi-Newton methods.

In Chapter 3, we present a new general algebraic form for the Hessian approximation in quasi-Newton methods. The general algebraic form incorporates both full- and limited-memory methods. The main result specifies which methods fitting this algebraic form will terminate within n iterations on an n-dimensional strictly quadratic function (assuming the line search to determine the step length is exact). This result generalizes and extends existing results about quadratic termination. One consequence of this result is that we now have quadratic termination results for limited-memory Broyden Family methods. Broyden Family methods are the most popular methods for solving small optimization problems. One member of the Broyden Family, BFGS, has a limited-memory analog (L-BFGS) which is very popular for solving large optimization problems. L-BFGS is known to have the quadratic termination property, but nothing was known about the other limited-memory Broyden Family methods. Our result shows that, in fact, BFGS is the only Broyden Family method to have a limited-memory analog with the quadratic termination property. Furthermore, we can derive new variations to L-BFGS which retain the quadratic termination property.

Chapter 4 focuses on a particular aspect of full-memory Broyden Class methods. We show that if we skip p updates to the Hessian approximation in a full-memory Broyden Family method (assuming the line search to determine the step length is exact), then the algorithm will terminate in no more than $n + p$ iterations on an n-dimensional strictly quadratic function.

Chapter 5 describes some new variations on the L-BFGS method and tests them on optimization problems from a standard problem collection. We will see that these variations have promise.

In Chapter 6 we explore a matrix approximation method that we call the semi-discrete decomposition (SDD). The SDD approximates an $m \times n$ matrix via a sum of rank-1 matrices of the form dxy^T where d is a positive real scalar, x is an m-vector whose entries are restricted to the set $\{-1, 0, 1\}$ and y is an n vector whose entries are also restricted to the set $\{-1, 0, 1\}$. We describe the SDD and how to construct it. We present new results which show that the approximation converges linearly to the true matrix, and we test methods for constructing the approximation.

In Chapter 7 we digress to describe the vector space method for information retrieval. In information retrieval, we wish to match a query to relevant documents in a collection. The vector space method represents the document collection as a matrix, called the term-document matrix, and the query as a vector. The (i, j)th entry of the term-document matrix is nonzero only if term i appears in document j. The exact value of the entry can be determined via numerous different schemes that are detailed in the chapter, but think of the value as a measurement of importance. Similarly, the ith entry of the query vector is a measurement of the ith term's importance to the query and is nonzero only if the ith term appears in the query. The jth column of the term-document matrix represents the jth document, and the score of that document for a given query is determined by computing the inner product of the document vector and the query vector. The documents can then be ranked by their inner products with the query. We describe this in more detail and survey recent improvements on this model in the chapter.

One major improvement on the vector space method is Latent Semantic Indexing (LSI). LSI is special because it has the ability to automatically recognize inter-word relationships In Chapter 8, we describe LSI in detail and show how the SDD can be used to drastically improve LSI with respect to storage requirements and time to perform a query. The main idea behind LSI is the replacement of the term-document matrix used in the vector space model with a low-rank approximation generated via the singular value decomposition (SVD). The main difficulty in this approach is that even a low-rank SVD requires substantially more storage than the original term-document matrix (which is sparse). Replacing the SVD with the SDD reduces the storage requirements substantially; in fact, the SDD approximation requires less storage than the original term-document matrix. The time for the query computation is also reduced. We also show how the SDD approximation can be updated in the event that new documents are added.

Chapter 2

Quasi-Newton Methods for Optimization

The next four chapters consist of material taken (sometimes verbatim) from Kolda, O'Leary and Nazareth [40].

2.1 Introduction

The problem we wish to solve in optimization is the following:

$$\min_{x \in \Re^n} f(x),$$

where the function f maps $\Re^n$ to $\Re$. We will assume that f is twice continuously differentiable and let g and G denote its gradient and Hessian respectively. The solution, x^*, is called the *minimizer*, and $f(x^*)$ is called the *minimum*. Specifically, we are looking for an x^* such that $f(x^*) \leq f(x)$ for all x in a neighborhood of x^*, in other words, for a *local* minimizer.

We will focus on *quasi-Newton* iterative methods for the solution of this problem, and more precisely, we are interested in techniques that are computationally feasible for large problems. Figure 2.1 outlines a general quasi-Newton method. Here x_k denotes the kth iterate, $g_k \equiv g(x_k)$, and H_k is the $n \times n$ *quasi-Newton matrix*.

At each iteration of a quasi-Newton method we model the function using a quadratic

$$f(x_k + d) \approx f(x_k) + d^T g_k + \frac{1}{2} d^T H_k^{-1} d \equiv \Phi_k(d). \tag{2.1}$$

The quadratic Φ_k is minimized when its gradient with respect to d is zero, that is,

$$g_k + H_k^{-1} d = 0,$$

and H_k is positive definite[1]. In that case, we can solve for the optimal d, denoted by d_k:

$$d_k = -H_k g_k.$$

[1] H_k may not always be chosen to be positive definite, but here we are just establishing the framework for the general method.

1. Let x_0 be the starting point. Compute g_0.
2. Let H_0 be an $n \times n$ positive definite matrix.
3. $k \leftarrow 0$.
4. Until convergence do:
   (a) Compute $d_k = -H_k g_k$.
   (b) Choose steplength $\alpha_k > 0$.
   (c) Compute $x_{k+1} = x_k + \alpha_k d_k$.
   (d) Compute g_{k+1}.
   (e) Choose quasi-Newton matrix H_{k+1}.
   (f) $k \leftarrow k + 1$.
5. End do.

Figure 2.1: Quasi-Newton Method

Although d_k is the minimizer of Φ_k, $(x_k + d_k)$ is not necessarily a minimizer of f in the direction d_k, so we digress for a moment to discuss possible choices for the *steplength*, α_k.

There are three ways we can choose α_k:

1. We say a method is *perfect* if we always choose α so that

$$f(x_k + \alpha_k d_k) \leq f(x_k + \alpha d_k), \text{ for all } \alpha > 0.$$

   Performing an exact line search at each iteration is generally too expensive to do in practice; however, perfect methods are interesting from a theoretical point of view.

2. We call a method a *direct prediction method* if we always choose $\alpha_k = 1$. Direct prediction methods can work well locally, but often cause problems if the initial starting point is not sufficiently close to the minimizer.

3. Otherwise we say the method uses an *inexact line search*. Here, we will accept a positive steplength α if it satisfies the *Wolfe conditions*[2]:

$$f(x + \alpha d) \leq f(x) + \omega_1 \, \alpha \, g(x)^T d, \tag{2.2}$$
$$g(x + \alpha d)^T d \geq \omega_2 \, g(x)^T d, \tag{2.3}$$

[2]These are also sometimes referred to as the *Goldstein-Armijo conditions.*

where $0 < \omega_1 < \omega_2 < 1$. The first condition guarantees sufficient decrease in f, and the second condition safeguards against steplengths being too small. If d_k is a descent direction, then there exists an α_k satisfying the Wolfe conditions; furthermore, if we always choose a descent direction and a steplength satisfying the Wolfe conditions, then we are guaranteed global convergence (assuming f is bounded below and g is Lipschitz continuous) [19]. In our experiments, we replace (2.3) with a slightly stronger curvature condition:

$$|\, g(x + \alpha d)^T d \,| \leq \omega_2 \; |\, g(x)^T d \,|. \tag{2.4}$$

Our theory will focus primarily on perfect methods but our experiments will use an inexact line search (see Chapter 5 for further details).

Note that there are two choices to make at each iteration: steplength and quasi-Newton matrix. We have already discussed some possible ways to choose the steplength. This research focuses on the choice of the quasi-Newton matrix. For the remainder of this chapter we will discuss existing quasi-Newton methods and their advantages and disadvantages. In the next chapter, we will present a result that generalizes some of the results seen here. Of special interest to us is the performance of these methods on strictly convex quadratic functions.

2.2 Newton's Method

Newton's method is the basis for quasi-Newton methods, so it is logical to discuss this method first. Here we choose H_k to be $[G(x_k)]^{-1}$, so the model Φ_k is the 2nd order Taylor expansion of f about x_k. Newton's method will find the exact minimizer of a strictly convex quadratic function in only one iteration.

If, for example, we use a backtracking line search strategy which obeys the Wolfe conditions (2.2) and (2.3) with $\omega_1 < \frac{1}{2}$, and always try a step length of one first, then Newton's method is *globally convergent*[3] and the rate of convergence is quadratic [19]. We must check to be sure that Newton's method always produces a descent direction.

There are also some computational disadvantages for Newton's Method. At each iteration, we must compute the Hessian of f and solve the equation

$$G(x_k)\; d = -g_k. \tag{2.5}$$

The Hessian may not be analytically available, and even if it is, solving the linear equation is expensive unless $G(x_k)$ has special structure. Furthermore, we are required to store the $n \times n$ matrix $G(x_k)$.

The disadvantages of Newton's method make it impractical for large scale optimization. This research will focus on using other matrices in place of $G(x_k)$, but the reader should also be aware of *truncated-Newton methods* that solve the Newton equation (2.5) using an

[3]By globally convergent, we mean that the method converges to some local minimizer from any starting point whenever f is sufficiently smooth and bounded below.

iterative method and finite-difference approximations to approximate the action of $G(x_k)$; see O'Leary [50] or Dembo and Steihaug [18] for more information on this approach.

2.3 Steepest Descent

Compared to Newton's method, the *method of steepest descent* is at the opposite end of the spectrum. As its name implies, at each iteration, we take a step in the direction of steepest descent, that is, $-g_k$. This corresponds to choosing H_k as the identity matrix.

Computationally, this method is attractive because each iteration only requires the computation of the gradient and the calculation of the steplength. Although the work per iteration is cheap, it requires a large number of iterations to converge to the true solution. The rate of convergence when using an inexact line search is linear, as compared to quadratic for Newton's method [19]. Furthermore, on a strictly convex quadratic function, the method may never find the exact minimizer [19].

In the next section we will discuss a method that has approximately the same work per iteration as steepest descent but performs better.

2.4 Conjugate Gradients

The method of conjugate gradients [35] for minimization is described in Section 8.6 of Luenberger [42].[4] In the preconditioned Fletcher-Reeves [26] version with preconditioner H_0, the search direction d_{k+1} can be expressed as

$$d_{k+1} = -\left(I - \frac{d_k g_k^T}{g_{k-1}^T g_{k-1}}\right) H_0 g_k.$$

Conjugate gradients terminates in no more than n iterations on a strictly convex quadratic function [30, 35, 42]. Furthermore, the method is globally convergent if every step satisfies the strong Wolfe conditions (2.2) and (2.4) with $0 < \omega_1 < \omega_2 < \frac{1}{2}$. [1]

2.5 Broyden Family

The Broyden Family methods [8] use an approximation to the inverse Hessian that is updated via a rank-1 or rank-2 symmetric update at each iteration. The update is of the form

$$H_{k+1} = \phi\, H_{k+1}^{BFGS} + (1-\phi)\, H_{k+1}^{DFP}, \qquad \phi \in \Re.$$

The BFGS [8, 29, 23, 60] and DFP [16, 25] updates are given by

$$H_{k+1}^{\mathrm{BFGS}} = (I - \rho_k s_k y_k^T) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T,$$

[4]The example in Luenberger requires a restart every n iterations, but we are not making that assumption.

and

$$H_{k+1}^{\mathrm{DFP}} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \rho_k s_k s_k^T,$$

where

$$\begin{aligned} s_k &\equiv x_{k+1} - x_k, \\ y_k &\equiv g_{k+1} - g_k, \\ \rho_k &\equiv s_k^T y_k. \end{aligned}$$

Every Broyden Family member satisfies the *secant condition*, that is,

$$H_{k+1} y_k = s_k, \tag{2.6}$$

and these methods are sometimes referred to as *symmetric secant methods* [19].

Powell [53] showed that the perfect DFP method has a superlinear rate of convergence, and Dixon [20] showed that all perfect Broyden Family methods produce exactly the same search directions at each iteration, assuming that the quasi-Newton matrix is always defined. Thus, perfect Broyden Family methods have a superlinear rate of convergence as long as the quasi-Newton matrix is defined. Furthermore, perfect Broyden Family methods always terminate in no more than n iterations on a strictly convex quadratic function [54].

If we use an inexact line search satisfying the Wolfe conditions with $0 < \omega_1 < \omega_2 < \frac{1}{2}$ and choose a step of one whenever possible, then Powell [55] showed that the BFGS method is globally and superlinearly convergent for any choice of x_0 and positive definite H_0 provided that f is convex, twice continuously differentiable and the set $\{x : f(x) \leq f(x_0)\}$ is bounded. These results were later extended to Broyden Family members with $\phi \in (0,1]$ [11]. It is still an open question whether or not DFP ($\phi = 0$) converges superlinearly with an inexact linesearch [48].

2.6 Limited-Memory Broyden Family

At each iteration in a Broyden Family method, we have an update of the form

$$H_{k+1} = U(H_k, s_k, y_k).$$

This establishes a recurrence relation:

$$\begin{aligned} H_{k+1} &= U(H_k, s_k, y_k) \\ &= U(U(H_{k-1}, s_{k-1}, y_{k-1}), s_k, y_k) \\ &= U(U(\cdots U(H_{k-m+1}, s_{k-m+1}, y_{k-m+1}) \cdots, s_{k-1}, y_{k-1}), s_k, y_k). \end{aligned}$$

If we know H_{k-m+1} and the m pairs (s_{k-m+1}, y_{m-k+1}) through (s_k, y_k) we can reconstruct H_{k+1}. We make this a limited-memory method by replacing H_{k-m+1} with H_0 which is a positive definite matrix that requires little storage; for example, the identity matrix. Thus, the update is given by

$$H_{k+1} = U(U(\cdots U(H_0, s_{k-m+1}, y_{k-m+1}) \cdots, s_{k-1}, y_{k-1}), s_k, y_k).$$

The limited-memory BFGS (L-BFGS) update has a very compact form [47]. It can be written as

$$H_{k+1} = V_{k-m_k+1,k}^T H_0 V_{k-m_k+1,k} + \sum_{i=k-m_k+1}^{k} V_{i+1,k}^T \frac{s_i s_i^T}{s_i^T y_i} V_{i+1,k}, \tag{2.7}$$

where $m_k = \min\{k+1, m\}$ and

$$V_{ik} = \prod_{j=i}^{k} \left(I - \frac{y_i s_i^T}{s_i^T y_i} \right).$$

This representation requires only $O(mn)$ storage and the search direction can be computed implicitly in only $O(mn)$ time [47]. Furthermore, L-BFGS terminates in no more than n iterations on a strictly convex quadratic function [47] and has been shown to be globally and linearly convergent on convex problems for any starting point [41].

The other Broyden methods do not reduce to such a nice form in their limited-memory versions [12] and will be discussed in the next chapter.

Chapter 3

Quadratic Termination Properties of Limited-Memory Quasi-Newton Methods

3.1 A General Form for Limited-Memory Quasi-Newton Updates

Using the notation developed in the previous chapter, we will present a general result that characterizes perfect quasi-Newton methods that terminate in n iterations on an n-dimensional strictly convex quadratic. We restrict ourselves to methods with an update of the form

$$H_{k+1} = \gamma_k P_k^T H_0 Q_k + \sum_{i=1}^{m_k} w_{ik} z_{ik}^T. \tag{3.1}$$

Here,

1. H_0 is an $n \times n$ symmetric positive definite matrix that remains constant for all k, and γ_k is a nonzero scalar that iteratively rescales H_0.

2. P_k is an $n \times n$ matrix that is the product of projection matrices of the form

$$I - \frac{uv^T}{u^T v}, \tag{3.2}$$

where $u \in \operatorname{span}\{y_0, \dots, y_k\}$ and $v \in \operatorname{span}\{s_0, \dots, s_{k+1}\}$[1], and Q_k is an $n \times n$ matrix that is the product of projection matrices of the same form where u is any n-vector and $v \in \operatorname{span}\{s_0, \dots, s_k\}$,

3. m_k is a nonnegative integer, w_{ik} $(i = 1, 2, \dots, m_k)$ is any n-vector, and z_{ik} $(i = 1, 2, \dots, m_k)$ is any vector in $\operatorname{span}\{s_0, \dots, s_k\}$.

[1]Although the vector s_{k+1} has not yet been explicitly calculated, it may be available implicitly as we will show for the limited-memory DFP in the proof of Proposition 3.1.

We refer to this as the *general form*. Many known quasi-Newton methods have updates that can be expressed in the general form (3.1). We do not assume that these quasi-Newton methods satisfy the secant condition (2.6), nor that H_{k+1} is positive definite and symmetric. Symmetric positive definite updates are desirable since this guarantees that the quasi-Newton method produces descent directions. Note that if the update is not positive definite, we may produce a d_k such that $d_k^T g_k > 0$ in which case we choose α_k over all *negative* α rather than all positive α.

3.1.1 Steepest Descent

The method of steepest descent (see Section 2.3) has an update that can be expressed in the general form (3.1). For each k we define

$$\gamma_k = 1, \quad m_k = 0, \text{ and } P_k = Q_k = H_0 = I. \tag{3.3}$$

Note that neither w nor z vectors is specified since $m_k = 0$.

3.1.2 Conjugate Gradients

The $(k+1)$st update for the conjugate gradient method (see Section 2.4) with preconditioner H_0 has an update that can be expressed in the general form (3.1) with

$$\gamma_k = 1, \quad m_k = 0, \quad P_k = I - \frac{y_k s_k^T}{s_k^T y_k}, \text{ and } Q_k = I. \tag{3.4}$$

3.1.3 L-BFGS

The L-BFGS update (see Section 2.6) has an update that can be expressed in the general form (3.1) if at iteration k we choose

$$\begin{aligned} &\gamma_k = 1, \quad m_k = \min\{k+1, m\}, \\ &P_k = Q_k = V_{k-m_k+1,k}, \text{ and} \\ &w_{ik} = z_{ik} = \frac{(V_{k-m_k+i+1,k})^T (s_{k-m_k+i})}{\sqrt{(s_{k-m_k+i})^T (y_{k-m_k+i})}}. \end{aligned} \tag{3.5}$$

Observe that P_k, Q_k and z_{ik} $(i = 1, \ldots, m_k)$ all obey the constraints imposed on their construction.

3.1.4 Limited-Memory DFP

We will define limited-memory DFP (L-DFP) using the framework established in Section 2.6. Let $m \geq 1$ and let $m_k = \min\{k+1, m\}$. In order to define the L-DFP update we need to create a sequence of auxiliary matrices for $i = 0, \ldots, m_k$. Let

$$\begin{aligned}\hat{H}_{k+1}^{(0)} &= H_0, \text{ and} \\ \hat{H}_{k+1}^{(i)} &= U_{\text{DFP}}(\hat{H}_{k+1}^{(i-1)}, s_{k-m_k+i}, y_{k-m_k+i}),\end{aligned}$$

where

$$U_{\text{DFP}}(H, s, y) = H - \frac{Hyy^T H}{y^T H y} + \frac{ss^T}{s^T y}.$$

The matrix $\hat{H}_{k+1}^{(m_k)}$ is the result of applying the DFP update m_k times to the matrix H_0 with the m_k most recent (s, y) pairs. Thus, the $(k+1)$st L-DFP matrix is given by

$$H_{k+1} = \hat{H}_{k+1}^{(m_k)}.$$

To simplify our description, note that $\hat{H}_{k+1}^{(i)}$ can be rewritten as

$$\begin{aligned}\hat{H}_{k+1}^{(i)} &= \left(I - \frac{\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i} y_{k-m_k+i}^T}{y_{k-m_k+i}^T \hat{H}_{k+1}^{(i-1)} y_{k-m_k+i}} \right) \hat{H}_{k+1}^{(i-1)} + \frac{s_{k-m_k+i} s_{k-m_k+i}^T}{s_{k-m_k+i}^T y_{k-m_k+i}} \\ &= \left(\hat{V}_{0k}^{(i)} \right)^T H_0 + \sum_{j=1}^{i} \left(\hat{V}_{jk}^{(i)} \right)^T \frac{s_{k-m_k+j} s_{k-m_k+j}^T}{s_{k-m_k+j}^T y_{k-m_k+j}},\end{aligned}$$

for $i \geq 1$ where

$$\hat{V}_{jk}^{(i)} = \prod_{l=j+1}^{i} \left[I - \frac{y_{k-m_k+l} \left(H_{k+1}^{(l-1)} y_{k-m_k+l} \right)^T}{y_{k-m_k+l}^T H_{k+1}^{(l-1)} y_{k-m_k+l}} \right].$$

Note that we define the product to be taken from left to right, that is,

$$\prod_{i=j}^{k} B_j = \begin{cases} B_j \cdot B_{j+1} \cdots B_k & \text{if } j \leq k, \\ I & \text{otherwise.} \end{cases}$$

Thus H_{k+1} can be written as

$$H_{k+1} = V_{0k}^T H_0 + \sum_{i=1}^{m_k} \left(V_{ik}^T \frac{s_{k-m_k+i} s_{k-m_k+i}^T}{s_{k-m_k+i}^T y_{k-m_k+i}} \right), \tag{3.6}$$

where

$$V_{ik} = \prod_{j=i+1}^{m_k} \left[I - \frac{y_{k-m_k+j} \left(\hat{H}_{k+1}^{(j-1)} y_{k-m_k+j} \right)^T}{y_{k-m_k+j}^T \hat{H}_{k+1}^{(j-1)} y_{k-m_k+j}} \right].$$

Equation (3.6) fits the general form (3.1) with the following choices:

$$\begin{aligned}&\gamma_k = 1, \quad P_k = V_{0k}, \quad Q_k = I, \\ &w_{ik} = V_{ik}^T s_{k-m_k+i} / (s_{k-m_k+i}^T y_{k-m_k+i}), \text{ and } z_{ik} = s_{k-m_k+i}.\end{aligned} \tag{3.7}$$

Except for the choice of P_k, it is trivial to verify that the choices satisfy the general form (3.1). To prove that P_k satisfies the requirements, we need to show

$$\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i} \in \operatorname{span}\{s_0, \ldots, s_{k+1}\}, \text{ for } i = 1, \ldots, m_k \text{ and all } k. \tag{3.8}$$

Proposition 3.1 *For limited-memory DFP, the following three conditions hold for each value of k:*

$$\begin{aligned}
&\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i} \in \operatorname{span}\{s_0, \ldots, s_k\} \textit{ for } i = 1, \ldots, m_k - 1 \textit{ and} && (3.9)\\
&\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i} \in \operatorname{span}\{s_0, \ldots, s_k, H_0 g_{k+1}\} \textit{ for } i = m_k, \textit{ and} && (3.10)\\
&\operatorname{span}\{H_0 g_0, \ldots, H_0 g_{k+1}\} \subseteq \operatorname{span}\{s_0, \ldots, s_{k+1}\}. && (3.11)
\end{aligned}$$

Proof. We will prove this via induction. Suppose $k = 0$. Then $m_0 = 1$. We have

$$\hat{H}_{k+1}^{(0)} y_k = H_0 y_0 = H_0 g_1 - H_0 g_0 \in \operatorname{span}\{s_0, H_0 g_1\}.$$

(Recall that $\operatorname{span}\{s_0\}$ is trivially equal to $\operatorname{span}\{H_0 g_0\}$.) Furthermore,

$$\begin{aligned}
s_1 &= -\alpha_1 H_1 g_1 \\
&= -\alpha_1 \left[H_0 g_1 - \frac{y_0^T H_0 g_1}{y_0^T H_0 y_0} (H_0 g_1 - H_0 g_0) + \frac{s_0^T g_1}{y_0^T s_0} s_0 \right].
\end{aligned}$$

So we can conclude,

$$\left(1 - \frac{y_0^T H_0 g_1}{y_0^T H_0 y_0}\right) H_0 g_1 = -\left[\frac{1}{\alpha_1} s_1 + \frac{y_0^T H_0 g_1}{y_0^T H_0 y_0} H_0 g_0 + \frac{s_0^T g_1}{y_0^T s_0} s_0 \right].$$

Hence, $H_0 g_1 \in \operatorname{span}\{s_0, s_1\}$, and so the base case holds.

Assume that

$$\begin{aligned}
&\hat{H}_k^{(i-1)} y_{k-1-m_{k-1}+i} \in \operatorname{span}\{s_0, \ldots, s_{k-1}\} \text{ for } i = 1, \ldots, m_{k-1} - 1, \text{ and} \\
&\hat{H}_k^{(i-1)} y_{k-1-m_{k-1}+i} \in \operatorname{span}\{s_0, \ldots, s_{k-1}, H_0 g_k\} \text{ for } i = m_{k-1}, \text{ and} \\
&\operatorname{span}\{H_0 g_0, \ldots, H_0 g_k\} \subseteq \operatorname{span}\{s_0, \ldots, s_k\}.
\end{aligned}$$

Using the induction assumption, we will show that (3.9) – (3.11) holds for $(k+1)$. We show (3.9) for $i = 1, \ldots, m_k - 1$. For $i = 1$ (assume $m_k > i$),

$$\hat{H}_{k+1}^{(0)} y_{k-m_k+1} = H_0 y_{k-m_k+1} = H_0 g_{k-m_k+2} - H_0 g_{k-m_k+1}.$$

Using the induction hypotheses, we get that

$$\hat{H}_{k+1}^{(0)} y_{k-m_k+1} \in \operatorname{span}\{s_0, \ldots, s_k\}.$$

Assume that

$$\hat{H}_{k+1}^{(j)} y_{k-m_k+j+1} \in \operatorname{span}\{s_0, \ldots, s_k\} \tag{3.12}$$

for j between 1 and $i-2$, $i \leq m_k - 2$. Then,

$$\begin{aligned}\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i} &= \left(\hat{V}_{0k}^{(i-1)}\right)^T H_0 y_{k-m_k+i} \\ &\quad + \sum_{j=1}^{i-1} \frac{s_{k-m_k+j-1}^T y_{k-m_k+i}}{s_{k-m_k+j-1}^T y_{k-m_k+j-1}} \left(\hat{V}_{jk}^{(i-1)}\right)^T s_{k-m_k+j-1}.\end{aligned}$$

For values of $i \leq m_k - 1$, $\left(\hat{V}_{jk}^{(i-1)}\right)^T$ maps any vector v into

$$\operatorname{span}\{v, \hat{H}_{k+1}^{(0)} y_{k-m_k+1}, \ldots, \hat{H}_{k+1}^{(i-2)} y_{k-2}\}.$$

and so $\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i}$ is in

$$\operatorname{span}\{H_0 y_{k-m_k+i}, \hat{H}_{k+1}^{(0)} y_{k-m_k+1}, \ldots, \hat{H}_{k+1}^{(i-2)} y_{k-2}, s_{k-m_k+1}, \ldots, s_{k-2}\}.$$

Using the induction hypothesis and (3.12), we get

$$\hat{H}_{k+1}^{(i-1)} y_{k-m_k+i} \in \operatorname{span}\{s_0, \ldots, s_k\},$$

and we can conclude that (3.9) is true for $i = 1, \ldots, m_k - 1$ in the $(k+1)$st case. If $i = m_k$, then

$$\hat{H}_{k+1}^{(m_k-1)} y_k \in \operatorname{span}\{H_0 y_k, \hat{H}_{k+1}^{(0)} y_{k-m_k+1}, \ldots, \hat{H}_{k+1}^{(m_k-2)} y_{k-1}, s_{k-m_k+1}, \ldots, s_{k-1}\},$$

so

$$\hat{H}_{k+1}^{(m_k-1)} y_k \in \operatorname{span}\{s_0, \ldots, s_k, H_0 g_{k+1}\}.$$

Hence (3.10) is true for $(k+1)$.

Now, consider

$$\begin{aligned} s_{k+1} &= -\alpha_{k+1} H_{k+1} g_{k+1} \\ &= V_{0k}^T H_0 g_{k+1} + \sum_{i=1}^{m_k} \frac{s_{k-m_k+i}^T g_{k+1}}{s_{k-m_k+i}^T y_{k-m_k+i}} V_{ik}^T s_{k-m_k+i}. \end{aligned}$$

Using the structure of V_{jk} and (3.9) we see that

$$H_0 g_{k+1} \in \operatorname{span}\{s_0, \ldots, s_{k+1}\}.$$

Hence, (3.11) also holds for $(k+1)$. $\square$

3.1.5 Limited-Memory Broyden Family

The Broyden Family is described in Section 2.5. The parameter ϕ is usually restricted to values that are guaranteed to produce a positive definite update, although recent work with SR1, a Broyden Family method, by Khalfan, Byrd and Schnabel [36] may change this practice. No restriction on ϕ is necessary for the development of our theory. The Broyden Family update can be expressed as

$$\begin{aligned} H_{k+1} &= H_k + \frac{s_k s_k^T}{s_k^T y_k} - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} \\ &\quad + \phi\, y_k^T H_k y_k \left(\frac{s_k}{s_k^T y_k} - \frac{H_k y_k}{y_k^T H_k y_k} \right) \left(\frac{s_k}{s_k^T y_k} - \frac{H_k y_k}{y_k^T H_k y_k} \right)^T . \end{aligned}$$

We sketch the explanation of how the full-memory version has an update that can be expressed in the general form (3.1). The limited-memory case is similar. We can rewrite the Broyden Family update as follows:

$$\begin{aligned} H_{k+1} &= H_k + (\phi - 1) \frac{H_k y_k y_k^T}{y_k^T H_k y_k} H_k - \phi \frac{s_k y_k^T}{s_k^T y_k} H_k + \frac{s_k s_k^T}{s_k^T y_k} \\ &\quad + \phi \frac{y_k^T H_k y_k \cdot s_k s_k^T}{(s_k^T y_k)^2} - \phi \frac{H_k y_k s_k^T}{s_k^T y_k} \\ &= \left[I - \frac{\left((1-\phi) s_k^T y_k \cdot H_k y_k + \phi\, y_k^T H_k y_k \cdot s_k \right) y_k^T}{y_k^T H_k y_k \cdot s_k^T y_k} \right] H_k \\ &\quad + \left[\left(1 + \phi \frac{y_k^T H_k y_k}{s_k^T y_k} \right) s_k - \phi\, H_k y_k \right] \frac{s_k^T}{s_k^T y_k}. \end{aligned}$$

Hence,

$$H_{k+1} = V_{0k} H_0 + \sum_{i=1}^{k+1} w_{ik} z_{ik}^T,$$

where

$$V_{ik} = \prod_{j=i}^{k} \left[I - \frac{\left((1-\phi) s_j^T y_j \cdot H_j y_j + \phi\, y_j^T H_j y_j \cdot s_j \right) y_j^T}{y_j^T H_j y_j \cdot s_j^T y_j} \right],$$

$$w_{ik} = V_{ik} \left[\left(1 + \phi \frac{y_{i-1}^T H_{i-1} y_{i-1}}{s_{i-1}^T y_{i-1}} s_{i-1} \right) - \phi\, H_{i-1} y_{i-1} \right], \text{ and } z_{ik} = \frac{s_{i-1}^T}{s_{i-1}^T y_{i-1}}.$$

It is left to the reader to show that $H_k y_k$ is in $\text{span}\{s_0, \ldots, s_{k+1}\}$, and thus the Broyden Family updates fit the general form (3.1).

3.2 Termination of Limited-Memory Methods

In this section we show that methods fitting the general form (3.1) produce conjugate search directions (Theorem 3.1) and terminate in n iterations (Corollary 3.1) on a strictly convex

n-dimensional quadratic *if and only if* P_k maps $\text{span}\{y_0, \ldots, y_k\}$ into $\text{span}\{y_0, \ldots, y_{k-1}\}$ for each $k = 1, 2, \ldots, n$. Furthermore, this condition on P_k is satisfied only if y_k is used in its formation (Corollary 3.2).

Theorem 3.1 *Suppose that we apply a quasi-Newton method (Figure 2.1) with an update that can be expressed in the general form* (3.1) *to minimize an n-dimensional strictly convex quadratic function*

$$f(x) = \frac{1}{2} x^T A x - b^T x.$$

Then for each k, we have

$$g_{k+1}^T s_j = 0, \text{ for all } j = 0, 1, \ldots, k, \tag{3.13}$$

$$s_{k+1}^T A s_j = 0, \text{ for all } j = 0, 1, \ldots, k, \text{ and} \tag{3.14}$$

$$\text{span}\{s_0, \ldots, s_{k+1}\} = \text{span}\{H_0 g_0, \ldots, H_0 g_{k+1}\}, \tag{3.15}$$

if and only if

$$P_j y_i \in \text{span}\{y_0, \ldots, y_{j-1}\}, \text{ for all } i = 0, 1, \ldots, j, \quad j = 0, 1, \ldots, k. \tag{3.16}$$

Proof. ($\Leftarrow$) Assume that (3.16) holds. We will prove (3.13)–(3.15) by induction. Since the line searches are exact, g_1 is orthogonal to s_0. Using the fact that $P_0 y_0 = 0$ from (3.16), and the fact that $z_{i0} \in \text{span}\{s_0\}$ implies $g_1^T z_{i0} = 0$, $i = 1, \ldots, m_k$, we see that s_1 is conjugate to s_0 since

$$\begin{aligned}
s_1^T A s_0 &= \alpha_1 d_1^T y_0 \\
&= -\alpha_1 g_1^T H_1^T y_0 \\
&= -\alpha_1 g_1^T \left(\gamma_0 Q_0^T H_0 P_0 + \sum_{i=1}^{m_0} z_{i0} w_{i0}^T \right) y_0 \\
&= -\alpha_1 \left(\gamma_0 g_1^T Q_0^T H_0 P_0 y_0 + \sum_{i=1}^{m_0} g_1^T z_{i0} w_{i0}^T y_0 \right) \\
&= 0.
\end{aligned}$$

Lastly, $\text{span}\{s_0\} = \text{span}\{H_0 g_0\}$, and so the base case is established.

We will assume that claims (3.13)–(3.15) hold for $k = 0, 1, \ldots, \hat{k} - 1$ and prove that they also hold for $k = \hat{k}$.

The vector $g_{\hat{k}+1}$ is orthogonal to $s_{\hat{k}}$ since the line search is exact. Using the induction hypotheses that $g_{\hat{k}}$ is orthogonal to $\{s_0, \ldots, s_{\hat{k}-1}\}$ and $s_{\hat{k}}$ is conjugate to $\{s_0, \ldots, s_{\hat{k}-1}\}$, we see that for $j < \hat{k}$,

$$g_{\hat{k}+1}^T s_j = (g_{\hat{k}} + y_{\hat{k}})^T s_j = (g_{\hat{k}} + A s_{\hat{k}})^T s_j = 0.$$

Hence, (3.13) holds for $k = \hat{k}$.

To prove (3.14), we note that

$$s_{\hat{k}+1}^T A s_j = -\alpha_{\hat{k}+1} g_{\hat{k}+1}^T H_{\hat{k}+1}^T y_j,$$

so it is sufficient to prove that $g_{\hat{k}+1}^T H_{\hat{k}+1}^T y_j = 0$ for $j = 0, 1, \ldots, \hat{k}$. We will use the following facts:

1. $g_{\hat{k}+1}^T Q_{\hat{k}}^T = g_{\hat{k}+1}^T$ since the v in each of the projections used to form $Q_{\hat{k}}$ is in span$\{s_0, \ldots, s_{\hat{k}}\}$ and $g_{\hat{k}+1}$ is orthogonal to that span.
2. $g_{\hat{k}+1}^T z_{i\hat{k}} = 0$ for $i = 1, \ldots, m_{\hat{k}}$ since each $z_{i\hat{k}}$ is in span$\{s_0, \ldots, s_{\hat{k}}\}$ and $g_{\hat{k}+1}$ is orthogonal to that span.
3. Since we are assuming that (3.16) holds true, for each $j = 0, 1, \ldots, \hat{k}$ there exist $\mu_0, \ldots, \mu_{\hat{k}-1}$ such that $P_{\hat{k}} y_j$ can be expressed as $\sum_{i=0}^{\hat{k}-1} \mu_i y_i$.
4. For $i = 0, 1, \ldots, \hat{k}-1$, $g_{\hat{k}+1}$ is orthogonal to $H_0 y_i$ because $g_{\hat{k}+1}$ is orthogonal to span$\{s_0, \ldots, s_{\hat{k}}\}$ and $H_0 y_i \in$ span$\{s_0, \ldots, s_{\hat{k}}\}$ from (3.15).

Thus,

$$\begin{aligned}
g_{\hat{k}+1}^T H_{\hat{k}+1}^T y_j &= g_{\hat{k}+1}^T \left(\gamma_{\hat{k}} Q_{\hat{k}}^T H_0 P_{\hat{k}} + \sum_{i=1}^{m_{\hat{k}}} z_{i\hat{k}} w_{i\hat{k}}^T \right) y_j \\
&= \gamma_{\hat{k}} g_{\hat{k}+1}^T Q_{\hat{k}}^T H_0 P_{\hat{k}} y_j + \sum_{i=1}^{m_{\hat{k}}} g_{\hat{k}+1}^T z_{i\hat{k}} w_{i\hat{k}}^T y_j \\
&= \gamma_{\hat{k}} g_{\hat{k}+1}^T H_0 P_{\hat{k}} y_j \\
&= \gamma_{\hat{k}} g_{\hat{k}+1}^T H_0 \left(\sum_{i=1}^{\hat{k}-1} \mu_i y_i \right) \\
&= \gamma_{\hat{k}} \sum_{i=1}^{\hat{k}-1} \mu_i g_{\hat{k}+1}^T H_0 y_i \\
&= 0.
\end{aligned}$$

Thus, (3.14) holds for $k = \hat{k}$.

Lastly, using (1) and (2) from above,

$$\begin{aligned}
s_{\hat{k}+1} &= -\alpha_{\hat{k}+1} H_{\hat{k}+1} g_{\hat{k}+1} \\
&= -\alpha_{\hat{k}+1} \left(\gamma_{\hat{k}} P_{\hat{k}}^T H_0 Q_{\hat{k}} g_{\hat{k}+1} + \sum_{i=1}^{m_{\hat{k}}} w_{i\hat{k}} z_{i\hat{k}}^T g_{\hat{k}+1} \right) \\
&= -\alpha_{\hat{k}+1} \gamma_{\hat{k}} P_{\hat{k}}^T H_0 g_{\hat{k}+1}.
\end{aligned}$$

Since $P_{\hat{k}}^T$ maps any vector v into span$\{v, s_0, \ldots, s_{\hat{k}+1}\}$ by construction, there exist $\sigma_0, \ldots, \sigma_{\hat{k}+1}$ such that

$$s_{\hat{k}+1} = -\alpha_{\hat{k}+1} \gamma_{\hat{k}} \left(H_0 g_{\hat{k}+1} + \sum_{i=0}^{\hat{k}+1} \sigma_i s_i \right).$$

Hence,

$$H_0 g_{\hat{k}+1} \in \mathrm{span}\{s_0, \ldots, s_{\hat{k}+1}\},$$

so

$$\mathrm{span}\{H_0 g_0, \ldots, H_0 g_{\hat{k}+1}\} \subseteq \mathrm{span}\{s_0, \ldots, s_{\hat{k}+1}\}.$$

To show equality of the sets, we will show that $H_0 g_{\hat{k}+1}$ is linearly independent of $\{H_0 g_0, \ldots, H_0 g_{\hat{k}}\}$. (We already know that the vectors $H_0 g_0, \ldots, H_0 g_{\hat{k}}$ are linearly independent since they span the same space as the linearly independent set $\{s_0, \ldots, s_{\hat{k}}\}$.) Suppose that $H_0 g_{\hat{k}+1}$ is not linearly independent. Then there exist $\phi_0, \ldots, \phi_{\hat{k}}$, not all zero, such that

$$H_0 g_{\hat{k}+1} = \sum_{i=0}^{\hat{k}} \phi_i H_0 g_i.$$

Recall that $g_{\hat{k}+1}$ is orthogonal to $\{s_0, \ldots, s_{\hat{k}}\}$. By our induction hypothesis, this implies that $g_{\hat{k}+1}$ is also orthogonal to $\{H_0 g_0, \ldots, H_0 g_{\hat{k}}\}$. Thus for any j between 0 and $\hat{k}$,

$$0 = g_{\hat{k}+1}^T H_0 g_j = \left(\sum_{i=0}^{\hat{k}} \phi_i H_0 g_i \right)^T g_j = \sum_{i=0}^{\hat{k}} \phi_i g_i^T H_0 g_j = \phi_j g_j^T H_0 g_j.$$

Since H_0 is positive definite and g_j is nonzero, we conclude that ϕ_j must be zero. Since this is true for every j between zero and k, we have a contradiction. Thus, the set $\{H_0 g_0, \ldots, H_0 g_{\hat{k}+1}\}$ is linearly independent. Hence, (3.15) holds for $k = \hat{k}$.

($\Rightarrow$) Assume that (3.13)–(3.15) hold for all k such that $g_{k+1} \neq 0$ but that (3.16) does not hold; i.e., there exist j and k such that $g_{k+1} \neq 0$, j is between 0 and k, and

$$P_k y_j \notin \mathrm{span}\{y_0, \ldots, y_{k-1}\} \tag{3.17}$$

This will lead to a contradiction. By construction of P_k, there exist $\mu_0, \ldots, \mu_k$ such that

$$P_k y_j = \sum_{i=0}^{k} \mu_i y_i. \tag{3.18}$$

By assumption (3.17), μ_k must be nonzero. From (3.14), it follows that $g_{k+1}^T H_{k+1}^T y_j = 0$. Using facts (1), (2), and (4) from before, (3.15) and (3.18), we get

$$\begin{aligned}
0 = g_{k+1}^T H_{k+1}^T y_j &= g_{k+1}^T \left(\gamma_k Q_k^T H_0 P_k + \sum_{i=1}^{m_k} z_{ik} w_{ik}^T \right) y_j \\
&= \gamma_k g_{k+1}^T Q_k^T H_0 P_k y_j + \sum_{i=1}^{m_k} g_{k+1}^T z_{ik} w_{ik}^T y_j \\
&= \gamma_k g_{k+1}^T H_0 P_k y_j \\
&= \gamma_k g_{k+1}^T H_0 \left(\sum_{i=0}^{k} \mu_i y_i \right)
\end{aligned}$$

$$\begin{aligned}
&= \gamma_k \sum_{i=0}^{k} \mu_i g_{k+1}^T H_0 y_i \\
&= \gamma_k \mu_k g_{k+1}^T H_0 y_k \\
&= \gamma_k \mu_k \left(g_{k+1}^T H_0 g_{k+1} - g_{k+1}^T H_0 g_k \right) \\
&= \gamma_k \mu_k g_{k+1}^T H_0 g_{k+1}.
\end{aligned}$$

Thus since neither γ_k nor μ_k is zero, we must have

$$g_{k+1}^T H_0 g_{k+1} = 0,$$

but this is a contradiction since H_0 is positive definite and g_{k+1} was assumed to be nonzero. $\square$

When a method produces conjugate search directions, we can say something about termination.

Corollary 3.1 *Suppose that the assumptions of Theorem 3.1 hold. Suppose further that condition* (3.16) *holds for all k and that $H_j g_j \neq 0$ whenever $g_j \neq 0$. Then the scheme reproduces the iterates from the conjugate gradient method with preconditioner H_0 and terminates in no more than n iterations.*

Proof. Let k be such that $g_0, \ldots, g_k$ are all nonzero and such that $H_i g_i \neq 0$ for $i = 0, \ldots, k$. Since we have a method of the type described in Theorem 3.1 satisfying (3.16), conditions (3.13) – (3.15) hold. We claim that the $(k+1)$st subspace of search directions, $\operatorname{span}\{s_0, \ldots, s_k\}$, is equal to the $(k+1)$st Krylov subspace, $\operatorname{span}\{H_0 g_0, \ldots, (H_0 A)^k H_0 g_0\}$.

From (3.15), we know that $\operatorname{span}\{s_0, \ldots, s_k\} = \operatorname{span}\{H_0 g_0, \ldots, H_0 g_k\}$. We will show via induction that $\operatorname{span}\{H_0 g_0, \ldots, H_0 g_k\} = \operatorname{span}\{H_0 g_0, \ldots, (H_0 A)^k H_0 g_0\}$. This base case is trivial, so assume that

$$\operatorname{span}\{H_0 g_0, \ldots, H_0 g_i\} = \operatorname{span}\{H_0 g_0, \ldots, (H_0 A)^i H_0 g_0\},$$

for some $i < k$. Now,

$$g_{i+1} = A x_{i+1} - b = A(x_i + s_i) - b = A s_i + g_i,$$

and from (3.15) and the induction hypothesis,

$$s_i \in \operatorname{span}\{H_0 g_0, \ldots, H_0 g_i\} = \operatorname{span}\{H_0 g_0, \ldots, (H_0 A)^i H_0 g_0\},$$

which implies that

$$H_0 A s_i \in \operatorname{span}\{(H_0 A) H_0 g_0, \ldots, (H_0 A)^{i+1} H_0 g_0\}.$$

So,

$$H_0 g_{i+1} \in \operatorname{span}\{H_0 g_0, \ldots, (H_0 A)^{i+1} H_0 g_0\}.$$

Hence, the search directions span the Krylov subspace. Since the search directions are conjugate (3.14) and span the Krylov subspace, the iterates are the same as those produced by conjugate gradients with preconditioner H_0.

Since we produce the same iterates as the conjugate gradient method and the conjugate gradient method is well-known to terminate within n iterations [30, 35, 42], we can conclude that this scheme terminates in at most n iterations. $\square$

Note that we require that $H_j g_j$ be nonzero whenever g_j is nonzero; this requirement is necessary since not all the methods produce positive definite updates and it is possible to construct an update that maps g_j to zero. If this were to happen, we would have a breakdown in the method.

The next corollary defines the role that the latest information (s_k and y_k) plays in the formation of the kth H-update.

Corollary 3.2 *Suppose we have a method of the type described in Theorem 3.1 satisfying* (3.16)*. Suppose further that at the kth iteration P_k is composed of p projections of the form in* (3.2)*. Then at least one of the projections must have $u = \sum_{i=0}^{k} \sigma_i y_i$ with $\sigma_k \neq 0$. Furthermore, if P_k is a single projection ($p = 1$), then v must be of the form $v = \rho_k s_k + \rho_{k+1} s_{k+1}$ with $\rho_k \neq 0$.*

Proof. Consider the case of $p = 1$. We have

$$P_k = I - \frac{uv^T}{v^T u},$$

where $u \in \text{span}\{y_0, \ldots, y_k\}$ and $v \in \text{span}\{s_0, \ldots, s_{k+1}\}$. We will assume that

$$u = \sum_{i=0}^{k} \sigma_i y_i \quad \text{ and } v = \sum_{i=0}^{k+1} \rho_i s_i.$$

for some scalars σ_i and ρ_i. By (3.16), there exist $\mu_0, \ldots, \mu_{k-1}$ such that

$$P_k y_k = \sum_{i=0}^{k-1} \mu_i y_i.$$

Then

$$y_k - \frac{v^T y_k}{v^T u} u = \sum_{i=0}^{k-1} \mu_i y_i,$$

and so

$$\frac{v^T y_k}{v^T u} u = y_k - \sum_{i=0}^{k-1} \mu_i y_i. \tag{3.19}$$

From (3.14), the set $\{s_0, \ldots s_k\}$ is conjugate and thus linearly independent. Since we are working with a quadratic, $y_i = A s_i$ for all i; and since A is symmetric positive definite, the set $\{y_0, \ldots, y_k\}$ is also linearly independent. So the coefficient of the y_k on the left-hand side of (3.19) must match that on the right-hand side, thus

$$\frac{v^T y_k}{v^T u} \sigma_k = 1.$$

Hence,

$$\sigma_k \neq 0, \tag{3.20}$$

and y_k must make a nontrivial contribution to P_k.

Next we will show that $\rho_0 = \rho_1 = \cdots = \rho_{k-1} = 0$. Assume that j is between 0 and $k-1$. Then

$$\begin{aligned} P_k y_j &= y_j - \frac{v^T y_j}{v^T u} u \\ &= y_j - \frac{\left(\sum_{i=0}^{k+1} \rho_i s_i \right)^T y_j}{v^T u} u \\ &= y_j - \frac{\sum_{i=0}^{k+1} \rho_i s_i^T A s_j}{v^T u} u \\ &= y_j - \frac{\rho_j s_j^T A s_j}{v^T u} u. \end{aligned}$$

Now $s_j^T A s_j$ is nonzero because A is positive definite. If ρ_j is nonzero then the coefficient of u is nonzero and so y_k must make a nontrivial contribution to $P_k y_j$, implying that $P_k y_j \notin \text{span}\{y_0, \ldots, y_{k-1}\}$. This is a contradiction. Hence, $\rho_j = 0$.

To show that $\rho_k \neq 0$, consider $P_k y_k$. Suppose that $\rho_k = 0$. Then

$$\begin{aligned} v^T y_k &= \rho_{k+1} y_k^T s_{k+1} + \rho_k y_k^T s_k \\ &= \rho_{k+1} s_k^T A s_{k+1} \\ &= 0, \end{aligned}$$

and so

$$P_k y_k = y_k - \frac{v^T y_k}{v^T u} u = y_k.$$

This contradicts $P_k y_k \in \text{span}\{y_0, \ldots, y_{k-1}\}$, so ρ_k must be nonzero.

Now we will discuss the $p > 1$ case. Label the u-components of the p projections as u_1 through u_p. Then

$$P_k y_k = y_k + \sum_{i=1}^{p} \gamma_i u_i,$$

for some scalars γ_1 through γ_p. Furthermore, each u_i can be written as a linear combination of $\{y_0, y_1, \ldots, y_k\}$, so

$$P_k y_k = y_k + \sum_{i=1}^{p} \sum_{j=0}^{k} \gamma_i \sigma_{ij} y_j,$$

for some scalars σ_{10} through σ_{pk}. Since

$$P_k y_k \in \text{span}\{y_0, \ldots, y_{k-1}\},$$

and

$$y_k \notin \text{span}\{y_0, \ldots, y_{k-1}\},$$

we must have

$$1 + \sum_{i=1}^{p} \gamma_i \sigma_{ik} = 0.$$

Thus σ_{ik} must be nonzero for some i, and we can conclude that at least one u_i must have a nontrivial contribution from y_k. $\Box$

3.3 Examples of Methods that Reproduce the CG Iterates

Here are some specific examples of methods that fit the general form (3.1), satisfy condition (3.16) of Theorem 3.1, and thus terminate in at most n iterations. The examples in Sections 3.3.1 – 3.3.3 are well-known results, but the corollary in Section 3.3.4 is original.

3.3.1 Conjugate Gradients

The conjugate gradient method with preconditioner H_0 (see (3.4)) satisfies condition (3.16) of Theorem 3.1 since

$$P_k y_j = \left(I - \frac{y_k s_k^T}{s_k^T y_k} \right) y_j = 0 \text{ for all } j = 0, \ldots, k.$$

3.3.2 L-BFGS

Limited-memory BFGS (see (3.5)) satisfies condition (3.16) of Theorem 3.1 since

$$P_k y_j = \begin{cases} 0 & \text{for } j = k - m_k + 1, \ldots, k, \text{ and} \\ y_j & \text{for } j = 0, \ldots, k - m_k. \end{cases}$$

3.3.3 DFP

DFP (with full memory), see (3.7), satisfies condition (3.16) of Theorem 3.1. Consider P_k in the full memory case. We have

$$P_k = \prod_{i=0}^{k} \left(I - \frac{y_i y_i H_i^T}{y_i^T H_i y_i} \right).$$

For full-memory DFP, $H_i y_j = s_j$ for $j = 0, \ldots, i-1$. Using this fact, one can easily verify that $P_k y_j = 0$ for $j = 0, \ldots, k$. Therefore, full-memory DFP satisfies condition (3.16) of Theorem 3.1. The same reasoning does not apply to the limited-memory case as we shall show in Section 3.4.2.

3.3.4 Variations on L-BFGS

The next corollary gives some ideas for other methods that are related to L-BFGS and terminate in at most n iterations on strictly convex quadratics.

Corollary 3.3 *The L-BFGS method with an exact line search will terminate in n iterations on an n-dimensional strictly convex quadratic function even if any combination of the following modifications is made to the update:*

1. *Vary the limited-memory constant, keeping $m_k \geq 1$.*
2. *Form the projections used in V_k from the most recent (s_k, y_k) pair along with any set of $m-1$ other pairs from $\{(s_0, y_0), \ldots, (s_{k-1}, y_{k-1})\}$.*
3. *Form the projections used in V_k from the most recent (s_k, y_k) pair along with any $m-1$ other linear combinations of pairs from $\{(s_0, y_0), \ldots, (s_{k-1}, y_{k-1})\}$.*
4. *Iteratively rescale H_0.*

Proof. For each variant, we show that the method has an update that can be expressed in the general form (3.1) and satisfies condition (3.16) of Theorem 3.1 and hence terminates by Corollary 3.1.

1. Let $m > 0$ be a value that may change from iteration to iteration, and define

$$V_{ik} = \prod_{j=i}^{k} \left(I - \frac{y_j s_j^T}{s_j^T y_j} \right).$$

Choose

$$\begin{aligned} \gamma_k &= 1, \quad m_k = \min\{k+1, m\}, \\ P_k &= Q_k = V_{k-m_k+1,k}, \text{ and} \\ w_{ik} &= z_{ik} = \frac{(V_{k-m_k+i+1,k})^T (s_{k-m_k+i})}{\sqrt{(s_{k-m_k+i})^T (y_{k-m_k+i})}}. \end{aligned}$$

These choices fit the general form (3.1). Furthermore,

$$P_k y_j = \begin{cases} 0 & \text{if } j = k-m_k, k-m_k+1, \ldots, k, \text{ and} \\ y_j & \text{if } j = 0, 1, \ldots, k-m_k-1, \end{cases}$$

so this variation satisfies condition (3.16) of Theorem 3.1.

2. This is a special case of the next variant.

3. At iteration k, let $(\hat{s}_k^{(i)}, \hat{y}_k^{(i)})$ denote the ith ($i = 1, \ldots, m-1$) choice of any linear combination from the span of the set

$$\{(s_0, y_0), \ldots, (s_{k-1}, y_{k-1})\},$$

and let $(\hat{s}_k^{(m)}, \hat{y}_k^{(m)}) = (s_k, y_k)$. Define

$$V_{ik} = \prod_{j=i}^{m} \left(I - \frac{(\hat{y}_k^{(i)})(\hat{s}_k^{(i)})^T}{(\hat{s}_k^{(i)})^T(\hat{y}_k^{(i)})} \right).$$

Choose

$$\begin{aligned} &\gamma_k = 1, \quad m_k = \min\{k+1, m\}, \\ &P_k = Q_k = V_{1,k}, \text{ and} \\ &w_{ik} = z_{ik} = \frac{(V_{i+1,k})^T(\hat{s}_k^{(i)})}{\sqrt{(\hat{s}_k^{(i)})^T(\hat{y}_k^{(i)})}}. \end{aligned}$$

These choices satisfy the general form (3.1). Furthermore,

$$P_k y_j = \begin{cases} 0 & \text{if } y_j = y_k^{(i)} \text{ for some } i, \text{ and} \\ y_j & \text{otherwise.} \end{cases}$$

Hence, this variation satisfies condition (3.16) of Theorem 3.1.

4. Let γ_k in be the scaling constant, and choose the other vectors and matrices as in L-BFGS (3.5).

Combinations of variants are left to the reader. □

Part 3 of the previous corollary shows that the "accumulated step" method of Gill and Murray [28] terminates on quadratics.

Part 4 of the previous corollary shows that scaling does not affect termination in L-BFGS. In fact, for any method that has an update that can be expressed in the general form (3.1), it is easy to see that scaling will not affect termination on quadratics.

3.4 Examples of Methods that Do Not Reproduce the CG Iterates

We will discuss several methods that fit the general form (3.1) but do not satisfy the conditions of Theorem 3.1.

3.4.1 Steepest Descent

Steepest descent, see (3.3), does not satisfy condition (3.16) of Theorem 3.1 and thus does not produce conjugate search directions. This fact is well-known; see, e.g., Luenberger [42].

3.4.2 Limited-Memory DFP

Limited-memory DFP, see (3.7), with $m < n$ does not satisfy the condition on P_k (3.16) for all k, and so the method will not produce conjugate directions. This fact was previously unknown.

For example, suppose that we have a convex quadratic with

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}, \text{ and } b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Using a limited-memory constant of $m = 1$ and exact arithmetic, it can be seen that the iteration does not terminate within the first 20 iterations of limited-memory DFP with $H_0 = I$. The MAPLE notebook file used to compute this example is available on the World Wide Web [37].

Using the above example, we can easily see that no limited-memory Broyden Family method except limited-memory BFGS terminates within the first n iterations.

Chapter 4

Quadratic Termination of Update-Skipping Broyden Family Methods

The previous chapter discussed limited-memory methods that behave like conjugate gradients on n-dimensional strictly convex quadratic functions. In this chapter, we are concerned with methods that skip some updates. The average computation cost per iteration is reduced, and it can save memory if the quasi-Newton matrix is stored implicitly. We establish conditions under which finite termination is preserved but delayed for the Broyden Family.

4.1 Termination when Updates are Skipped

It was shown by Powell [54] that if we skip every other update and take *direct prediction steps* (i.e. steps of length one) in a Broyden Family method, then the procedure will terminate in no more than $2n+1$ iterations on an n-dimensional strictly convex quadratic function. An alternate proof of this result is given by Nazareth [46].

We will prove a related result. Suppose that we are using a perfect Broyden Family method on a strictly convex quadratic function and decide to "skip" p updates to H (i.e. choose $H_{k+1} = H_k$ on p occasions). Then, the algorithm terminates in no more than $n+p$ iterations. In contrast to Powell's result, it does not matter which updates are skipped or if multiple updates are skipped in a row.

Theorem 4.1 *Suppose that a Broyden Family method using exact line searches is applied to an n-dimensional strictly convex quadratic function*

$$f(x) = \frac{1}{2}x^T Ax - b^T x,$$

and p updates are skipped. Let

$$J(k) = \{j \le k : \text{ the update at iteration } j \text{ is not skipped}\}.$$

Then for all $k = 0, 1, \ldots$

$$g_{k+1}^T s_j = 0, \text{ for all } j \in J(k), \text{ and} \tag{4.1}$$

$$s_{k+1}^T A s_j = 0, \text{ for all } j \in J(k). \tag{4.2}$$

Furthermore, the method terminates in at most $n + p$ iterations at the exact minimizer.

Proof. We will use induction on k to show (4.1) and

$$H_{k+1} y_j = s_j, \text{ for all } j \in J(k). \tag{4.3}$$

Then (4.2) follows easily since for all $j \in J(k)$,

$$\begin{aligned} s_{k+1}^T A s_j &= -\alpha_{k+1} g_{k+1}^T H_{k+1} y_j \\ &= -\alpha_{k+1} g_{k+1}^T s_j \\ &= 0. \end{aligned}$$

Let k_0 be the least value of k such that $J(k)$ is nonempty; i.e., $J(k_0) = \{k_0\}$. Then g_{k_0+1} is orthogonal to s_{k_0} since line searches are exact, and $H_{k_0+1} y_{k_0} = s_{k_0}$ since all members of the Broyden Family satisfy the secant condition. Hence, the base case is true. Now assume that (4.1) and (4.3) hold for all values of $k = 0, 1, \ldots, \hat{k} - 1$. We will show that they also hold for $k = \hat{k}$.

Case I. Suppose that $\hat{k} \notin J(\hat{k})$. Then $H_{\hat{k}+1} = H_{\hat{k}}$ and $J(\hat{k} - 1) = J(\hat{k})$, so for any $j \in J(\hat{k})$,

$$\begin{aligned} g_{\hat{k}+1}^T s_j &= (g_{\hat{k}} + A s_{\hat{k}})^T s_j \\ &= g_{\hat{k}}^T s_j + s_{\hat{k}}^T A s_j \\ &= 0, \end{aligned} \tag{4.4}$$

and

$$H_{\hat{k}+1} y_j = H_{\hat{k}} y_j = s_j.$$

Case II. Suppose that $\hat{k} \in J(\hat{k})$. Then $H_{\hat{k}+1}$ satisfies the secant condition and $J(\hat{k}) = J(\hat{k} - 1) \cup \{\hat{k}\}$. Now $g_{\hat{k}+1}$ is orthogonal to s_k since the line searches are exact, and it is orthogonal to the older s_j by the argument in (4.4). The secant condition guarantees that $H_{\hat{k}+1} y_{\hat{k}} = s_{\hat{k}}$, and for $j \in J(\hat{k})$ but $j \neq \hat{k}$ we have

$$\begin{aligned} H_{\hat{k}+1} y_j &= H_{\hat{k}} y_j + \frac{s_{\hat{k}} s_{\hat{k}}^T}{s_{\hat{k}}^T y_{\hat{k}}} y_j - \frac{H_{\hat{k}} y_{\hat{k}} y_{\hat{k}}^T H_{\hat{k}}}{y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}}} y_j \\ &\quad + \phi \left(y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}} \right) \left(\frac{s_{\hat{k}}}{s_{\hat{k}}^T y_{\hat{k}}} - \frac{H_{\hat{k}} y_{\hat{k}}}{y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}}} \right) \left(\frac{s_{\hat{k}}}{s_{\hat{k}}^T y_{\hat{k}}} - \frac{H_{\hat{k}} y_{\hat{k}}}{y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}}} \right)^T y_j \\ &= s_j + \frac{s_{\hat{k}}^T A s_j}{s_{\hat{k}}^T y_{\hat{k}}} s_{\hat{k}} - \frac{H_{\hat{k}} y_{\hat{k}} y_{\hat{k}}^T s_j}{y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}}} \\ &\quad + \phi \left(y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}} \right) \left(\frac{s_{\hat{k}}}{s_{\hat{k}}^T y_{\hat{k}}} - \frac{H_{\hat{k}} y_{\hat{k}}}{y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}}} \right) \left(\frac{s_{\hat{k}}^T A s_j}{s_{\hat{k}}^T y_{\hat{k}}} - \frac{y_{\hat{k}}^T s_j}{y_{\hat{k}}^T H_{\hat{k}} y_{\hat{k}}} \right) \\ &= s_j. \end{aligned}$$

In either case, the induction result follows.

Suppose that we skip p updates. Then the set $J(n+p-1)$ has cardinality n. Without loss of generality, assume that the set $\{s_i\}_{i \in J(n+p-1)}$ has no zero elements. From (4.2), the vectors are linearly independent. By (4.1),

$$g_{n+p}^T s_j = 0, \text{ for all } j \in J(n-1+p),$$

and so g_{n+p} must be zero. This implies that x_{n+p} is the exact minimizer of f. $\square$

4.2 Loss of Termination for Update Skipping with Limited-Memory

Unfortunately, updates that use both limited-memory and repeated update-skipping do not produce conjugate search directions for n-dimensional strictly convex quadratics, and the termination property is lost. We will show a simple example, limited-memory BFGS with $m = 1$, skipping every other update. Note that according to Corollary 3.2, we would still be guaranteed termination if we used the most recent information in each update.

Example. Suppose that we have a convex quadratic with

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 4 \end{bmatrix}, \text{ and } b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

We apply limited-memory BFGS with limited-memory constant $m = 1$ and $H_0 = I$ and skip the update to H on even iterations. Using exact arithmetic in MAPLE, we observe that the process does not terminate even after 100 iterations [37].

Chapter 5

Experimental Results

The results of Chapters 3 and 4 lead to a number of ideas for new methods for unconstrained optimization. In this chapter, we motivate, develop, and test these ideas. We describe the collection of test problems in Section 5.2. The test environment is described in Section 5.3. Section 5.4.1 outlines the implementation of the L-BFGS method (our base for all comparisons) and Sections 5.4.2–5.4.7 describe the variations. Pseudo-code for L-BFGS and its variations is given in Appendix B. Complete numerical results, many graphs of the numerical results, and the original FORTRAN code are available [37].

5.1 Motivation

So far we have only given results for convex quadratic functions. While termination on quadratics is beautiful in theory, it does not necessarily yield insight into how these methods will do in practice.

We will not present any new results relating to convergence of these algorithms on general functions; however, many of these can be shown to converge using the convergence analysis presented in Section 7 of [41]. In [41], Liu and Nocedal show that a limited-memory BFGS method implemented with a line search that satisfies the strong Wolfe conditions (see Section 2.1 for a definition) is R-linearly convergent on a convex function that satisfies a few modest conditions.

5.2 Test Problems

For our test problems, we used the Constrained and Unconstrained Testing Environment (CUTE) by Bongartz, Conn, Gould and Toint. The package is documented in [7] and can be obtained via the World Wide Web [6] or via ftp [5]. The package contains a large collection of test problems as well as the interfaces necessary for using the problems. We chose a collection of 22 unconstrained problems. The problems ranged in size from 10 to 10,000 variables, but each took L-BFGS with limited-memory constant $m = 5$ at least 60 iterations to solve. Table 5.1 enumerates the problems, giving the SIF file name, the dimension (n),

No.	SIF Name	Size	Description & Reference
1	EXTROSNB	10	Extended Rosenbrock function (nonseparable version) [64, Problem 10].
2	WATSONS	31	Watson problem [43, Problem 20].
3	TOINTGOR	50	Toint's operations research problem [63].
4	TOINTPSP	50	Toint's PSP operations research problem [63].
5	CHNROSNB	50	Chained Rosenbrock function [63].
6	ERRINROS	50	Nonlinear problem similar to CHNROSNB [62].
7	FLETCHBV	100	Fletcher's boundary value problem [24, Problem 1].
8	FLETCHCR	100	Fletcher's chained Rosenbrock function [24, Problem 2].
9	PENALTY2	100	Second penalty problem [43, Problem 24].
10	GENROSE	500	Generalized Rosenbrock function [44, Problem 5].
11	BDQRTIC	1000	Quartic with a banded Hessian with bandwidth=9 [14, Problem 61].
12	BROYDN7D	1000	Seven diagonal variant of the Broyden tridiagonal system with a band away from diagonal [63].
13	PENALTY1	1000	First penalty problem [43, Problem 23].
14	POWER	1000	Power problem by Oren [52].
15	MSQRTALS	1024	The dense matrix square root problem by Nocedal and Liu (case 0)[9, Problem 204].
16	MSQRTBLS	1025	The dense matrix square root problem by Nocedal and Liu (case 1)[9, Problem 201].
17	CRAGGLVY	5000	Extended Cragg & Levy problem [64, Problem 32].
18	NONDQUAR	10000	Nondiagonal quartic test problem [14, Problem 57].
19	POWELLSG	10000	Extended Powell singular function [43, Problem 13].
20	SINQUAD	10000	Another function with nontrivial groups and repetitious elements [31].
21	SPMSRTLS	10000	Liu and Nocedal tridiagonal matrix square root problem [9, Problem 151].
22	TRIDIA	10000	Shanno's TRIDIA quadratic tridiagonal problem [64, Problem 8].

Table 5.1: Optimization test problem collection. Each problems was chosen from the CUTE package.

and a description for each problem. The CUTE package also provides a starting point (x_0) for each problem.

5.3 Test Environment

We used FORTRAN77 code on an SGI Indigo[2] to run the algorithms, with FORTRAN BLAS routines from NETLIB. We used the compiler's default optimization level.

Figure 2.1 outlines the general quasi-Newton implementation that we followed. For the line search, we use the routines `cvsrch` and `cstep` written by Jorge J. Moré and David Thuente from a 1983 version of `MINPACK`. The line search finds an α that meets the strong Wolfe conditions (2.2) and (2.4). We used $\omega_1 = 1.0 \times 10^{-4}$ and $\omega_2 = 0.9$. Except for the first iteration, we always attempt a step length of 1.0 first and only use an alternate value if 1.0 does not satisfy the Wolfe conditions. In the first iteration, we initially try a step length equal to $\|g_0\|^{-1}$. The remaining line search parameters are detailed in Appendix A.

We generate the matrix H_k by either the limited-memory update or one of the variations described in Section 5.4, storing the matrix implicitly in order to save both memory and computation time.

We terminate the iterations if any of the following conditions are met at iteration k:

1. The inequality
$$\|g_k\| < 1.0 \times 10^{-5} \cdot \max\{1, \|x_k\|\},$$
is satisfied,

2. the line search fails due to rounding errors, or

3. the number of iterations exceeds 3000.

We say that the iterates have converged if the first condition is satisfied. Otherwise, the method has failed.

5.4 L-BFGS and Its Variations

We tried a number of variations to the standard L-BFGS algorithm. L-BFGS and these variations are described in this section and summarized in Tables 5.2–5.3.

5.4.1 L-BFGS: Algorithm 0

The limited-memory BFGS update is given in (2.7) and described fully by Nocedal [47]. Our implementation and the following description come essentially from Byrd, Nocedal and Schnabel [12].

Let H_0 be symmetric and positive definite and assume that the m_k pairs

$$\{s_i, y_i\}_{i=k-m_k}^{k-1}$$

No.	**Reference**	**Brief Description**
0	§ 5.4.1	L-BFGS with no options.
1	§ 5.4.2, Variation 1	Allow m to vary iteratively basing the choice of m of $\|g\|$ and not allowing m to decrease.
2	§ 5.4.2, Variation 2	Allow m to vary iteratively basing the choice of m of $\|g\|$ and allowing m to decrease.
3	§ 5.4.2, Variation 3	Allow m to vary iteratively basing the choice of m of $\|g/x\|$ and not allowing m to decrease.
4	§ 5.4.2, Variation 4	Allow m to vary iteratively basing the choice of m of $\|g/x\|$ and allowing m to decrease.
5	§ 5.4.3	Dispose of old information if the step length is greater than one.
6	§ 5.4.4, Variation 1	Back-up if the current iteration is odd.
7	§ 5.4.4, Variation 2	Back-up if the current iteration is even.
8	§ 5.4.4, Variation 3	Back-up if a step length of 1.0 was used in the last iteration.
9	§ 5.4.4, Variation 4	Back-up if $\|g_k\| > \|g_{k-1}\|$.
10	§ 5.4.4, Variation 3*	Back-up if a step length of 1.0 was used in the last iteration and we did not back-up on the last iteration.
11	§ 5.4.4, Variation 4*	Back-up if $\|g_k\| > \|g_{k-1}\|$ and we did not back-up on the last iteration.

Table 5.2: Description of Numerical Optimization Algorithms (Part I)

No.	Reference	Brief Description
12	§ 5.4.5, Variation 1	Merge if neither of the two vectors to be merged is itself the result of a merge and the 2nd and 3rd most recent steps taken were of length 1.0.
13	§ 5.4.5, Variation 2	Merge if we did not do a merge the last iteration and there are at least two old s vectors to merge.
14	§ 5.4.6, Variation 1	Skip update on odd iterations.
15	§ 5.4.6, Variation 2	Skip update on even iterations.
16	§ 5.4.6, Variation 3	Skip update if $\|g_{k+1}\| > \|g_k\|$.
17	Alg. 5 & Alg. 8	Dispose of old information and back-up on the next iteration if the step length is greater than one.
18	Alg. 13 & Alg. 1	Merge if we did not do a merge the last iteration and there are at least two old s vectors to merge, and allow m to vary iteratively basing the choice of m of $\|g\|$ and not allowing m to decrease.
19	Alg. 13 & Alg. 3	Merge if we did not do a merge the last iteration and there are at least two old s vectors to merge, and allow m to vary iteratively basing the choice of m of $\|g/x\|$ and not allowing m to decrease.
20	Alg. 13 & Alg. 2	Merge if we did not do a merge the last iteration and there are at least two old s vectors to merge, and allow m to vary iteratively basing the choice of m of $\|g\|$ and allowing m to decrease.
21	Alg. 13 & Alg. 4	Merge if we did not do a merge the last iteration and there are at least two old s vectors to merge, and allow m to vary iteratively basing the choice of m of $\|g/x\|$ and allowing m to decrease.

Table 5.3: Description of Numerical Optimization Algorithms (Part II)

each satisfy $s_i^T y_i > 0$.

We will let

$$S_k = [s_{k-m_k} s_{k-m_k+1} \cdots s_{k-1}] \text{ and } Y_k = [y_{k-m_k} y_{k-m_k+1} \cdots y_{k-1}],$$

where $m_k = \min\{k+1, m\}$, and m is some positive integer. We will assume that $H_0 = I$ and that H_0 is iteratively rescaled by a constant γ_k as is commonly done in practice. Then, the matrix H_k obtained by k applications of the limited-memory BFGS update can be expressed as

$$H_k = \gamma_k I + \begin{pmatrix} S_k & \gamma_k Y_k \end{pmatrix} \begin{pmatrix} U_k^{-T}(D_k + \gamma_k Y_k^T Y_k) U_k^{-1} & -U_k^{-T} \\ -U_k^{-1} & 0 \end{pmatrix} \begin{pmatrix} S_k^T \\ \gamma_k Y_k^T \end{pmatrix},$$

where U_k and D_k are the $m_k \times m_k$ matrices given by

$$(U_k)_{ij} = \begin{cases} {s_{k-m_k-1+i}}^T y_{k-m_k-1+j} & \text{if } i \leq j, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$D_k = \mathrm{diag}\{s_{k-m_k}^T y_{k-m_k}, \ldots, s_{k-1}^T y_{k-1}\}.$$

We will describe how to compute $d_k = -H_k g_k$ in the case that $k > 0$. Let x_k be the current iterate. Let $m_k = \min\{k+1, m\}$. Given s_{k-1}, y_{k-1}, g_k, the matrices $S_{k-1}, Y_{k-1}, U_{k-1}, Y_{k-1}^T Y_{k-1}, D_{k-1}$, and the vectors $S_{k-1}^T g_{k-1}, Y_{k-1}^T g_{k-1}$:

1. Update the $n \times m_{k-1}$ matrices S_{k-1} and Y_{k-1} to get the $n \times m_k$ matrices S_k and Y_k using s_{k-1} and y_{k-1}

2. Compute the m_k-vectors $S_k^T g_k$ and $Y_k^T g_k$.

3. Compute the m_k-vectors $S_k^T y_{k-1}$ and $Y_k^T y_{k-1}$ by using the fact that

$$y_{k-1} = g_k - g_{k-1}.$$

   We already know $m_k - 1$ components of $S_k g_{k-1}$ from $S_{k-1} g_{k-1}$, and likewise for $Y_k g_{k-1}$. We need only compute $s_{k-1}^T g_{k-1}$ and $y_{k-1}^T g_{k-1}$ and do the subtractions.

4. Compute U_k^{-1}. Rather than recomputing U_k^{-1}, we update the matrix from the previous iteration by deleting the leftmost column and topmost row if $m_k = m_{k-1}$ and appending a new column on the right and a new row on the bottom. Let $\rho_{k-1} = 1/s_{k-1}^T y_{k-1}$ and let $(U_{k-1}^{-1})'$ be the $(m_k - 1) \times (m_k - 1)$ lower right submatrix of U_{k-1}^{-1} and let $(S_k^T y_{k-1})'$ be the upper $m_k - 1$ elements of $S_k^T y_{k-1}$. Then

$$U_k^{-1} = \begin{pmatrix} (U_{k-1}^{-1})' & -\rho_{k-1}(U_{k-1}^{-1})'(S_k^T y_{k-1})' \\ 0 & \rho_{k-1} \end{pmatrix}.$$

   Note that $s_{k-1}^T y_{k-1} = (S_k^T y_{k-1})_{m_k}$ and so is already computed.

5. Assemble $Y_k^T Y_k$. We have already computed all the components.

6. Update D_k using D_{k-1} and $s_{k-1}^T y_{k-1} = (S_k^T y_{k-1})_{m_k}$.

7. Compute
$$\gamma_k = y_{k-1}^T s_{k-1} / y_{k-1}^T y_{k-1}.$$
Note that both $y_{k-1}^T s_{k-1}$ and $y_{k-1}^T y_{k-1}$ have already been computed.

8. Compute two intermediate values
$$\begin{aligned} p_1 &= U_k^{-1} S_k^T g_k, \\ p_2 &= U_k^{-1}(\gamma_k Y_k^T Y_k p_1 + D_k p_1 - \gamma_k Y_k^T g_k). \end{aligned}$$

9. Compute
$$d_k = \gamma_k Y_k p_1 - S_k p_2 - \gamma_k g_k.$$

The storage costs for this are very low. In order to reconstruct H_k, we need to store $S_k, Y_k, U_k^{-1}, Y_k^T Y_k$, D_k (a diagonal matrix) and a few m-vectors. This requires only $2mn + 2m^2 + O(m)$ storage. Assuming $m << n$, this is much less storage than the n^2 storage required for a typical implementation of BFGS.

Step	**Operation Count**
2	$4mn - 2m$
3	$4n + 2m - 2$
4	$2m^2 - 4m + 3$
7	1
8	$8m^2 + 2m$
9	$4m^2 + 2m$

Table 5.4: Operations count for computation of $H_k g_k$. Steps with no operations are not shown.

The computation of Hg takes at most $O(mn)$ operations assuming $n >> m$. (See Table 5.4.) This is much less than the $O(n^2)$ time normally needed to compute Hg when the whole matrix H is stored.

We are using L-BFGS as our basis for comparison. For information on the performance of L-BFGS see Liu and Nocedal [41] and Nash and Nocedal [45].

Alg. No.	m = 5	m = 10	m = 15	m = 50
0	1	0	0	1
1	0	0	0	0
2	1	0	0	0
3	2	0	0	1
4	1	0	0	1
5	0	0	0	0
6	1	0	0	1
7	0	0	0	1
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0
11	0	0	0	0
12	1	0	0	1
13	1	0	0	1
14	12	12	12	12
15	5	5	5	5
16	11	11	9	10
17	0	0	0	0
18	1	1	0	0
19	1	0	0	1
20	1	1	1	1
21	3	1	0	1

Table 5.5: The number of failures of the algorithms on the 22 test problems. An algorithm is said to have "failed" on a particular problem if a line search fails or the maximum allowable number of iterations (3000 in our case) is exceeded.

5.4.2 Varying m Iteratively: Algorithms 1–4

In typical implementations of L-BFGS, m is fixed throughout the iterations: once m updates have accumulated, m updates are always used. We considered the possibility of varying m iteratively, preserving finite termination on convex quadratics. Using an argument similar to that presented in [41], we can also prove that this algorithm has a linear rate of convergence on a convex function that satisfies a few modest conditions.

We tried four different variations on this theme. All were based on a linear formula that scales m in relation to the size of $\|g\|$. The motivation is that we will need a stronger model using more information as $\|g\|$ gets smaller and we are closer to the minimizer; far away we require less information. Let m_k be the number of iterates saved at the kth iteration, with $m_0 = 1$. Here, think of m as the maximum allowable value of m_k. Let the convergence test

Alg. No.	m = 5	m = 10	m = 15	m = 50
1	8/22	10/22	17/22	17/22
2	7/22	13/22	13/22	19/22
3	14/21	14/22	12/22	15/21
4	12/21	17/22	15/22	16/21
5	19/22	20/22	20/22	21/22
6	21/21	22/22	22/22	21/21
7	8/22	12/22	10/22	10/22
8	12/22	14/22	12/22	15/22
9	6/22	13/22	12/22	16/22
10	12/22	14/22	12/22	15/22
11	10/22	10/22	11/22	14/22
12	21/21	22/22	22/22	21/21
13	3/22	4/22	4/22	4/22
14	1/21	1/22	1/22	1/21
15	1/22	1/22	1/22	0/22
16	0/22	1/22	1/22	0/22
17	12/22	13/22	12/22	14/22
18	3/22	4/22	5/22	4/22
19	2/22	3/22	4/22	4/22
20	2/22	4/22	4/22	5/22
21	1/22	2/22	4/22	4/22

Table 5.6: Function Evaluations Comparison. The first number in each entry is the number of times the algorithm did as well as or better than normal L-BFGS in terms of function evaluations. The second number is the total number of problems solved by at least one of the two methods (the algorithm and/or L-BFGS).

be given by $\|g_k\|/\max\{1, \|x_k\|\} < \epsilon$. Then the formula for m_k at iteration k is

$$m_k = \min\left\{m_{k-1} + 1, \left\lfloor (m-1)\frac{\log \delta_k - \log \delta_0}{\log 100\epsilon - \log \delta_0}\right\rfloor + 1\right\}.$$

We can specify δ_k to be $\|g_k\|$ without any regard to normalization, or we can normalize the gradient with respect to $\|x_k\|$. (If $\|x_k\| < 1$, we normalize by 1.) The choice may depend on the sensitivity of the gradient to scale. We begin with $m_1 = 1$ and then compute m_k each time based on the formula above. We may, however, want to restrict m_k so it cannot go below the previous value, m_{k-1}. The four variations are

1. $\delta_k = \|g_k\|$ and require $m_k \geq m_{k-1}$,
2. $\delta_k = \|g_k\|$,

Alg. No.	m = 5	m = 10	m = 15	m = 50
1	15/22	18/22	20/22	18/22
2	16/22	19/22	18/22	18/22
3	16/21	14/22	15/22	15/21
4	17/21	18/22	20/22	18/21
5	15/22	13/22	14/22	15/22
6	16/21	19/22	15/22	15/21
7	11/22	11/22	10/22	7/22
8	11/22	7/22	6/22	5/22
9	9/22	10/22	7/22	8/22
10	11/22	8/22	5/22	5/22
11	9/22	8/22	9/22	5/22
12	11/21	12/22	8/22	11/21
13	5/22	10/22	13/22	17/22
14	1/21	1/22	1/22	2/21
15	5/22	6/22	9/22	9/22
16	0/22	2/22	3/22	2/22
17	11/22	8/22	5/22	4/22
18	8/22	14/22	19/22	20/22
19	11/22	11/22	17/22	19/22
20	10/22	14/22	17/22	19/22
21	9/22	16/22	16/22	18/22

Table 5.7: Time Comparison. The first number in each entry is the number of times the algorithm did as well as or better than normal L-BFGS in terms of time. The second number is the total number of problems solved by at least one of the two methods (the algorithm and/or L-BFGS).

3. $\delta_k = \|g_k\|/\max\{1, \|x_k\|\}$ and require $m_k \geq m_{k-1}$, and
4. $\delta_k = \|g_k\|/\max\{1, \|x_k\|\}$.

We used four values of m: 5, 10, 15 and 50, for each algorithm. The results are summarized in Tables 5.5 – 5.9. More extensive results can be obtained [37].

Table 5.5 shows that these algorithms had roughly the same number of failures as L-BFGS.

Table 5.6 compares each algorithm to L-BFGS in terms of function evaluations. For each algorithm and each value of m, the number of times that the algorithm used *as few or fewer* function evaluations than L-BFGS is listed relative to the total number of admissible problems. Problems are admissible if at least one of the two methods solved it. We observe that in all but three cases, the algorithm used as few or fewer function evaluations than L-BFGS for over half the test problems.

Alg. No.	m = 5	m = 10	m = 15	m = 50
1	1.054	1.017	0.931	1.008
2	1.099	0.976	0.968	0.945
3	1.006	0.957	1.391	1.014
4	0.998	1.297	0.970	1.000
5	1.021	0.971	1.005	1.010
6	1.000	1.000	1.000	1.000
7	1.099	0.996	1.205	1.020
8	0.991	1.677	1.507	0.891
9	1.035	1.371	1.005	0.947
10	0.991	1.677	1.507	0.891
11	1.044	0.992	0.981	0.916
12	1.000	1.000	1.000	1.000
13	1.137	1.178	1.244	1.373
14	8.227	8.666	9.073	9.308
15	3.185	4.754	5.317	6.032
16	9.687	5.926	6.032	7.054
17	0.981	1.023	0.924	0.918
18	1.201	1.529	1.209	1.365
19	1.212	1.959	1.242	1.387
20	1.263	1.101	1.226	1.375
21	1.406	1.161	1.178	1.394

Table 5.8: Mean function evaluations ratios for each algorithm compared to L-BFGS. Problems for which either method failed are not used in this mean.

Table 5.7 compares each algorithm to L-BFGS in terms of time. The entries are similar to those in Table 5.6. Observe that Algorithms 1-4 did very well in terms of time, doing as well or better than L-BFGS in nearly every case.

For each problem in each algorithm, we computed the ratio of the number of function evaluations for the algorithm to the number of function evaluations for L-BFGS. Table 5.8 lists the means of these ratios. A mean below 1.0 implies that the algorithm does better than L-BFGS on average. The average is better for the first four algorithms in 6 out of 16 cases. Observe, however, that all the means are close to one.

We experience savings in terms of time for the first four algorithms. These algorithms will tend to save fewer vectors than L-BFGS since m_k is typically less than m; and so less work is done computing $H_k g_k$ in these algorithms. Table 5.9 gives the mean of the ratios of time to solve for each value of m in each algorithm. Note that most of the ratios are far below one in this case.

These variations did particularly well on problem 7. See [37] for more information.

Alg. No.	m = 5	m = 10	m = 15	m = 50
1	0.972	0.894	0.784	0.884
2	0.993	0.831	0.780	0.783
3	0.955	0.870	1.071	0.898
4	0.907	1.119	0.823	0.856
5	1.041	0.969	0.993	1.004
6	1.007	0.983	0.977	0.995
7	1.088	1.010	1.179	1.692
8	1.057	1.421	1.426	1.425
9	1.032	1.220	1.043	1.173
10	1.056	1.405	1.440	1.412
11	1.062	1.050	1.062	1.208
12	1.008	1.011	1.013	1.002
13	1.083	1.082	0.983	0.960
14	5.008	4.046	3.527	2.646
15	2.079	2.507	2.579	2.735
16	9.272	5.736	5.129	6.407
17	1.053	1.166	1.089	1.399
18	1.081	1.229	0.860	0.885
19	1.130	1.423	0.915	0.923
20	1.114	0.867	0.837	0.916
21	1.258	0.927	0.859	0.974

Table 5.9: Mean time ratios for each algorithm compared to L-BFGS. Problems for which either method failed are not used in this mean.

5.4.3 Disposing of Old Information: Algorithm 5

We may decide that we are storing too much old information and that we should stop using it. For example, we may choose to throw away everything except for the most recent information whenever we take a big step, since the old information may not be relevant to the new neighborhood. We use the following test: If the last step length was bigger than 1, dispose of the old information.

The algorithm performed nearly the same as L-BFGS. There was substantial deviation on only one or two problems for each value of m, and this seemed evenly divided in terms of better and worse. From Table 5.5, we see that this algorithm successfully converged on every problem. Table 5.6 shows that it almost always did as well or better than L-BFGS in terms of function evaluations. However, Table 5.8 shows that the differences were minor. In terms of time, we observe that the algorithm generally was faster than L-BFGS (Table 5.7), but again, considering the mean ratios of time (Table 5.9), the differences were minor. The method also does particularly well on problem 7 [37].

5.4.4 Backing Up in the Update to H: Algorithms 6-11

As discussed in Section 3.3, if we always use the most recent s and y in the update, we preserve quadratic termination regardless of which older values of s and y we use.

Using this idea, we created some algorithms. Under certain conditions, we discard the next most recent values of s and y in the H although we still use the *most* recent s and y vectors and any other vectors that have been saved from previous iterations. We call this "backing up" because it is as if we back-up over the next most recent values of s and y. These algorithms used the following four tests to trigger backing up:

1. The current iteration is odd.

2. The current iteration is even.

3. A step length of 1.0 was used in the last iteration.

4. $\|g_k\| > \|g_{k-1}\|$.

In two additional algorithms, we varied situations 3 and 4 by not allowing a back-up if a back-up was performed on the previous iteration.

The backing up strategy seemed robust in terms of failures. In 4 out of the 6 variations we did for this algorithm, there were no failures at all. See Table 5.5 for more information.

It is interesting to observe that backing up on odd iterations (Algorithm 6) and backing up on even iterations (Algorithm 7) caused very different results. Backing up on odd iterations seemed to have almost no effect on the number of function evaluations (Table 5.8) and little effect on the time (Table 5.9). However, backing up on even iterations causes much different behavior from L-BFGS. It does worse than L-BFGS on most problems, but better on a few.

Algorithms 8 and 10 were two variations of the same idea: backing up if the previous step length was one. This wipes out the data from the previous iteration after it has been used in one update. Both show improvement over L-BFGS in terms of function evaluations; in fact, these two algorithms have the best function evaluation ratio for the $m = 50$ case (Table 5.8). Unfortunately, these algorithms did not compete with L-BFGS in terms of time (Table 5.9). There is little difference between Algorithms 8 and 10 — probably because there were rarely two steps of length one is a row.

Algorithms 9 and 11 are also two variations of the same idea: back-up on iteration $k+1$ if the norm of g_k is bigger than the norm of g_{k+1}. There is a larger difference between the results of 9 and 11 than there was between 8 and 10. In terms of function evaluation ratios (Table 5.8), Algorithm 11 did better, indicating that it may not be wise to back-up twice in a row. Both of these did poorly in terms of time as compared with L-BFGS (Table 5.9).

5.4.5 Merging s and y Information in the Update: Algorithms 12 and 13

Yet another idea is to "merge" s data so that it takes up less storage and computation time. By merging, we mean forming some linear combination of various s vectors. The y vectors would be merged correspondingly. Corollary 3.3 shows that as long as the most recent s and y are used without merge, old s vectors may be replaced by any linear combination of the old s vectors in L-BFGS.

We used this idea in the following way: if certain criteria were met, we replaced the second and third newest s vectors in the collection by their sum, and did similarly for the y vectors. We used various tests to determine when we would do a merge:

1. Neither of the two vectors to be merged is itself the result of a merge and the second and third most recent steps taken were of length 1.0.

2. We did not do a merge the last iteration and there are at least two old s vectors to merge.

The first variation (Algorithm 12) performs almost identically to L-BFGS, especially in terms of time (Table 5.6). Occasionally it did worse in terms of time (Table 5.7). These observations are also reflected in the other results in Table 5.8 and Table 5.9. It is likely that very few vectors were merged.

The second variation (Algorithm 13) makes gains in terms of time, especially for the larger values of m (Table 5.7 and Table 5.9). Unfortunately, this reflects only a saving in the amount of linear algebra required. The number of function evaluations generally is larger for this algorithm than L-BFGS (Table 5.6 and Table 5.8).

5.4.6 Skipping Updates to H: Algorithms 14–16

If every other update to H is skipped and a step length of one is always chosen, BFGS will terminate in $2n + 1$ iterations on a strictly convex quadratic function. Similarily, if every other update to H is skipped and an exact line search is used, BFGS will terminate in $2n$ iterations on a strictly convex quadratic function. (See Chapter 4.) Unfortunately, neither property holds in the limited-memory case. We will, however, try some algorithms motivated by this idea.

The idea is that, every so often, we do not use the current s and y to update H, and instead just use the old H. There are three variations on this theme.

1. Skip update on odd iterations.

2. Skip update on even iterations.

3. Skip update if $\|g_{k+1}\| > \|g_k\|$.

As with the algorithms that did back-ups, the results of the skipping on odd or even iterations were quite different. Skipping on odd updates (Algorithm 14) did extremely well for every value of m only on problem 1. Otherwise, it did very badly. Skipping on even updates (Algorithm 15) performed somewhat better. It did extremely well on problem 7 but not on problems 1 and 12. It also did better than L-BFGS in terms of time on more occasions than Algorithm 14 (Table 5.7). Neither did well in terms of function evaluations, but the mean ratios for function evaluations (Table 5.8) and time (Table 5.9) were usually far greater than one.

Skipping the update if the norm of g increased (Algorithm 16) did not do well at all. It only did better in terms of function evaluations for one problem for two values of m (Table 5.6) and rarely did better in terms of time (Table 5.7). Its ratios were very bad for function evaluations (Table 5.8) and time (Table 5.9)

5.4.7 Combined Methods: Algorithms 17-21

We did some experimentation with combinations of methods described in the previous sections.

In Algorithm 17, we combined Algorithms 5 and 8: we dispose of old information and back-up on the next iterations if the step length is greater than one. Essentially we are assuming that we have stepped out of the region being modeled by the quasi-Newton matrix if we take a long step and we should thus rid the quasi-Newton matrix of that information. This algorithm did well in terms of function evaluations, having mean ratios of less than one for three values of m (Table 5.8), but it did not do as well in terms of time.

In Algorithms 19-21, we combined merging and varying m. These algorithms did well in terms of time for larger m (Table 5.9) but not in terms of function evaluations (Table 5.8).

Chapter 6

A Semi-Discrete Matrix Decomposition

The semi-discrete decomposition (SDD) was first proposed for image compression by O'Leary and Peleg [51]. In this chapter we will review the decomposition and the algorithm for generating it. Furthermore, we will present some new convergence results and also give some idea of how this decomposition relates to the singular value decomposition. In Chapter 7, we will describe information retrieval, and in Chapter 8 we will show how the SDD can be used in information retrieval.

6.1 Introduction

The O'Leary-Peleg idea [51] is to find a matrix approximation of the form

$$A_k = \underbrace{\begin{bmatrix} x_1 & x_2 & \cdots & x_k \end{bmatrix}}_{X_k} \underbrace{\begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_k \end{bmatrix}}_{D_k} \underbrace{\begin{bmatrix} y_1^T \\ y_2^T \\ \vdots \\ y_k^T \end{bmatrix}}_{Y_k^T} = \sum_{i=1}^{k} d_i x_i y_i^T,$$

to an $m \times n$ matrix A. Here each x_i is an m-vector with entries from the set $\mathcal{S} = \{-1, 0, 1\}$, each y_i is an n-vector with entries from the set $\mathcal{S}$, and each d_i is a positive scalar. We call this the *semi-discrete decomposition* (SDD) of *rank* k.[1]

The SDD approximation is formed iteratively. The remainder of this section comes from [51] but is presented here in a slightly different form. Let $A_0 = 0$, and let R_k be the residual matrix at the kth step, that is, $R_k = A - A_{k-1}$. We wish to find a triplet (d_k, x_k, y_k) that solves

$$\min_{\substack{x \in \mathcal{S}^m \\ y \in \mathcal{S}^n \\ d > 0}} F_k(d, x, y) \equiv \| R_k - dxy^T \|_F^2. \tag{6.1}$$

[1]This matrix may not be rank-k *algebraically*, but it is formed as the sum of k rank-1 matrices.

This is a mixed integer programming problem.

We can formulate this as an integer programming problem by eliminating d. We have

$$\begin{aligned} F(d,x,y) &= \sum_{i=1}^{m}\sum_{j=1}^{n}(r_{ij} - dx_iy_j)^2 \\ &= \sum_{i=1}^{m}\sum_{j=1}^{n} r_{ij}^2 - 2dx_ir_{ij}y_j + d^2x_i^2y_j^2 \\ &= \|R\|_F^2 - 2dx^TRy + d^2\|x\|_2^2\|y\|_2^2, \end{aligned}$$

where for convenience, we have dropped the subscript k. At the optimal solution,

$$\partial F/\partial d = -2x^TRy + 2d\|x\|_2^2\|y\|_2^2 = 0,$$

so the optimal value of d is given by

$$d^* = \frac{x^TRy}{\|x\|_2^2\|y\|_2^2}.$$

Plugging d^* into F, we get

$$\begin{aligned} F(d^*,x,y) &= \|R\|_F^2 - 2\left(\frac{x^TRy}{\|x\|_2^2\|y\|_2^2}\right)x^TRy + \left(\frac{x^TRy}{\|x\|_2^2\|y\|_2^2}\right)^2\|x\|_2^2\|y\|_2^2 \\ &= \|R\|_F^2 - \frac{(x^TRy)^2}{\|x\|_2^2\|y\|_2^2}. \end{aligned} \tag{6.2}$$

Thus (6.1) is equivalent to

$$\max_{\substack{x\in\mathcal{S}^m \\ y\in\mathcal{S}^n}} \tilde{F}(x,y) \equiv \frac{(x^TRy)^2}{\|x\|_2^2\|y\|_2^2}, \tag{6.3}$$

which is an integer programming problem with $3^{(m+n)}$ feasible points.

For small values of both m and n, we can compute the value of $\tilde{F}$ at each feasible point to determine the minimizer. However, as the size of m and/or n grows, the cost of this approach grows exponentially. We know of no better algorithm for solving this problem exactly, so we do not. Instead we use an *alternating algorithm* to generate an approximate solution. We begin by fixing y and solving (6.3) for x, we then fix that x and solve (6.3) for y, we then fix that y and solve (6.3) for x, and so on.

Solving (6.3) is very easy when either x or y is fixed. Suppose that y is fixed. Then we must solve

$$\max_{x\in\mathcal{S}^m} \frac{(x^Ts)^2}{\|x\|_2^2}, \tag{6.4}$$

where $s = Ry/\|y\|_2^2$ is fixed. Sort the elements of s so that

$$|s_{i_1}| \geq |s_{i_2}| \geq \cdots \geq |s_{i_m}|.$$

If we knew x had exactly J nonzeros, then it is clear that the solution to (6.4) would be given by

$$x_{i_j} = \begin{cases} \text{sign}(s_{i_j}) & \text{if } 1 \le j \le J \\ 0 & \text{if } J+1 \le j \le m \end{cases} .$$

Therefore, there are only m possible x-vectors we need to check to determine the optimal solution for (6.4).

Let $R_1 = A$.

Outer Iteration ($k = 1, 2, \ldots, k_{\max}$):

  Choose starting vector y such that $R_k y \neq 0$.

  Inner Iteration ($i = 1, 2, \ldots, i_{\max}$):

    Fix y and let x solve $\displaystyle\max_{x \in \mathcal{S}^m} \frac{x^T R_k y}{\|x\|_2^2}$.

    Fix x and let y solve $\displaystyle\max_{y \in \mathcal{S}^n} \frac{y^T R_k^T x}{\|y\|_2^2}$.

  End Inner Iteration.

  Let $x_k = x$, $y_k = y$, $d_k = \dfrac{x_k^T R_k y_k}{\|x_k\|_2^2 \|y_k\|_2^2}$.

  Let $R_{k+1} = R_k - d_k x_k y_k^T$.

End Outer Iteration.

Figure 6.1: O'Leary-Peleg Algorithm

Figure 6.1 shows the O'Leary-Peleg algorithm to find the SDD approximation of rank-$k_{\max}$ to an $m \times n$ matrix A. We specify a set number of iterations for the inner loop, but we may use a heuristic stopping criterion instead. From (6.2) note that

$$\begin{aligned} \|R_{k+1}\|_F^2 &= \|R_k - d_k x_k y_k^T\|_F^2 \\ &= \|R_k\|_F^2 - \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2}. \end{aligned} \tag{6.5}$$

So for a given (x, y) pair, we can predict exactly what the F-norm of R_{k+1} will be if we accept them. The method to determine when to stop the inner iterations proposed in [51] is the following: At the beginning of the inner iterations, set `change = 1`. Then at the end of each inner iteration, compute

$$\texttt{newchange} = \frac{(x^T R_k y)^2}{\|x\|_2^2 \|y\|_2^2}, \text{ and}$$

$$
\begin{aligned}
\texttt{improvement} &= \frac{|\texttt{newchange} - \texttt{change}|}{\texttt{change}}, \\
\texttt{change} &= \texttt{newchange}.
\end{aligned}
$$

Once `improvement` falls below a given level, say 0.01, we terminate the inner iterations. In other words, we iterate until the improvement in the residual has stagnated.

6.2 Reconstruction via the SDD

We will show that the algorithm can exactly reproduce a rank-1 matrix of the form $\hat{d}\hat{x}\hat{y}^T$ where $\hat{d} > 0$, $\hat{x} \in \mathcal{S}^m$, and $\hat{y} \in \mathcal{S}^n$, but it does depend on a fine point that will be discussed in the proof.

Proposition 6.1 *Let $A = \hat{d}\hat{x}\hat{y}^T$ where $\hat{d} > 0$, $\hat{x} \in \mathcal{S}^m$, and $\hat{y} \in \mathcal{S}^n$. Then the O'Leary-Peleg algorithm can find $\hat{d}$, $\hat{x}$ and $\hat{y}$ in exactly one inner iteration.*

Proof. Choose y such that $Ay \neq 0$. The first step of the inner iteration is to find $x \in \mathcal{S}^m$ that maximizes

$$\frac{x^T Ay}{\|x\|_2^2} = (\hat{d}\hat{y}^T y)\frac{x^T \hat{x}}{\|x\|_2^2}$$

If $\hat{x}$ has more than one non-zero element, then the maximizer is not unique. However, if we always break the tie by choosing the candidate with the largest number of nonzero elements, then we will get $x = \hat{x}$. A similar argument applies to the second step, and we get $y = \hat{y}$. Lastly,

$$d = \frac{x^T Ay}{\|x\|_2^2 \|y\|_2^2} = \hat{d}\frac{(\hat{x}^T\hat{x})(\hat{y}^T\hat{y})}{\|\hat{x}\|_2^2\|\hat{y}\|_2^2} = \hat{d}. \quad \Box$$

6.3 Convergence of the SDD

Next we will focus on the convergence of the semi-discrete approximation. First we will show that the norm of the residual generated by the O'Leary-Peleg algorithm is strictly decreasing. Then we will show that under certain circumstances, the approximation generated by the O'Leary-Peleg algorithm converges linearly to the true matrix.

Proposition 6.2 *The residual matrices generated by the O'Leary-Peleg Algorithm satisfy*

$$\|R_{k+1}\|_F < \|R_k\|_F \textit{ for all } k.$$

Proof. Assume that $R_k \neq 0$. (Otherwise the algorithm has terminated at the exact solution.) At the end of the inner iterations, we are guaranteed to have found x_k and y_k such that $x_k^T R_k y_k > 0$. Using (6.5) and

$$0 < \frac{(x_k^T R_k y_k)^2}{\|x\|_2^2\|y\|_2^2} \leq (x_k^T R_k y_k)^2 \leq \|R_k\|_F^2,$$

we know that $\|R_k\|_F^2 > \|R_{k+1}\|_F^2 \geq 0$. $\square$

Now we will discuss several strategies for initializing the inner loop in the O'Leary-Peleg algorithm and give convergence results for each.

Initialization Choice 1 (Max Element) *At iteration k, assume $R_k = [r_{ij}]$. Choose $y = e_{\hat{j}}$ where $|r_{\hat{i}\hat{j}}| = \max_{ij} |r_{ij}|$.*

Theorem 6.1 *If we use initialization choice 1, then the sequence $\{A_k\}$ generated by the O'Leary-Peleg algorithm converges to A in the Frobenius norm. Furthermore, the rate of convergence is at least linear.*

Proof. Without loss of generality, assume that $R_k \neq 0$ for all k. (Otherwise the algorithm has terminated at the exact solution.) Let ρ_k denote $\|R_k\|_F^2$. Then $\rho_k \neq 0$ for all k, and by Proposition 6.2 we know that the sequence $\{\rho_k\}$ is strictly decreasing; furthermore, the sequence is bounded below by zero. Thus, the sequence must converge to a limit point, say, ρ^*. If $\rho^* = 0$, then the proof is complete, so assume $\rho^* \neq 0$. Then

$$\rho_k \geq \rho^* > 0 \text{ for all } k. \tag{6.6}$$

Set $\epsilon = \frac{\rho^*}{mn}$. By definition of convergence, there exists $\hat{k}(\epsilon)$ such that $\rho_k < \rho^* + \epsilon$ for all $k > \hat{k}(\epsilon)$. Fix $k > \hat{k}(\epsilon)$. Let $[r_{ij}] = R_k$. Choose an initial $y = e_{\hat{j}}$ where $|r_{\hat{i}\hat{j}}| = \max_{ij} |r_{ij}|$. Since we know the first part of the inner iteration picks the optimal x, it must be as least as good as choosing $x = e_{\hat{i}}$, so

$$\frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} \geq \frac{(e_{\hat{i}}^T R_k e_{\hat{j}})^2}{\|e_{\hat{i}}\|_2^2 \|e_{\hat{j}}\|_2^2} \geq |r_{\hat{i}\hat{j}}| \geq \frac{\|R_k\|_F^2}{mn} = \frac{\rho_k}{mn} > \frac{\rho^*}{mn} = \epsilon \tag{6.7}$$

Thus

$$\rho_{k+1} = \rho_k - \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} < (\rho^* + \epsilon) - \epsilon = \rho^*.$$

But this contradicts (6.6), so we conclude that $\rho^* = 0$. Hence, $A_k \to A$.

Now consider the rate of convergence. Using (6.5) and (6.7) we get

$$\rho_{k+1} \leq \rho_k - \frac{\rho_k}{mn} = \left(1 - \frac{1}{mn}\right) \rho_k \leq \left(1 - \frac{1}{mn}\right)^k \rho_0.$$

Hence the rate of convergence is at least linear. $\square$

We will construct an example to show further that the rate of convergence is at most linear.

The next proposition shows that, for any n, we can construct an $n \times n$ matrix such that the norm of the residual generated by the O'Leary-Peleg algorithm decreases at a constant rate for the first $n - 1$ steps.

Proposition 6.3 *Using initialization choice 1, for any n, we can construct a matrix of size n such that the residual matrices generated by the O'Leary-Peleg algorithm satisfy $\|R_{k+1}\|_F^2 = \frac{1}{2}\|R_k\|_F^2$ for $k = 1, 2, \ldots, n-1$.*

Proof. For a given n, A^2 is given by

$$A^2 = \text{diag}\{2^{n-2}, 2^{n-3}, \ldots, 2^1, 2^0, 1\}.$$

Then at the kth step, we can show by induction that R_k is equal to A with the first $k-1$ diagonal elements deleted. This is trivially true for R_1, so assume it is true for R_k, and we will show that it is true for R_{k+1}. Using initialization choice 1, we choose $y = e_k$ which corresponds to the largest element in R_k. Then

$$s = R_k y = \begin{bmatrix} 0 & \cdots & 0 & \sqrt{2^{n-k-1}} & 0 & \cdots & 0 \end{bmatrix}^T,$$

so choosing $x = e_k$ is optimal. Thus we will have $x_k = y_k = e_k$ and $d_k = \sqrt{2^{n-k-1}}$, and so $R_{k+1} = R_k - d_k x_k y_k^T$ will be equal to A with the first $k+1$ nonzeros deleted. (Note that this particular triplet is *the* optimal triplet regardless of the inner iteration initialization choice.)

Next observe that

$$\|R_k\|_F^2 = 1 + \sum_{i=0}^{n-k-1} 2^i = 2^{n-k}.$$

Thus

$$\|R_{k+1}\|_F^2 = 2^{n-k-1} = \frac{1}{2}(2^{n-k}) = \frac{1}{2}\|R_k\|_F^2,$$

for $k = 1, \ldots, n-1$. □

Note that we have chosen an example where the O'Leary-Peleg algorithm terminates. Although we can find a finite "semi-discrete" expansion of any matrix A, that is, $\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij} e_i e_j^T$, we are uncertain whether or not the O'Leary-Peleg algorithm is guaranteed to always yield a finite expansion.

Initialization choice 1 may be computationally expensive if we store R_k implicitly as $A - A_k$, so we consider some alternative initialization strategies. First we will propose cycling through the unit vectors although the rate of convergence for this is only guaranteed to be at least n-step linear. Next we will propose a threshold test that is at least linear and requires much less work than the Max Element test.

Initialization Choice 2 (Cycling) *Cycle through the unit vectors in some order, being sure to use each possible unit vector as the starting y every n outer iterations. We refer to each unit of n outer iterations as a* sweep.

Theorem 6.2 *If we use initialization choice 2, then the sequence $\{A_k\}$ generated by the O'Leary-Peleg algorithm converges to A in the Frobenius norm. Furthermore, the rate of convergence is at least n-step linear.*

Proof. The proof is similar to that for Theorem 6.1. Instead of considering each step, consider each sweep. □

This is a very weak result, but we can improve it by using a *threshold test* to determine the initial y.

Initialization Choice 3 (Threshold) *At iteration k, cycle through the unit vectors (beginning where we left off at the last iteration) until we have*

$$\|R_k e_j\|_2^2 \geq \frac{\|R_k\|_F^2}{n},$$

and let $y = e_j$. We are guaranteed that at least one unit vector will satisfy the above inequality by definition of the F-norm.

Even though R_k is stored implicitly, this threshold test is easy to perform because we only need to multiply R_k by a vector. Furthermore, if we accept the first vector we try, we incur no extra computational expense because the computed vector $s = R_k y$ is used in the inner iteration.

Theorem 6.3 *If we use initialization choice 3, then the sequence $\{A_k\}$ generated by the O'Leary-Peleg algorithm converges to A in the Frobenius norm. Furthermore, the rate of convergence is at least linear.*

Proof. The proof is similar to that for Theorem 6.1, and so is omitted. □

6.4 Computational Comparisons

We will compare the three inner iteration initialization strategies for the O'Leary-Peleg algorithm described in the last section.

In our first sequence of examples, we will use the O'Leary-Peleg algorithm to generate a approximation to a 25×25 dense matrix. Figure 6.2 plots the relative residual ($\|R_k\|_F / \|R_0\|_F$) against the number of outer iterations when we fix the number of inner iterations at 1. The solid line represents the result of initialization choice 1 (Max Element). It is does not reduce the residual as quickly as the other two choices because it is, in some sense, too greedy. Initialization choices 2 (Cycle) and 3 (Threshold) are represented by the dotted and dashed lines respectively. The threshold test does better until approximately iteration 60, and then the two are nearly identical.

Figure 6.3 plots the relative residual against the number of outer iterations when we use a varying number of inner iterations — the stopping heuristic is described at the end of Section 6.1. Table 6.1 shows the average number of inner iterations for each initialization choice. In the figure, we see that initialization choice 2 (Cycling) seems to reduce the residual in the fewest number of outer iterations. Figure 6.4 plots the relative residual against the

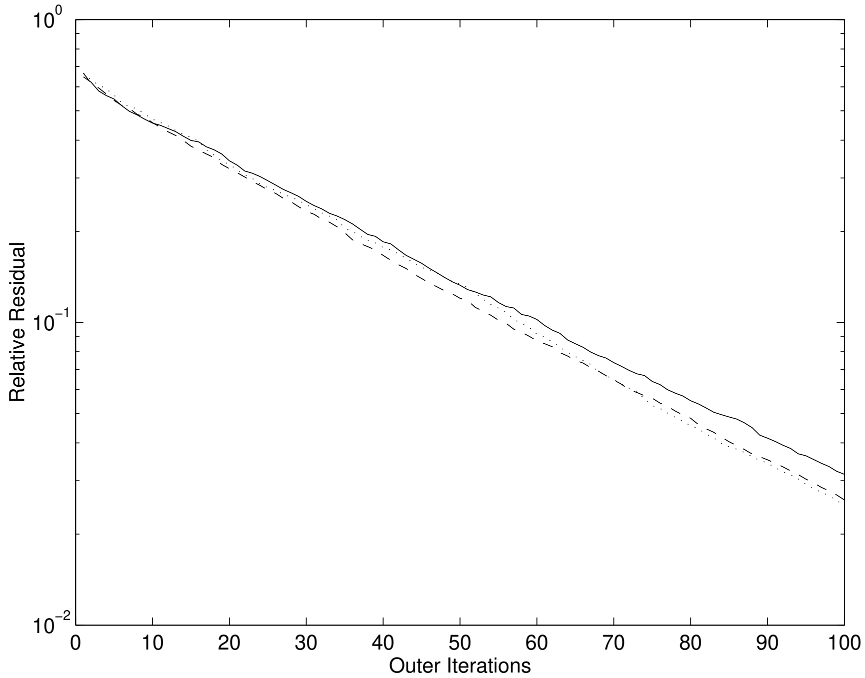

Figure 6.2: Relative residual vs. number of outer iterations of three initialization strategies for the O'Leary-Peleg Algorithm on a 25×25 dense matrix using exactly one inner iteration for each outer iteration. Initialization strategies 1, 2 and 3 are represented by the solid, dotted, and dashed lines respectively.

Initialization Choice	1	2	3
Average Inner Iterations	4.73	4.93	4.52

Table 6.1: Average number of inner iteration for each of three initialization strategies for the O'Leary-Peleg Algorithm on a 25×25 dense matrix using a variable number of inner iterations each outer iteration.

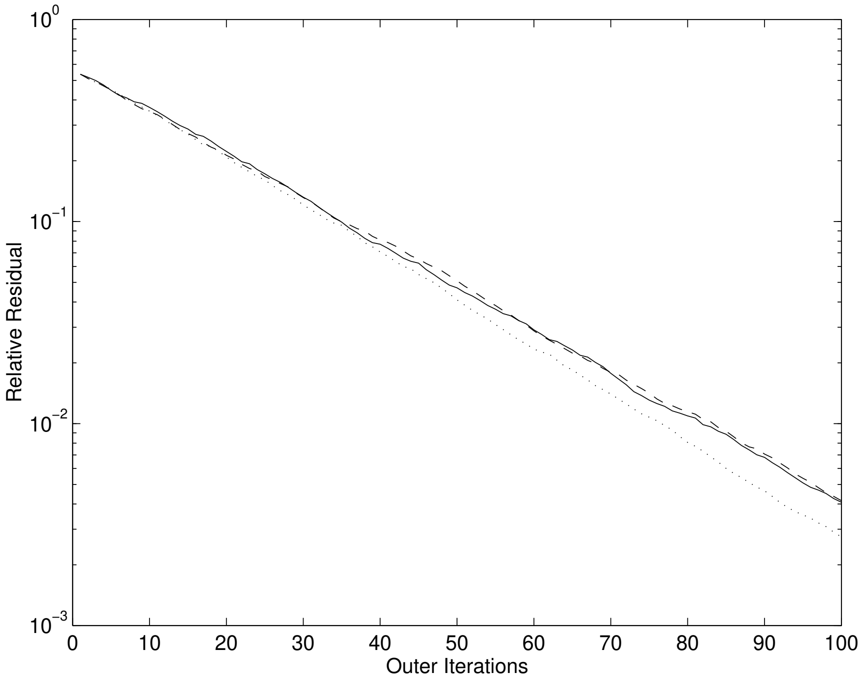

Figure 6.3: Relative residual vs. number of outer iterations for three initialization strategies in the O'Leary-Peleg Algorithm on a 25×25 dense matrix using a variable number of inner iterations each outer iteration. Initialization strategies 1, 2 and 3 are represented by the solid, dotted, and dashed lines respectively.

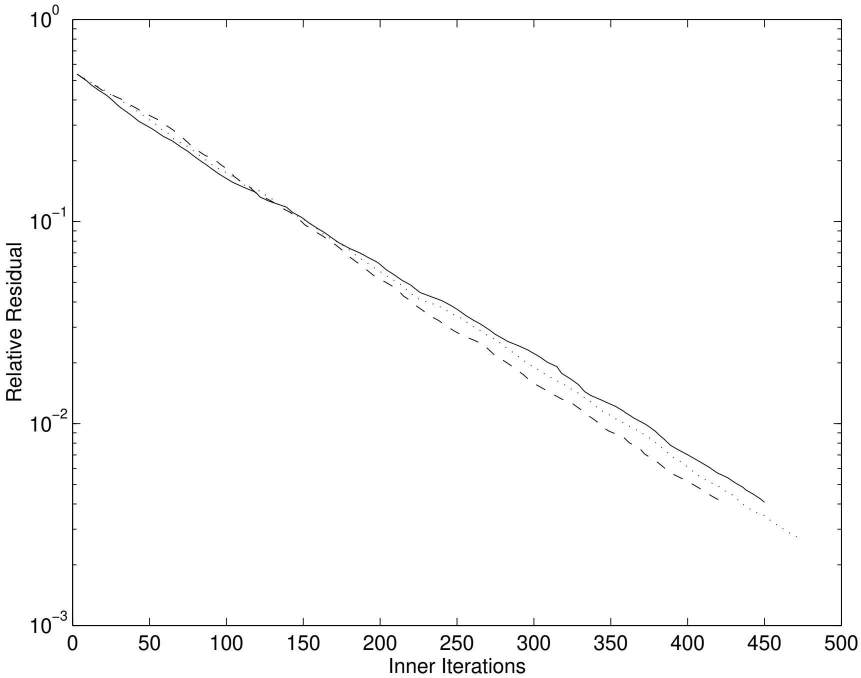

Figure 6.4: Relative residual vs. number of inner iterations for three initialization strategies in the O'Leary-Peleg Algorithm on a 25×25 dense matrix using a variable number of inner iterations each outer iteration. Initialization strategies 1, 2 and 3 are represented by the solid, dotted, and dashed lines respectively.

total number of inner iterations. In terms of inner iterations, the methods seem almost identical initially, and the threshold method seems to be slightly better after approximately 100 inner iterations.

Figure 6.5 plots relative residual against the number of inner iterations for a sparse 100×100 matrix. We performed 200 outer iterations. Initially, the greediest strategy, choice 1 (Max Element), does best, but in the end the threshold test seems to be the best.

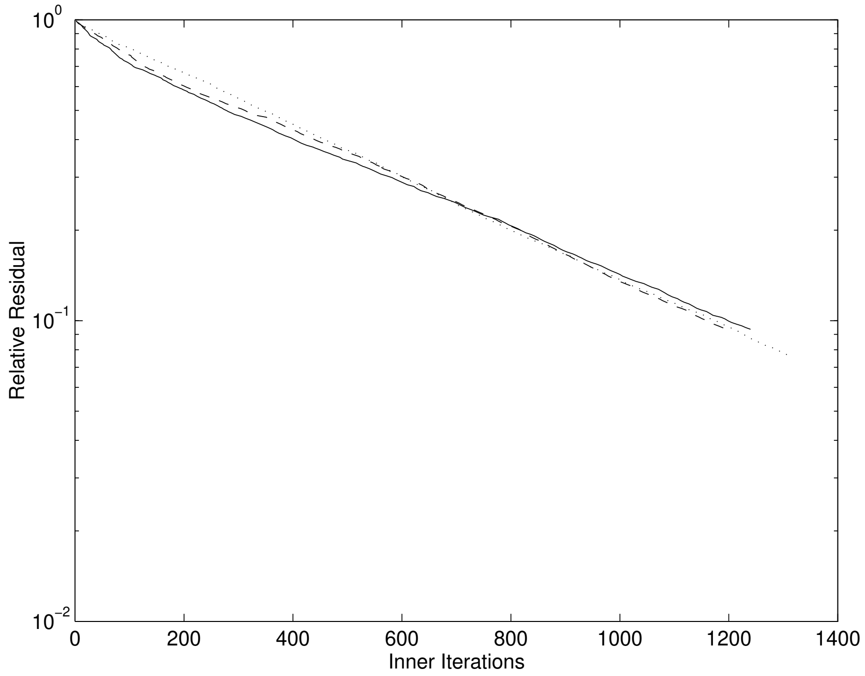

Figure 6.5: Relative residual vs. number of inner iterations for three initialization strategies in the O'Leary-Peleg Algorithm on a 100×100 sparse matrix (approx. 1000 nonzeros) using a variable number of inner iterations each outer iteration. Initialization strategies 1, 2 and 3 are represented by the solid, dotted, and dashed lines respectively.

6.5 Using the SVD to Generate Starting Vectors

The alternating method described in Section 6.1 is one way to heuristically solve (6.3), but here we introduce another possible approach. The singular value decomposition (SVD) approximates an $m \times n$ matrix A by

$$A_k = \underbrace{\begin{bmatrix} u_1 & u_2 & \cdots & u_k \end{bmatrix}}_{U_k} \underbrace{\begin{bmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_k \end{bmatrix}}_{\Sigma_k} \underbrace{\begin{bmatrix} v_1^T \\ v_2^T \\ \vdots \\ v_k^T \end{bmatrix}}_{V_k^T} = \sum_{i=1}^{k} \sigma_i u_i v_i^T.$$

Here each $u_i \in \Re^m$, each $v_i \in \Re^n$, each $\sigma_i \in \Re$, $U_k^T U_k = V_k^T V_k = I$ and $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k \geq 0$. For convenience of discussion, assume $n < m$. After n iterations, the SVD reconstructs A exactly, that is, $A_n = A$. (See Golub and Van Loan [30] for more information.)

If we consider the relaxed version of (6.3),

$$\max_{\substack{u \in \Re^m \\ v \in \Re^n}} \tilde{F}(u, v) \equiv \frac{(u^T R v)^2}{\|u\|_2^2 \|v\|_2^2}, \tag{6.8}$$

the first left, u_1^R, and right, v_1^R, singular vectors of R (not A) maximize $\tilde{F}$ whose maximum is the square of the first singular value of R, $(\sigma_1^R)^2$ by the Courant-Fischer Minimax Theorem [30]. This is an upper bound for the integer program (6.3). Furthermore, we can get a lower bound for (6.3) since $\tilde{F}$ is bounded below in the relaxed case by the square of the least singular value of R, $(\sigma_n^R)^2$.

We can use the solution of the relaxed problem (6.8), which we will denote by u and v, to generate an approximate solution to the discrete problem by finding an $x \in \mathcal{S}^m$ that is a discrete approximation to u, that is, an x that solves

$$\min_{\substack{x \in \mathcal{S}^m \\ \hat{x} \equiv x / \|x\|_2}} \|\hat{x} - u\|_2. \tag{6.9}$$

This problem is easy to solve. Suppose that x has exactly J nonzeros, and order the elements of u so that

$$|u_{i_1}| \geq |u_{i_2}| \geq \cdots \geq |u_{i_m}|.$$

Then choose

$$x_{i_j} = \begin{cases} \operatorname{sign}(u_{i_j}) & \text{if } 1 \leq j \leq J \\ 0 & \text{if } J+1 \leq j \leq m \end{cases}.$$

Therefore, there are only m possible x-vectors we need to check to determine the optimal solution for (6.9). We can find y in a similar fashion.

This leads to two possible initialization strategies:

Initialization Choice 4 (SVD Approximation) *At iteration k, use the discrete x and y vectors that are respectively closest to the first left and right singular vectors of R_k. Do not do any inner iterations.*

Initialization Choice 5 (SVD Start) *At iteration k, chose the discrete y closest to the first right singular vector of R_k.*

We can show that initialization choice 5 yields a linearly convergent method.

Theorem 6.4 *If we use initialization choice 5, then the sequence* $\{A_k\}$ *generated by the O'Leary-Peleg algorithm converges to* A *in the Frobenius norm. Furthermore, the rate of convergence is at least linear.*

Proof. Without loss of generality, assume that $R_k \neq 0$ for all k. (Otherwise the algorithm has terminated at the exact solution.) Let ρ_k denote $\|R_k\|_F^2$. Then $\rho_k \neq 0$ for all k, and by Proposition 6.2 we know that the sequence $\{\rho_k\}$ is strictly decreasing; furthermore, the sequence is bounded below by zero. Thus, the sequence must converge to a limit point, say, ρ^*. If $\rho^* = 0$, then the proof is complete, so assume $\rho^* \neq 0$. Then

$$\rho_k \geq \rho^* > 0 \text{ for all } k. \tag{6.10}$$

Set

$$\epsilon = \frac{\rho^*}{\min\{m,n\}m^2n^2}.$$

By definition of convergence, there exists $\hat{k}(\epsilon)$ such that $\rho_k < \rho^* + \epsilon$ for all $k > \hat{k}(\epsilon)$. Fix $k > \hat{k}(\epsilon)$. Let σ, u and v be the first singular value, left singular vector and right singular vector or R_k respectively, and assume that $R_k = [r_{ij}]$. Choose an initial y that solves

$$\min_{\substack{y \in S^n \\ \hat{y} = y/\|y\|_2}} \|\hat{y} - v\|_2.$$

Without loss of generality, assume that the elements of v are ordered so that

$$|v_1| \geq |v_2| \geq \ldots \geq |v_n|,$$

and assume that y has J nonzeros. Now observe that

$$\sigma = u^T R v = \sum_{j=1}^{J} v_j \sum_{i=1}^{m} r_{ij} u_i + \sum_{j=J+1}^{n} v_j \sum_{i=1}^{m} r_{ij} u_i,$$

and the largest magnitude elements of v must correspond to the largest magnitude elements of Ru, so we can conclude that

$$\sum_{j=1}^{J} v_j \sum_{i=1}^{m} r_{ij} u_i \geq \frac{J}{n}\sigma.$$

Each v_i is less than or equal to one, so substituting y in place of v yields

$$\sum_{j=1}^{n} y_j \sum_{i=1}^{m} r_{ij} u_i \geq \frac{J\sigma}{n}.$$

(Note that this guarantees us that $Ry \neq 0$.) Rewriting this, we get

$$\sum_{i=1}^{m} u_i \sum_{j=1}^{n} r_{ij} y_j \geq \frac{J\sigma}{n},$$

so there exists $\hat{\imath}$ such that

$$u_{\hat{\imath}} \sum_{j=1}^{n} r_{ij} y_j \geq \frac{J\sigma}{mn}.$$

So, if we set $x = e_{\hat{\imath}}$, we have

$$\begin{aligned} \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} &\geq \frac{J^2\sigma^2}{Jm^2n^2} \geq \frac{\sigma^2}{m^2n^2} \geq \frac{\|R_k\|_F^2}{\min\{m,n\} \cdot m^2n^2} \\ &= \frac{\rho_k}{\min\{m,n\} \cdot m^2n^2} \geq \frac{\rho^*}{\min\{m,n\} \cdot m^2n^2} = \epsilon \end{aligned} \tag{6.11}$$

Thus

$$\rho_{k+1} = \rho_k - \frac{(x_k^T R_k y_k)^2}{\|x_k\|_2^2 \|y_k\|_2^2} < (\rho^* + \epsilon) - \epsilon = \rho^*.$$

But this contradicts (6.10), so we conclude that $\rho^* = 0$. Hence, $A_k \to A$.

Now consider the rate of convergence. Using (6.5) and (6.11) we get

$$\begin{aligned} \rho_{k+1} &\leq \rho_k - \frac{\rho_k}{\min\{m,n\} \cdot m^2n^2} = \left(1 - \frac{1}{\min\{m,n\} \cdot m^2n^2}\right) \rho_k \\ &\leq \left(1 - \frac{1}{\min\{m,n\} \cdot m^2n^2}\right)^k \rho_0. \end{aligned}$$

Hence the rate of convergence is at least linear. □

Figure 6.6 plots the relative residual against the number of outer iterations for initialization choice 4 and 1. For the Max Element initialization, we allow 3 inner iterations. This is because we estimate it would take approximately the same amount of work to generate the left and right singular vectors for R_k.

Figure 6.7 plots the relative residual against the number of outer iterations for initialization choices 1 and 5 using a variable number of inner iterations. Here we see that using the SVD start does better in terms of outer iterations. Table 6.2 shows that SVD start only requires about 2.6 inner iterations per outer iteration, but remember that some work was required to generate the left and right singular vectors of R_k, so we argue that the work is about equal.

Initialization Choice	1	5
Average Inner Iterations	4.73	2.61

Table 6.2: Average number of inner iteration for two initialization strategies for the O'Leary-Peleg Algorithm on a 25×25 dense matrix using a variable number of inner iterations each outer iteration.

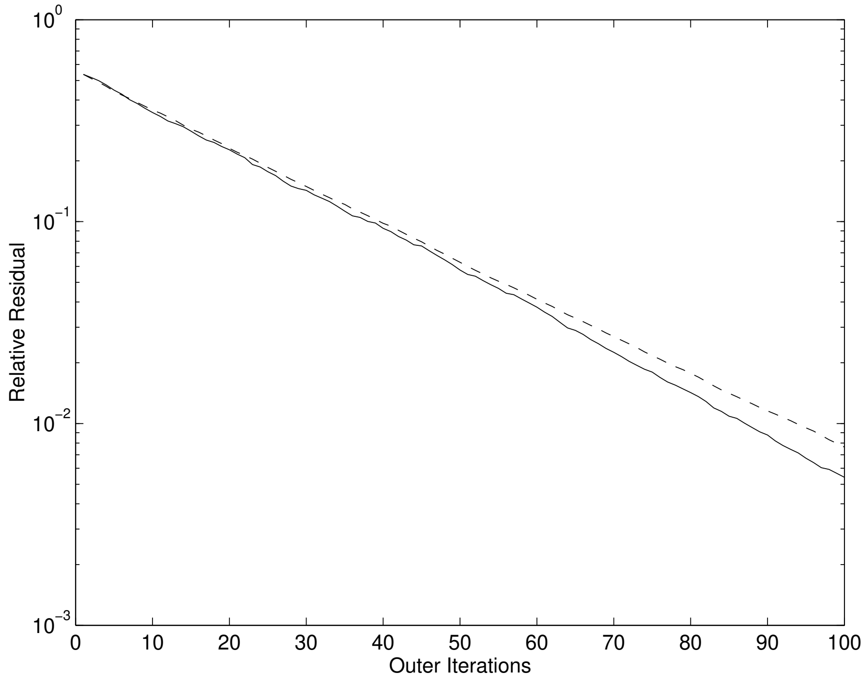

Figure 6.6: Relative residual vs. number of outer iterations comparing initialization choice 4 (dashed line) and initialization choice 1 with three inner iterations (solid line) in the O'Leary-Peleg Algorithm on a 25×25 dense matrix.

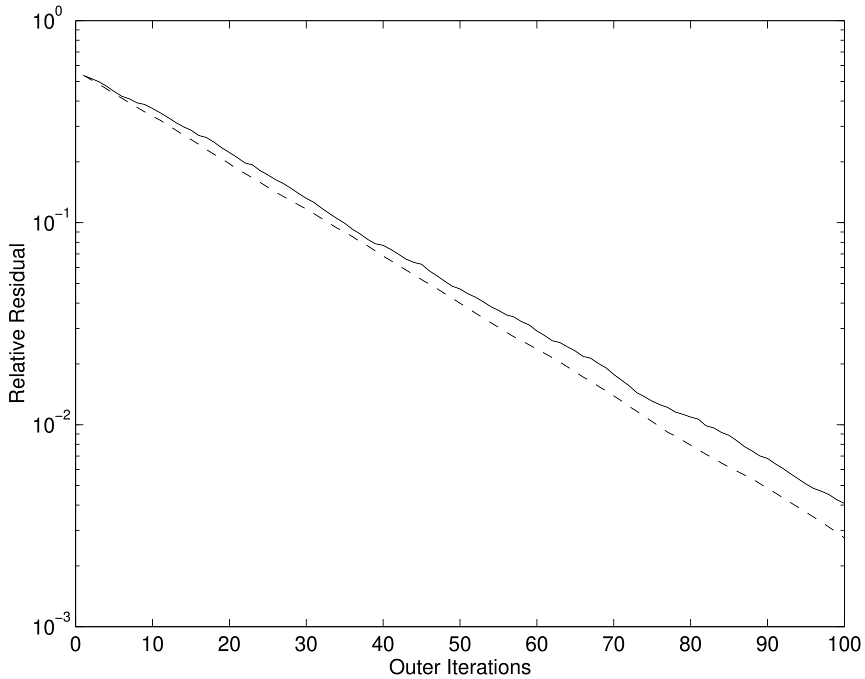

Figure 6.7: Relative residual vs. number of outer iterations comparing the initialization choice 5 (dashed line) and initialization choice 1 (solid line) in the O'Leary-Peleg Algorithm on a 25×25 dense matrix. Each is allowed a variable number of inner iterations.

Chapter 7

The Vector Space Model in Information Retrieval

Here we describe the problem in information retrieval and give a detailed description of the vector space model. In the next chapter, we will discuss an extension of the vector space model called latent semantic indexing and show how this model can be improved using the semi-discrete decomposition described in the previous chapter.

7.1 Introduction

There is an unprecedented amount of electronic textual information available today. This information is useless unless it can be efficiently and effectively searched. In the field of *information retrieval*, we match *queries*, that is, statements of information needs, with relevant *documents*, that is, information items.

Most people have used an information retrieval system at one point or another. Examples include electronic library search catalogs, World Wide Web search engines such as Alta Vista, and Unix tools such as the `grep` utility. To see both the effectiveness and short-comings of an information retrieval system, we have an example of several queries in the University of Maryland VICTOR electronic library catalog in Figure 7.1. For each query, it took only a matter of seconds to search through the more than 2.4 million volumes [65] in the University

Query	**Items Found**
German, math, dictionary	0
German, mathematical, dictionary	2
German, mathematical, dictionaries	3
German, mathematics, dictionary	6
German, mathematics, dictionaries	10

Figure 7.1: Search results from five semantically equivalent queries in the University of Maryland VICTOR electronic library catalog.

of Maryland College Park library — an indication of how efficient modern search engines are. On the other hand, we see that each of the semantically equivalent queries returned a different number of items, and one did not return anything at all — an indication of how far we have yet to go.

There are many different approaches to information retrieval. People are most familiar with the Boolean system: search terms are connected with Boolean operators such as `AND`, `OR`, and `NOT`. Boolean systems can be quite efficient, but are not always very effective. The Boolean system is but one *conceptual model* for an information retrieval system. Another possible model is based on probability theory; the most well-known example of this is the INQUERY system [13]. In this model, each document is assigned a probability of being relevant. The probability is determined via an inference net. A brief survey of these and other modern information retrieval techniques is given in Salton [56]; a more in-depth treatment is given in Frakes and Baeza-Yates [27].

We are concerned here with a conceptual model known as the *vector space model*. This model was proposed over 25 years ago. One implementation of this idea is SMART, see e.g. [59]. In the vector space model, both documents and queries are represented as vectors and compared via inner products. This model will be explained in detail in Section 7.2. One of the most important issues in the vector space model is *term weighting*, the determination of the entries in the vectors representing the documents and queries. We present a computational comparison of the weighting strategies in Section 7.3 and describe evaluation criteria. Many advances and variations have been proposed for the vector space model; these are presented in Section 7.4.

7.2 The Vector Space Model

7.2.1 Preprocessing of the Documents

In the vector space model, we represent documents and queries as vectors. Before we can construct the representation, we must compile a list of *index terms* by preprocessing the documents. We do this as follows:

1. Transform the documents to a list of words by removing all punctuation and numbers. Convert every letter to lower case.

2. Remove *stop words*. Stop words are common words that contain no semantic content such as "of", "next", and "already." This is language and context dependent. Standard lists of stop words exist for the English language. We performed stop word removal using the program and stop word list in Frakes and Baeza-Yates [27].

3. Remove words that appear in only one document. This greatly reduces the number of terms we need to handle. The remaining words form the set of index terms.

This is but one way to preprocess the documents. More elaborate systems might be used in other settings. For example, we might want to treat upper- and lowercase differently.

Punctuation can also be meaningful; in particular, we may want to do something special with hyphenations. There is also no reason to limit ourselves to just single words; we may want to look at phrases as well (see Section 7.4).

One common preprocessing technique that we did not use is *stemming* which reduces words to their common stems. For example, "climb," "climbs," and "climbing" would all be reduced to the stem "climb." This is a heuristic method with many shortcomings, but the example in Figure 7.1 illustrates why stemming might be useful. See Frakes and Baeza-Yates [27] for more information and a sample stemming program.

Lastly, we have specified that, after removing stop words, we will use only terms that appear in at least two documents. In some situations we may find it useful to use all the remaining terms, while in other situations we may want to trim the list even further by removing any term that does not appear in at least, say, five documents.

7.2.2 Description of the Model

We will now formulate the vector space model. Suppose that we have n documents and m index terms, hereafter referred to as just *terms*. We represent the documents as an $m \times n$ matrix, $A = [a_{ij}]$, where a_{ij} represents the *weight* of term i in document j. We describe how to compute the weight of a term in a document in the next section. The jth column of A represents document j. Note that the matrix will be sparse since only a few of the possible terms will appear in each document.

Queries are preprocessed in much the same way as documents. A query is represented as an m-vector, $q = [q_i]$, where q_i represents the *weight* of term i in the query.

The n-vector

$$s = q^T A,$$

is the vector of *scores*. The jth element of s, $s_j = q^T A e_j$ where e_j is the jth unit vector, represents the score of document j. For a given query, the documents are ranked according to score, highest to lowest.

7.2.3 Term Weighting

The success or failure of the vector space method is based on the term weighting. There has been much research on term weighting techniques but little consensus on which method is best. We will summarize a number of the techniques that have been suggested in the literature. Computational comparisons of these techniques will be given in Section 7.3.

There are three components to a weighting scheme. The ij-th entry of A is computed by

$$a_{ij} = g_i \, t_{ij} \, d_j,$$

where g_i is the *global weight* of the ith term, t_{ij} is the *local weight* of the ith term in the jth document, and d_j is the *normalization factor* for the jth document.

Tables 7.1 – 7.3 list possible formulas for each component. Here we assume that f_{ij} is the frequency of term i in document j, all logs are base two, and χ denotes the signum function,

$$\chi(t) = \begin{cases} 1 & \text{if } t > 0, \\ 0 & \text{if } t = 0. \end{cases}$$

Table 7.1 lists some popular local weight formulas; each row of the table gives a unique character symbol, the formula for t_{ij}, a brief description that is the name that the formula is generally known by, and one or more references. Local term formulas depend only on the frequencies *within the document*; they do not depend on inter-document frequencies.

The binary formula (b) and term frequency formula (t) are simple and obvious possibilities for the term frequency component. A major drawback of the binary formula is that it gives every word that appears in a document equal relevance; this might, however, be useful when the number of times a word appears is not considered important. The frequency formula gives more credit to words that appear more frequently, but often *too much* credit. For instance, a word that appears ten times in a document is not usually ten times more important than a word that only appears once. What we would like to do is to give credit to any word that appears and then give some additional credit to words that appear frequently. The augmented normalized term frequency (c) is one such attempt. This gives a value of 0.5 for appearing in the document plus a bonus (no more than 0.5) that depends on the frequency. A more general formula was proposed by Croft [15]; this formula is parameterized by a value K,

$$t_{ij} = K\ \chi(f_{ij}) + (1-K)\left(\frac{f_{ij}}{\max_k f_{kj}}\right).$$

It is suggested that K be set to something low (e.g. 0.3) for large documents and to higher values (e.g. 0.5) for shorter documents. Logarithms are another way to deemphasize the effect of frequency. Two different logarithm term frequency components are popular in the literature; here we call them the log (l) and alternate log (a).

A comparison of the different frequency components for frequencies ranging between 0 and 100 is shown in Figure 7.2; we assume that the maximum frequency (used in the augmented normalized term frequency formula) is 100. Observe that the raw frequency count grows very quickly whereas the other local weightings grow more slowly. Although it is not clear in the picture, every local weight formula assigns a value of zero to t_{ij} if term i does not appear in document j.

Global term weights (see Table 7.2) are used to place emphasis on terms that are discriminating, and they are based on the dispersion of a particular term throughout the documents. For example, the inverse document frequency weighting (f) will be zero if the given term appears in every document, and the weight increases as the number of documents in which the term appears decreases. Various other inverse document frequency (IDF) measures have been proposed; see Frakes and Baeza-Yates for some examples. The probabilistic inverse weight (p), is also referred to as an IDF weight. It assigns weights ranging from $-\infty$ for

Symbol	Formula for t_{ij}	Brief Description	Ref.
b	$\chi(f_{ij})$	Binary	[58]
t	f_{ij}	Term Frequency	[58]
c	$.5\ \chi(f_{ij}) + .5 \left(\frac{f_{ij}}{\max_k f_{kj}} \right)$	Augmented Normalized Term Frequency	[27, 58]
l	$\log(f_{ij} + 1)$	Log	[27]
a	$\chi(f_{ij})(\log(f_{ij}) + 1)$	Alternate Log	[21]

Table 7.1: Local Term Weight Formulas

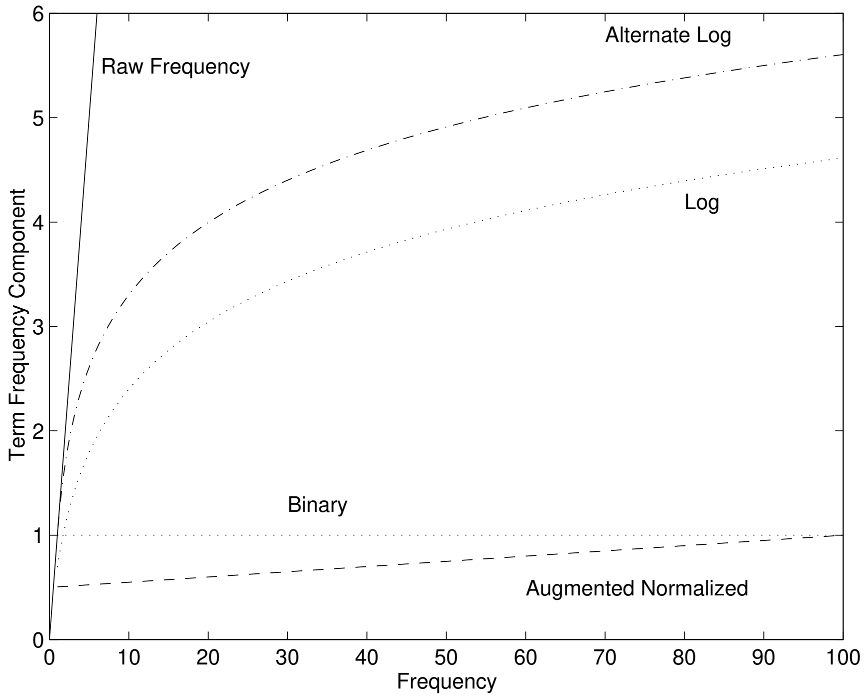

Figure 7.2: Comparison of Local Term Weighting Schemes

Symbol	Formula for g_i	Brief Description	Ref.
x	1	No change	[58]
f	$\log\left(\frac{n}{\sum_{k=1}^{n}\chi(f_{ik})}\right)$	Inverse Document Frequency (IDF)	[58]
p	$\log\left(\frac{n-\sum_{k=1}^{n}\chi(f_{ik})}{\sum_{k=1}^{n}\chi(f_{ik})}\right)$	Probabilistic Inverse	[27, 58]
g	$\frac{\sum_{k=1}^{n} f_{ik}}{\sum_{k=1}^{n}\chi(f_{ik})}$	GfIdf	[21]
e	$1+\sum_{j=1}^{n}\left(\frac{p_{ij}\log p_{ij}}{\log n}\right)$	Entropy ($p_{ij}=f_{ij}/\sum_{k=1}^{n} f_{ik}$)	[21, 27]

Table 7.2: Global Term Weight Formulas

Symbol	Formula for d_i	Brief Description	Ref.
x	1	No Change	[58]
n	$\left(\sum_{k=1}^{m}(g_k t_{kj})^2\right)^{-1/2}$	Normal	[58]

Table 7.3: Normalization Formulas

a term that appears in every document[1] to $\log(n-1)$ for a term that appears in only one document. The GfIdf (g) computes the ratio of the total number of times the term appears in the collection to the number of documents it appears in. The entropy weight (e) assigns weights between zero and one, zero for a term that appears at the same frequency in every document and one for a term that appears in only one document. Global weighting is very successful as we shall see when we look at the experimental results in Section 7.3.

The use of global weighting can, in theory, eliminate the need for stop word removal since stop words should have very small global weights. In practice, however, it is easier to remove the stop words in the preprocessing phase so that there are fewer terms to handle.

Once we have computed the local and global term weights, it is often useful to normalize the columns in the final matrix. If we do not, short documents may not be recognized as relevant. We present two choices in Table 7.3: no normalization (x) and normalization by the 2-norm (n). Other normalization strategies are possible, but the 2-norm is the most popular.

In the vector space method literature, the *cosine score* is often mentioned. This is computed by

$$s_j = \frac{q^T A e_j}{\|q\|_2 \|Ae_j\|_2},$$

which gives the angle between the vector representing the query, q, and the vector representing the document Ae_j. If the columns of A and q have been normalized, the inner product is equivalent to the cosine score. Thus whenever we choose the `n` normalization weighting for the documents, we are using a scoring equivalent to the cosine score. Note that normalizing q makes no difference in the final ranking of documents, so we never do it.

Thus far we have explained how to weight the documents, but not how to weight the query. The query weighting is

$$q_i = g_i \hat{t}_i,$$

where g_i is the global term weight computed as usual from the frequency counts in the *documents* and $\hat{t}_i$ is the local term weight that is computed using the local weight formulas with f_{ij} replaced by $\hat{f}_i$, the frequency of term i in the query. As mentioned before, there is no need to normalize the query.

A weighting scheme is specified by specifying a six-letter combination that indicates local, global, and normalization components for the term-document matrix and the local, global, and normalization components for the query. (The normalization component for the query will always be `x`, but we specify it anyway.) For example, if we were to specify a weighting scheme of `axn.afx`, we would use the following formulas:

$$a_{ij} = \begin{cases} (\log(f_{ij}) + 1) \cdot \left(\sum_{k=1}^{m} (\log(f_{kj}) + 1)^2 \right)^{-1/2} & \text{if } f_{ij} \neq 0, \\ 0 & \text{otherwise,} \end{cases}$$

[1]In practice, if a term were to appear in every document, it would generally be removed from the list of index words.

and

$$q_i = \begin{cases} \left(\log(\hat{f}_i) + 1\right) \cdot \log\left(\frac{n}{\sum_{k=1}^{n} \chi(f_{ik})}\right) & \text{if } \hat{f}_i \neq 0, \\ 0 & \text{otherwise.} \end{cases}$$

This particular weighting is described in Buckley *et al.* [10].

We have a total of $5 \cdot 5 \cdot 2 = 50$ possible weighting schemes for the documents and $5 \cdot 5 = 25$ possible weighting schemes for the queries, yielding a possible $50 \cdot 25 = 1250$ possible weightings. Rather than compare all of these, we will restrict our attention to only a few schemes. These are summarized in Table 7.4. Some are schemes that have been suggested in the literature, and the remainder are modifications of those schemes.

In addition to the weightings discussed above, there are two additional weightings that we will use; they were suggested by Singhal *et al.* [61]. We will talk more about where these weightings originated in Section 7.4. The first is called the OKAPI weight, that shall be denoted by `okapi` in the tests. For OKAPI, the weight on the term-document matrix is given by

$$a_{ij} = \frac{f_{ij} \cdot \log\left(\frac{n - \sum_{k=1}^{n} f_{ik} + 0.5}{\sum_{k=1}^{n} f_{ik} + 0.5}\right)}{2 \cdot \left(0.25 + 0.75\left(\frac{d_j}{\text{mean}_k d_k}\right)\right) + f_{ij}},$$

where d_j is the length of document j in bytes, but we set d_j equal to the number of terms in document j. The query is just the raw term frequency weight, i.e.,

$$q_i = \hat{f}_i.$$

The second weight is called the INQUERY weight, denoted by `inquery`, and is defined as

$$a_{ij} = 0.4\ \chi(f_{ij}) + 0.6 \cdot \left(0.4 \cdot H_j + 0.6 \cdot \frac{\log(f_{ij} + 0.5)}{\log(\max_k f_{kj} + 1.0)}\right) \cdot \frac{\log\left(\frac{n}{\sum_{k=1}^{n} \chi(f_{ik})}\right)}{\log n},$$

if $f_{ij} \neq 0$, and $a_{ij} = 0$ otherwise. Here,

$$H_j = \begin{cases} 1.0 & \text{if } \max_k f_{kj} \leq 25, \\ 25/\max_k f_{kj} & \text{otherwise.} \end{cases}$$

Raw term frequency is used for the query:

$$q_i = \hat{f}_i.$$

7.3 Experimental Comparisons

In this section, we will describe the test sets we are working with and the basis for comparison, and present computational results.

SMART-like weighting schemes	
`txx.txx`	Raw Frequency.
`txn.txx`	Original SMART weighting [21, 58].
`axn.afx`	SMART "ltc" weighting [10].
`lxn.lfx`	Modification of "ltc" weight.

Weightings from Dumais [21]	
`tgn.tgx`	Term frequency plus GfIdf.
`tfn.tfx`	IDF plus term frequency.
`ten.tex`	Entropy with term frequency.
`len.lex`	Log-Entropy.

Weightings from Dumais [21] without Global Weights on Term-Document Matrix	
`txn.tgx`	Term frequency plus GfIdf.
`txn.tfx`	IDF plus term frequency.
`txn.tex`	Entropy with term frequency.
`lxn.lex`	Log-Entropy.

Weightings from Salton & Buckley [58]	
`bxx.bxx`	Coordination level binary vectors.
`bxx.bpx`	Binary independence probabilistic.
`bfx.bfx`	Classical IDF without normalization.
`tfn.cfx`	Best fully weighted system.
`txn.cfx`	Weighted with inverse frequency.
`tfx.tfx`	Classical term frequency plus IDF.
`cxx.bpx`	Best weighted probabilistic.

Other Weightings			
`cxn.bpx`	`cxn.lpx`	`cxn.tpx`	`cxn.bfx`
`cxn.lfx`	`cxn.tfx`	`lxn.bpx`	`lxn.lpx`
`lxn.tpx`	`lxn.bfx`	`lxn.lfx`	`lxn.tfx`

Table 7.4: Term Weightings

7.3.1 Test Collections

An information retrieval system is "good" if it ranks the relevant documents high and the irrelevant documents low. In order to evaluate an information retrieval system, therefore, we need a set of documents, one or more queries, and a set of relevance judgments for each query. Relevance judgments are binary; a document is either relevant or it is not. The validity of the relevance judgments is often questioned, but we will not consider these arguments here; see, e.g., Salton [57].

	MEDLINE	**CRANFIELD**	**CISI**
Number of Documents	1033	1399	1460
Number of (Indexing) Terms	5526	4598	5574
Avg. No. of Terms/Document	48	57	46
Avg. No. of Documents/Term	9	17	12
% Nonzero Entries in Matrix	0.87	1.24	0.82
Storage for Matrix (MB):	0.4	0.6	0.5
Number of Queries	30	225	35
Avg. No. of Terms/Query	10	9	7
Avg. No. Relevant/Query	23	8	50

Table 7.5: Characteristics of the Information Retrieval Test Sets

A number of standard test collections are available. We will be focusing on three: MEDLINE, a collection of medical abstracts; CRANFIELD, a collection of aerospace abstracts; and CISI, a collection of library science abstracts. A summary of the characteristics of these test sets is given in Table 7.5. Since these are collections of abstracts, they only contain a small number of terms (about 50) per document. We have sparse term-document matrices, all around 1% dense. The queries contain approximately eight key terms each, and so are reasonably specific. Both MEDLINE and CISI have a small number of queries (about 30) and a large number of relevant documents per query; CRANFIELD, on the other hand, has a large number of queries but only a small number of relevant documents per query.

The test sets described in Table 7.5 are considered small by modern standards. For the past five years, the TREC competition [32, 33, 34] has been providing test sets with hundreds of thousands of documents. Unfortunately, it is difficult to prepare code for TREC data, and there is little in between in terms of size.

7.3.2 Performance Evaluation Criteria

We will compare the systems by looking at some standard measures used in the information retrieval community: non-interpolated average precision, interpolated average precision, and r-precision.

i	r_i	Recall	Precision
1	1	0.027	1.000
5	5	0.135	1.000
10	9	0.243	0.900
20	16	0.432	0.800
40	27	0.730	0.675
60	35	0.946	0.583
71	37	1.000	0.5211

Non-Interp Avg Prec	0.763
Average Precision	0.781
R-Precision	0.703

Table 7.6: Sample recall and precision values, interpolated and non-interpolated average precision, and r-precision for the first MEDLINE query using the vector space model with weighting `cxn.bpx`. There are a total of 37 relevant documents for this query.

Non-interpolated average precision. When we evaluate a query, we receive an ordered list of documents. Let r_i denote the number of relevant documents up to and including position i in the ordered list. For each document, we compute two values: recall and precision. The recall at the ith document is the proportion of relevant documents returned so far, that is,

$$\frac{r_i}{r_n}.$$

(Note that r_n is the total number of relevant documents.) The precision at the ith document, p_i, is the proportion of documents returned so far that are relevant, that is,

$$p_i = \frac{r_i}{i}.$$

Let I be the set of positions of the relevant documents in the ordered list, then the *non-interpolated average precision* for a single query is defined as

$$\frac{1}{r_n} \sum_{i \in I} p_i.$$

The non-interpolated average precision for multiple queries is defined as the mean of the non-interpolated average precisions for all queries.

The first MEDLINE query has a total of 37 relevant documents. Table 7.6 presents some sample recall and precision values, and Figure 7.3 plots the 37 recall-precision pairs as circles (`o`). We used the `cxn.bpx` weighting in this example. The non-interpolated average precision for the first query is 0.763, and the (mean) non-interpolated average precision over all queries is 0.521.

	Mean	Std Deviation
Non-Interp Avg Prec	0.521	0.200
Average Precision	0.536	0.191
R-Precision	0.495	0.175

Table 7.7: Various measures for the thirty MEDLINE queries using the vector space method with weighting `cxn.bpx`.

Interpolated average precision. The *pseudo-precision* at recall level $x \in [0,1]$, $\tilde{p}(x)$, is defined as

$$\tilde{p}(x) = \max\{p_i \mid r_i \geq x \cdot r_n,\ i = 1, \ldots, n\}.$$

The N-point *interpolated average precision* for a single query is defined as

$$\frac{1}{N}\sum_{i=0}^{N-1} \tilde{p}\left(\frac{i}{N-1}\right).$$

Typically, 11-point interpolated average precision is used. The N-point interpolated average precision for multiple queries is the mean of the interpolated average precisions for all queries.

The *precision-recall graph* plots the N pseudo-precision points generated above against the N evenly spaced recall values between 0 and 1. This is always a non-increasing curve.

Figure 7.3 shows the interpolated precision-recall graph for the first query in the MEDLINE test set. The circles (`o`) represent precision-recall data points from the 37 relevant documents, that is, $\{(r_i, p_i)\}_{i \in I}$, and the asterisks (`*`) represent the interpolated data points, $\tilde{p}$, at 11 evenly-spaced recall levels. The asterisks are connected to form the precision-recall curve. The interpolated average precision for the first query in MEDLINE is 0.781.

For multiple queries, we plot the mean of the pseudo-precision points for each of the N recall values and upper and lower "error lines" that show the standard deviations from each point along the precision recall curve for multiple queries.

Figure 7.4 shows the precision-recall curve for multiple queries including the "error lines." The asterisks (`*`) represent the mean value of all pseudo-precisions at each recall level, and the circles (`o`) represent the points one standard deviation away on either side. The value of the interpolated average precision over all MEDLINE queries is 0.536.

R-precision. For a single query, *r-precision* is the precision after r documents have been returned, that is,

$$p_{r_n}.$$

The r-precision for multiple queries is the mean of the r-precisions of all queries.

The r-precision for the first query of MEDLINE is 0.703 and the average r-precision for all queries of MEDLINE is 0.495.

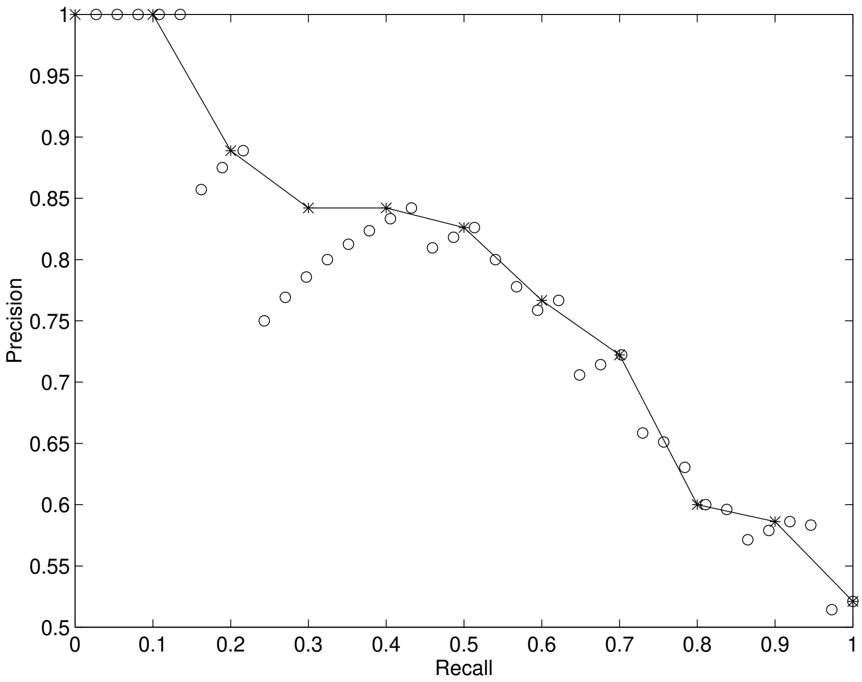

Figure 7.3: The 11-point interpolated precision-recall curve for the first query in the the MEDLINE data set with weighting `cxn.bpx`. The circles (`o`) represent the 37 precision-recall pairs from the relevant documents, and the asterisks (`*`) represent the interpolated points. The asterisks are connected forming the precision-recall curve.

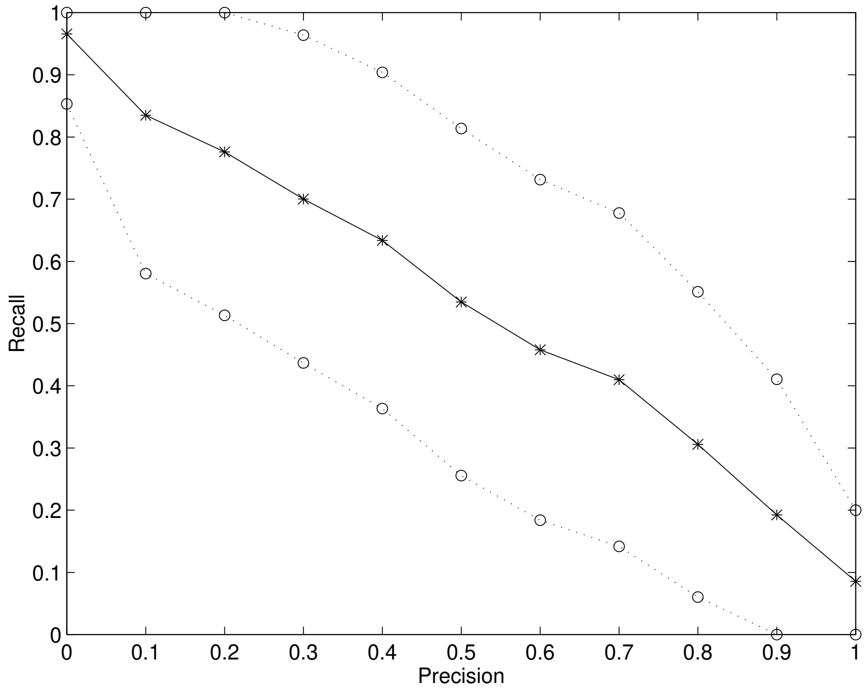

Figure 7.4: The 11-point interpolated precision-recall curve for the 30 queries in the MEDLINE data set using the weighting `cxn.bpx`. At each of the 11 recall levels, the asterisk (`*`) represents the mean of all pseudo-precisions at that level and the circles (`o`) represent the distance one standard deviation from the mean.

Weighting	Average Precision		Non-Interp Avg Prec		R-Precision	
	mean	s.d.	mean	s.d.	mean	s.d.
lxn.bpx	0.546	0.185	0.536	0.195	0.504	0.163
lxn.bfx	0.546	0.185	0.536	0.194	0.501	0.164
lxn.lpx	0.538	0.190	0.529	0.202	0.499	0.172
lxn.lfx	0.537	0.191	0.528	0.202	0.499	0.172
lxn.lfx	0.537	0.191	0.528	0.202	0.499	0.172
cxn.bpx	0.536	0.190	0.521	0.200	0.494	0.175
cxx.bpx	0.536	0.190	0.521	0.200	0.494	0.175
cxn.bfx	0.536	0.189	0.521	0.199	0.494	0.175
tgn.tgx	0.535	0.193	0.513	0.204	0.493	0.198
okapi	0.535	0.199	0.521	0.213	0.489	0.187
lxn.tpx	0.534	0.192	0.521	0.202	0.498	0.178
axn.afx	0.533	0.191	0.520	0.200	0.504	0.180
cxn.lpx	0.532	0.196	0.516	0.204	0.496	0.181
cxn.lfx	0.532	0.194	0.516	0.204	0.495	0.181
lxn.tfx	0.532	0.193	0.520	0.203	0.496	0.181
cxn.tpx	0.526	0.197	0.509	0.206	0.484	0.182
cxn.tfx	0.525	0.195	0.507	0.204	0.482	0.180
txn.cfx	0.525	0.182	0.513	0.193	0.497	0.171
tfn.cfx	0.525	0.180	0.508	0.193	0.482	0.164
txn.tgx	0.522	0.187	0.503	0.198	0.481	0.185
txn.tfx	0.517	0.186	0.501	0.194	0.482	0.183
len.lex	0.516	0.191	0.500	0.205	0.482	0.183
lxn.lex	0.516	0.191	0.500	0.205	0.480	0.184
tfn.tfx	0.516	0.186	0.495	0.195	0.478	0.178
tfx.tfx	0.516	0.186	0.495	0.195	0.478	0.178
bxx.bpx	0.498	0.198	0.485	0.209	0.449	0.182
ten.tex	0.483	0.192	0.467	0.201	0.438	0.187
txn.tex	0.483	0.192	0.467	0.201	0.438	0.187
txx.txx	0.482	0.193	0.467	0.201	0.436	0.185
txn.txx	0.482	0.192	0.467	0.201	0.436	0.185
bfx.bfx	0.479	0.193	0.460	0.203	0.443	0.187
bxx.bxx	0.470	0.192	0.455	0.200	0.430	0.173
inquery	0.422	0.215	0.392	0.220	0.381	0.208

Table 7.8: Results on the MEDLINE data set for various weightings. The results are sorted highest to lowest in terms of average precision.

Weighting	Average Precision		Non-Interp Avg Prec		R-Precision	
	mean	s.d.	mean	s.d.	mean	s.d.
okapi	0.461	0.687	0.401	0.687	0.336	0.211
lxn.lpx	0.457	0.686	0.396	0.260	0.339	0.211
lxn.lfx	0.456	0.687	0.396	0.262	0.330	0.212
lxn.lfx	0.456	0.687	0.396	0.262	0.330	0.212
lxn.tfx	0.456	0.686	0.396	0.260	0.330	0.213
lxn.tpx	0.456	0.686	0.395	0.259	0.336	0.213
lxn.bpx	0.455	0.687	0.395	0.261	0.340	0.209
lxn.bfx	0.455	0.686	0.394	0.258	0.330	0.209
axn.afx	0.454	0.686	0.393	0.260	0.331	0.214
cxn.lfx	0.442	0.686	0.382	0.257	0.325	0.207
cxn.bfx	0.441	0.685	0.381	0.256	0.322	0.204
cxn.tfx	0.439	0.686	0.380	0.259	0.325	0.206
cxn.lpx	0.436	0.685	0.376	0.255	0.321	0.207
cxn.bpx	0.434	0.684	0.374	0.251	0.320	0.203
cxx.bpx	0.434	0.684	0.374	0.251	0.320	0.203
txn.cfx	0.433	0.688	0.372	0.261	0.317	0.209
cxn.tpx	0.433	0.686	0.373	0.255	0.321	0.206
txn.tfx	0.430	0.688	0.370	0.260	0.317	0.208
tfn.tfx	0.425	0.688	0.363	0.258	0.307	0.210
tfx.tfx	0.425	0.688	0.363	0.258	0.307	0.210
lxn.lex	0.425	0.687	0.365	0.258	0.316	0.205
len.lex	0.424	0.687	0.364	0.257	0.312	0.205
tfn.cfx	0.424	0.687	0.363	0.257	0.306	0.210
ten.tex	0.397	0.685	0.337	0.248	0.292	0.200
txn.txx	0.396	0.685	0.336	0.247	0.290	0.199
txn.tex	0.395	0.685	0.335	0.248	0.287	0.199
txx.txx	0.395	0.685	0.335	0.247	0.287	0.200
bxx.bpx	0.392	0.684	0.333	0.240	0.286	0.192
txn.tgx	0.376	0.687	0.316	0.250	0.264	0.205
bxx.bxx	0.366	0.682	0.306	0.230	0.268	0.190
bfx.bfx	0.361	0.685	0.301	0.235	0.262	0.194
tgn.tgx	0.341	0.688	0.282	0.245	0.238	0.198
inquery	0.296	0.678	0.232	0.198	0.205	0.181

Table 7.9: Results on the CRANFIELD data set for various weightings. The results are sorted highest to lowest in terms of average precision.

Weighting	Average Precision		Non-Interp Avg Prec		R-Precision	
	mean	s.d.	mean	s.d.	mean	s.d.
okapi	0.186	0.114	0.161	0.107	0.178	0.126
lxn.tfx	0.184	0.119	0.160	0.112	0.182	0.131
lxn.tpx	0.183	0.116	0.160	0.110	0.183	0.125
lxn.lpx	0.183	0.112	0.159	0.106	0.183	0.123
lxn.lfx	0.182	0.115	0.158	0.107	0.180	0.127
lxn.lfx	0.182	0.115	0.158	0.107	0.180	0.127
cxn.tfx	0.182	0.112	0.155	0.104	0.180	0.128
cxn.tpx	0.182	0.110	0.156	0.103	0.178	0.124
cxn.lfx	0.180	0.107	0.153	0.097	0.180	0.124
axn.afx	0.179	0.118	0.157	0.111	0.179	0.125
cxn.lpx	0.179	0.105	0.154	0.097	0.177	0.121
tfn.tfx	0.178	0.112	0.157	0.107	0.181	0.117
tfx.tfx	0.178	0.112	0.157	0.107	0.181	0.117
lxn.bpx	0.178	0.104	0.154	0.097	0.182	0.124
lxn.bfx	0.177	0.106	0.152	0.097	0.177	0.123
txn.cfx	0.176	0.114	0.153	0.107	0.164	0.127
tfn.cfx	0.176	0.105	0.154	0.099	0.182	0.112
txn.tfx	0.175	0.119	0.154	0.115	0.166	0.127
cxn.bpx	0.175	0.101	0.148	0.091	0.173	0.118
cxx.bpx	0.175	0.101	0.148	0.091	0.173	0.118
cxn.bfx	0.174	0.102	0.146	0.090	0.175	0.120
bxx.bpx	0.166	0.091	0.140	0.081	0.165	0.108
bfx.bfx	0.163	0.086	0.134	0.075	0.161	0.099
lxn.lex	0.143	0.114	0.123	0.103	0.141	0.118
len.lex	0.143	0.113	0.123	0.103	0.141	0.118
inquery	0.140	0.082	0.109	0.069	0.125	0.106
bxx.bxx	0.137	0.086	0.113	0.074	0.137	0.106
ten.tex	0.134	0.115	0.118	0.108	0.127	0.119
txn.tex	0.134	0.115	0.118	0.108	0.127	0.119
txn.txx	0.134	0.115	0.118	0.108	0.127	0.119
txx.txx	0.134	0.115	0.118	0.108	0.127	0.119
txn.tgx	0.129	0.111	0.109	0.102	0.116	0.111
tgn.tgx	0.126	0.110	0.104	0.102	0.108	0.110

Table 7.10: Results on the CISI data set for various weightings. The results are sorted highest to lowest in terms of average precision.

MEDLINE		CRANFIELD		CISI	
lxn.bfx	0.546	okapi	0.461	okapi	0.186
lxn.bpx	0.546	lxn.lpx	0.457	lxn.tfx	0.184
lxn.lpx	0.538	lxn.lfx	0.456	lxn.lpx	0.183
lxn.lfx	0.537	lxn.tpx	0.456	lxn.tpx	0.183
cxx.bpx	0.536	lxn.lfx	0.456	lxn.lfx	0.182
cxn.bpx	0.536	lxn.tfx	0.456	cxn.tpx	0.182
cxn.bfx	0.536	lxn.bpx	0.456	cxn.tfx	0.182

Table 7.11: The best weightings for each data set in terms of (interpolated) average precision.

7.3.3 Computational Results

Tables 7.8, 7.9, and 7.10 show numerical results for the MEDLINE, CRANFIELD and CISI data sets using the weightings listed in Table 7.4. For each test, we report the mean and standard deviation for the average precision, the non-interpolated average precision, and the r-precision. In general, one compares methods via the average precision measure. As we can see from the three tables, the average precision, non-interpolated average precision and r-precision are closely correlated.

The `okapi` weighting is the best on the CRANFIELD and CISI data. We observe that in all three cases, the matrix weighting `lxn` gives good results. The weighting `cxn` is also good for the MEDLINE and CISI sets and `axn` is good for the CRANFIELD test set. The binary matrix weightings, `bfx` and `bxx`, and the raw term frequency weightings, `txx` and `txn`, tend to be ranked towards the bottom, indicating that more sophisticated terms weightings, such as the log weights, should be employed.

It seemed to make little difference which query weighting was chosen, probably because the difference between the schemes is minimal when the terms only appear once or twice.

In terms of global weighting, note that the schemes that did not employ global term weighting on the matrix generally fared as well or better than the schemes that did use global weighting on the term document matrix. In particular, compare the weightings with and without a global weight on the term-document matrix. We suggest applying global weights to the query. The best global weightings were the Inverse Document Frequency (`f`) and the Probabilistic Inverse (`p`) weightings.

Table 7.11 lists the best weightings (in terms of average precision) for each data set.

7.4 Recent Advances in the Vector Space Model

A number of extensions and improvements to the vector space model have been proposed. Most important is the latent semantic indexing (LSI) model described in the next chapter. Here we will give a brief survey of some of the other advances, many of which can also be applied in the LSI model.

Started in 1992, the annual Text REtrieval Conference (TREC) spurned many new developments in information retrieval and filtering. TREC participants have access to large realistic document collections with queries and relevance judgments. The TREC collections are a few gigabytes in size; for example, the collection used for the third TREC competition is over three gigabytes and contained over one million documents on many different topics [34]. Before TREC, test collections were only a few megabytes. In the TREC competition, various teams test their products against one another. The competition has sparked a wealth of new ideas.

The SMART group has been among the top competitors in every TREC competition; however, they have had to improve their system to keep up.

One modification is the use of *term phrases* in addition to individual terms. Specifically, let any pair of adjacent non-stop words be a potential phrase. The final set of term phrases are those potential phrases that appear in 25 documents or more. This may also affect normalization [10].

One of the difficulties in information retrieval is forming the query; it generally contains only a few terms and can make retrieval extremely difficult. A number of competitors, including SMART, use something called *massive query expansion*. The idea is to add many terms to the query — the problem is determining those terms. Thesaurus look-up is one possible method, but it can give misleading results for words that have multiple meanings, for example, "foot." The method used in TREC is interesting because it did not work on smaller test sets. In this case, the first few documents, say 25, from the search with the normal query are assumed to be relevant. Terms from these documents are then used in some way to form a new query with many more terms and the search is repeated. This method works in TREC because the first few documents are relevant. In smaller test sets, that is not usually the case [10].

Another difficulty in the vector space model is the representation of documents of varying length. A long document may cover many topics, but each term would have a very small weight if column normalization is done (as is done in the SMART "ltc" weighting). Singhal *et al.* [61] noted that, in the TREC collection, longer documents are statistically more likely to be relevant than shorter documents. This is because longer documents tend to cover more subjects. Two new weighting schemes were proposed based on two of SMART's best competitors: OKAPI and INQUERY. These weightings were presented in Section 7.2.3. In tests, the new weightings did better than the SMART "ltc" weightings [61]. We included these methods in our own tests. The OKAPI weighting did better than the best of all the other weightings tested on the CRANFIELD and CISI data. The INQUERY weighting did not do well on any test set. Both of these weightings were originally designed for very large document collections, so it is not surprising that the INQUERY weighting does poorly. The constants should probably be chosen differently for short documents like the ones in our test sets.

Notice that the Okapi weight, using the modified definition of d_j, breaks into local and global components. The INQUERY weight, on the other hand, is slightly too complicated to break into these components. The fraction on the far right, which is almost a global weight, is only multiplied by part of the expression.

Chapter 8

Latent Semantic Indexing

In the previous chapter, we described the vector space model for information retrieval. In this chapter we will describe an improvement to this model known as latent semantic indexing and show how it can be improved using the semi-discrete decomposition described in Chapter 6. This chapter consists of material taken (sometimes verbatim) from Kolda and O'Leary [38, 39].

8.1 Introduction

Oftentimes in information retrieval, users are searching for documents about a concept that is not accurately described by their list of keywords. For example, a query on a term such as "Mark Twain" is unlikely to retrieve documents that only mention "Samuel Clemens." *We* might know that these are the same people, but the information retrieval systems have no way of knowing. *Latent semantic indexing* (LSI) is an approach to retrieval that attempts to automatically discover latent relationships in the document collection.

LSI is based on the vector space model described in Chapter 7, but the $m \times n$ term-document matrix is replaced by a low-rank singular value decomposition (SVD) approximation (see Section 6.5 for more about the SVD) [17]. This approximation to the term-document matrix is optimal in the sense of minimizing the distance between that matrix and all rank-k matrices. LSI has performed well in both large and small tests; see, for example, Dumais [21, 22].

Thus far, only the singular value decomposition and its relatives, the ULV and URV decompositions [2], have been used in LSI. We propose using a very different decomposition, the semi-discrete decomposition (SDD) described in Chapter 6. This decomposition is constructed via a greedy algorithm and is not an optimal decomposition in the sense that it minimizes with respect to any norm; however, for equal query times, the SDD does as well as the SVD method and requires approximately one-tenth the storage. The trade-off is that the SDD takes substantially longer to compute for sparse matrices, but this is only a one-time expense. Computational comparisons with the SVD are given in Section 8.4.

In many information retrieval settings, the document database is constantly being updated. Much work has been done on updating the SVD approximation to the term-document matrix [4, 49], but it can be as expensive as computing the original SVD. Efficient algorithms for updating the SDD are given in Section 8.5.

8.2 LSI via the SVD

LSI is an improvement on the vector space model. In LSI, we can use a matrix approximation to the term-document matrix generated by the SVD. The SVD is described in Section 6.5. It can be shown that A_k is the best rank-k approximation to A in the Frobenius norm and in the Euclidean norm [30].

The approximation matrix is a "noisy" version of the original matrix. Suppose that "Clemens" and "Twain" often appear together in the document collection. If we then have one document that only mentions "Twain," then ideally the approximation will add some noise to the "Clemens" entry as a result of compressing the rank of the matrix. The amount of noise depends on the size of k. For very small values of k, there is lots of noise — usually too much — and as k grows, the noise gets smaller until it completely disappears. At some intermediate value of k, we have about the right amount of noise.

We can process queries using our approximation for A:

$$\begin{aligned} s &= q^T A \approx q^T A_k \\ &= q^T U_k \Sigma_k V_k^T \\ &= (q^T U_k \Sigma_k^{\alpha})(\Sigma_k^{1-\alpha} V_k^T) \\ &\equiv \tilde{q}^T \tilde{A}. \end{aligned}$$

The scalar α controls the splitting of the Σ_k matrix and has no effect unless we re-normalize the columns of $\tilde{A}$. We will experiment with various choices for α and re-normalization in Section 8.4.1.

The SVD has been used quite effectively for information retrieval, as documented in numerous reports. We recommend the original LSI paper [17], a paper reporting the effectiveness of the LSI approach on the TREC-3 dataset [21], and a more mathematical paper [4] for further information on the SVD for LSI.

8.3 LSI via the SDD

The SVD produces the best rank-k approximation to a matrix, but generally, even a small SVD approximation requires more storage than the original matrix if the original matrix is sparse. To save storage and query time, we propose replacing the SVD by the SDD, described in Chapter 6. This decomposition does not reproduce A exactly, even if $k = n$, but the rank-k approximation can use substantially less storage. The SDD requires only the storage of $2k(n + m)$ values from the set $\{-1, 0, 1\}$ and k scalars. An element of the set $\{-1, 0, 1\}$ can be expressed using $\log_2 3$ bits, although our implementation uses two bits per element for simplicity. Furthermore, the SDD requires only single precision scalars because it is a self-correcting algorithm; on the other hand, the SVD has been computed in double precision accuracy for numerical stability. Assuming that double precision scalars require 8 bytes and single precision scalars require 4, and packing 8 bits in a byte, we obtain the following storage comparison between a rank-k SVD and SDD approximation to an $m \times n$ matrix:

Method	Component		Total Bytes
SVD	U	km double precision numbers	$8k(m+n+1)$
	V	kn double precision numbers	
	Σ	k double precision numbers	
SDD	X	km numbers from $\{-1,0,1\}$	$4k + \frac{1}{4}k(m+n)$
	Y	kn numbers from $\{-1,0,1\}$	
	D	k single precision numbers	

We evaluate queries in much the same way as we did for the SVD, by computing $s = \tilde{q}^T \tilde{A}$, with

$$\tilde{A} = D_k^{1-\alpha} Y_k^T, \quad \tilde{q} = D_k^{\alpha} X_k^T q.$$

Again, we generally re-normalize the columns of $\tilde{A}$.

For decompositions of equal rank, processing the query for the SDD requires significantly fewer floating-point operations than processing the query for the SVD:

Operation	SDD	SVD
Additions	$k(m+n)$	$k(m+n)$
Multiplications	k	$k(1+m+n)$

If we re-normalize the columns of $\tilde{A}$ then each each method requires n additional multiplies and storage of n additional floating point numbers.

8.4 Computational Comparison of LSI Methods

In this section, we present computational results comparing the SDD- and SVD-based LSI methods. All tests were run on a Sparc 20. Our code is in C, with the SVD taken from SVDPACKC [3]. We will compare the SDD- and SVD-based LSI methods using the same test sets as we did in the last chapter.

8.4.1 Parameter Choices

We have two parameter choices to make for the SDD and SVD methods: the choice of the splitting parameter α, and the choice of whether or not to re-normalize the columns of $\tilde{A}$.

	SDD		SVD	
	Re-Normalize?		Re-Normalize?	
α	Yes	No	Yes	No
0	62.1	61.2	**65.1**	64.2
0.5	**62.6**	61.2	64.7	64.2
-0.5	57.9	61.2	64.7	64.2
1.0	61.7	61.2	64.2	64.2
-1.0	48.6	61.2	62.3	64.2

Table 8.1: Mean average precision for the SDD and SVD methods with different parameter choices on the MEDLINE data set with k=100 and weighting `lxn.bpx`.

We experimented with the SVD and SDD methods on the MEDLINE data set using the weighting `lxn.bpx`. The results are summarized in Table 8.1. In all further tests, we will use $\alpha = 0.5$ with re-normalization for the SDD method and $\alpha = 0$ with re-normalization for the SVD method. We experimented using other weighings and other data sets and confirmed that these parameter choices are always best or very close to it.

8.4.2 Comparisons

We tried the SDD and SVD methods with a number of weighings. We selected these particular weighings for testing in LSI based on their good performance for the vector space method (see Chapter 7), but we excluded those that included any global weighting on the term-document matrix. We present mean average precision results in Table 8.2 using a rank $k = 100$ approximation in each method; this table also includes vector space (VS) results for comparison.

	MEDLINE			CRANFIELD			CISI		
Weight	SDD	SVD	VS	SDD	SVD	VS	SDD	SVD	VS
`lxn.bfx`	62.6	64.6	54.6	35.7	**40.4**	45.5	15.6	16.6	17.7
`lxn.bpx`	**62.6**	**65.1**	**54.6**	35.6	39.9	45.5	15.2	16.9	17.8
`lxn.lfx`	61.2	64.0	53.7	**35.8**	40.3	45.6	16.0	16.6	18.2
`lxn.lpx`	61.3	64.3	53.8	35.5	40.1	**45.7**	15.5	16.9	18.3
`lxn.tfx`	60.9	63.5	53.2	35.7	40.2	45.6	16.3	16.9	**18.4**
`lxn.tpx`	60.9	63.8	53.4	35.4	39.9	45.6	15.7	17.0	18.3
`cxx.bpx`	57.9	59.6	53.6	32.9	38.9	43.4	17.1	**17.9**	17.5
`cxn.bfx`	58.4	62.5	53.6	33.1	38.7	44.1	17.8	16.5	17.4
`cxn.bpx`	58.4	63.0	53.6	32.6	38.7	43.4	**18.1**	17.6	17.5
`cxn.tfx`	56.8	61.5	52.5	33.3	38.8	43.9	17.1	16.9	18.2
`cxn.tpx`	57.0	61.8	52.6	32.7	38.2	43.3	17.1	17.7	18.2

Table 8.2: Mean average precision results for the SDD and SVD methods with k=100.

To continue our comparisons, we select a "best" weighting for each data set. In Table 8.2 we have highlighted the "best" results for each data set in boldface type. We will use only the corresponding weighings for the remainder of the paper, although further experiments show similar results for other weighings.

In Figures 8.1 – 8.3, we compare the SVD and SDD methods on the data sets.

In Figure 8.1, we present results for the MEDLINE data. The upper right graph plots the mean average precision vs. query time, and the upper left graph plots the median average precision vs. query time. (The query time is the total time required to execute all queries associated with the data set.) Observe that the SDD method has maximal precision at a query time of 3.4 seconds, corresponding to $k = 140$, a mean average precision of 63.6 and a median average precision of 71.4. The SVD method reaches its peak at 8.4 seconds, corresponding to $k = 110$, and mean and median average precisions of 65.5 and 71.7 respectively.

In terms of storage, the SDD method is extremely economical. The middle left graph plots mean average precision vs. decomposition size in megabytes (MB), and the middle right graph plots median average precision vs. the decomposition size. Note that a significant amount of extra storage space is required in the computation of the SVD; this is not reflected in these numbers. From these plots, we see that even a rank-30 SVD takes 50% more storage than a rank-600 SDD, and each increment of 10 in rank adds approximately 0.5 MB of additional storage to the SVD. The original data takes only 0.4 MB, but SVD requires over 1.5 MB before it even begins to come close to what the SDD can do in less than 0.2 MB.

The lower left graph illustrates the growth in required storage as the rank of the decomposition grows. For a rank-600 approximation, the SVD requires over 30 MB of storage while the SDD requires less than 1 MB.

It is interesting to see how good these methods are at approximating the matrix. The lower right graph shows the Frobenius norm (F-norm) of the residual, divided by the Frobenius norm of the original matrix, as a function of storage (logarithmic scale). The SVD eventually forms a better approximation to the term-document matrix, making it behave more like the vector space method. This is not necessarily desirable.

The CRANFIELD dataset is troublesome for LSI techniques; they do not do as well as the vector space method. From the upper two graphs in Figure 8.2 we see that, for equal query times, the SDD method does as well as the SVD method. The other graphs show that, as in the MEDLINE test, the SDD is much more economical in terms of storage and achieves a somewhat less accurate approximation of the matrix.

In Figure 8.3 we compare the SVD and SDD methods on the CISI data. The SDD method is better overall than the SVD method in terms of query time, and its mean average precision peaks higher than the SVD method — 19.1 versus 18.3. Again, the storage differences are dramatic.

Table 8.3 compares the two methods for the query time at which the SDD method peaks on mean average precision. On all three data sets, the SDD has higher mean and median precisions than the SVD. Since all the methods have similar performance in terms of the mean and median average precision, observe that the trade-off is in the decomposition computation time and the decomposition storage requirement; the SVD is much faster to compute, but

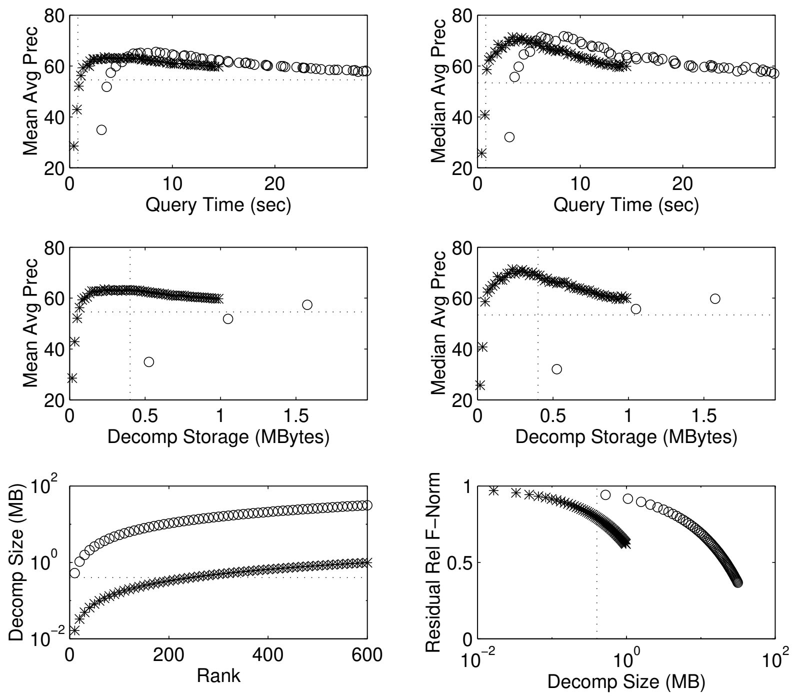

Figure 8.1: A comparison of the SVD (∘) and SDD (∗) on the MEDLINE data set. We plot 60 data points for each graph, corresponding to $k = 10, 20, \ldots, 600$. The dotted lines show the corresponding data for the vector space method.

	MEDLINE		CRANFIELD		CISI	
	SDD	SVD	SDD	SVD	SDD	SVD
Query Time (Sec)	3.4	3.6	63.8	77.3	4.3	4.4
Dimension (k)	140	20	390	210	140	30
Mean Avg Prec	**63.6**	51.8	**44.9**	44.5	**19.1**	15.2
Median Avg Prec	**71.4**	55.7	37.3	**37.4**	**19.4**	12.2
Decomp Storage (MB)	**0.2**	1.1	**0.6**	10.1	**0.2**	1.7
Decomp Time (Sec)	245.4	**4.7**	1313.8	**91.5**	279.0	**13.0**
Rel F-Norm of Resid	**0.85**	0.90	0.63	**0.59**	**0.85**	0.89

Table 8.3: Comparison of the SDD and SVD methods at the query time where the SDD has the highest mean average precision.

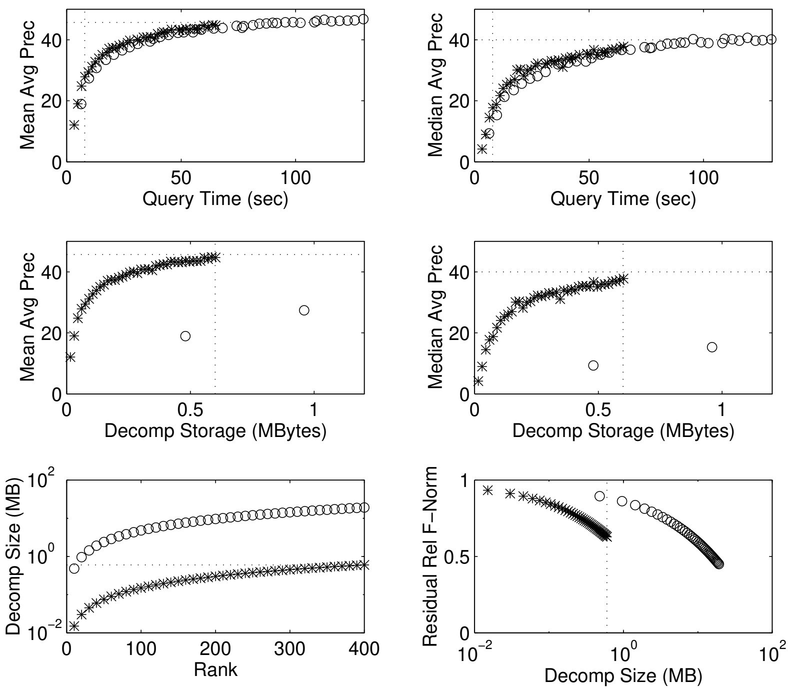

Figure 8.2: A comparison of the SVD (∘) and SDD (∗) on the CRANFIELD data set. We plot 40 data points for each graph, corresponding to $k = 10, 20, \ldots, 400$. The dotted lines show the corresponding data for the vector space method.

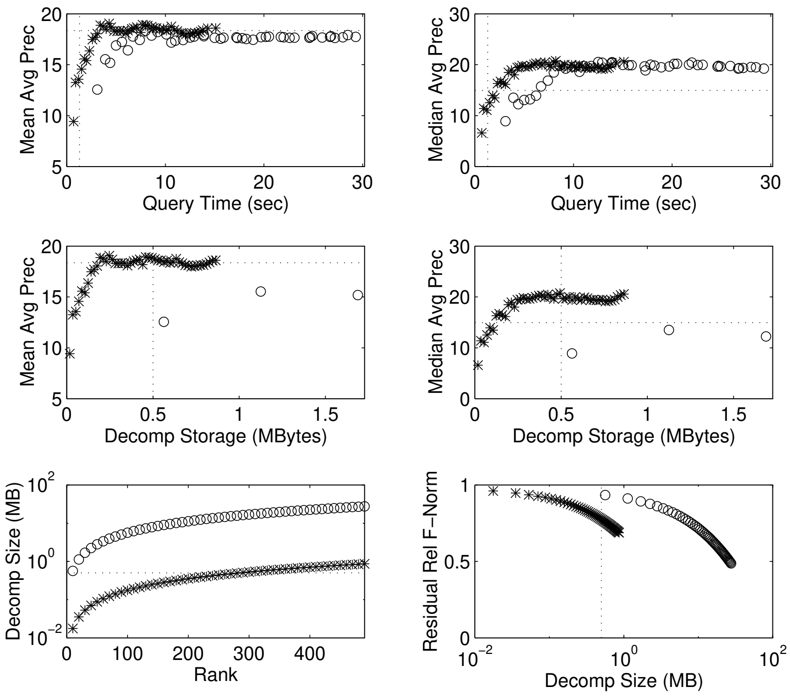

Figure 8.3: A comparison of the SVD (o) and SDD (*) on the CISI data set. We plot 49 data points for each graph, corresponding to $k = 10, 20, \ldots, 490$. The dotted lines show the corresponding data for the vector space method.

the SDD is much smaller.

The results on the three data sets can be summarized as follows: the SDD method is competitive with the SVD method for information retrieval. For equal query times, the SDD method generally has a better mean and median average precision. The SDD requires much less storage and may be the only choice when storage is at a premium. The only disadvantage is the long time required for the initial decomposition, but this is generally a one-time-only expense. Further research should be done on improving the decomposition algorithm.

8.5 Modifying the SDD when the Document Collection Changes

Thus far we have discussed the usefulness of the SDD on a fixed document collection. In practice, it is common for the document collection to be dynamic: new documents are added, and old documents are removed. Thus, the list of terms might also change. In this section, we will focus on the problem of modifying a SDD decomposition when the document collection changes.

SVD-updating has been studied by O'Brien [49]. He reports that updating the SVD takes almost as much time as re-computing it, but that it requires less memory. His methods are similar to what we do in Method 1 in the next section.

8.5.1 Adding or Deleting Documents

Suppose that we have an SDD approximation for a document collection and then wish to add more documents. Rather than compute a new approximation, we will use the approximation from the original document collection to generate a new approximation for the enlarged collection.

Let m_1 and n_1 be the number of terms and documents in the original collection, n_2 be the number of documents added, and m_2 be the number of *new* terms[1]. Let the new document collection be represented as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

where

- A_{11} is an $m_1 \times n_1$ matrix representing the original document collection,
- A_{12} in an $m_1 \times n_2$ matrix representing the new documents indexed by the m_1 terms used in the original collection,

[1]Recall that a term is any word which appears at least twice in the collections and is not a stop word. The addition of new documents may add new terms, some of which may have appeared once in the original document collection.

- A_{21} is an $m_2 \times n_1$ matrix representing the original documents indexed by the newly introduced terms, and

- A_{22} is an $m_2 \times n_2$ matrix representing the new documents indexed by the newly introduced terms.

Assume that $X^{(1)}$, $D^{(1)}$, and $Y^{(1)}$ are the components of the SDD approximation for A_{11}. We propose two methods for updating this decomposition. Each methods is a two-step process: In the first step, we incorporate the new documents using the existing terms, and in the second step, we incorporate the new terms (for both old and new documents).

Method 1: Append rows to $Y^{(1)}$ and $X^{(1)}$. The simplest update method is to keep the existing decomposition fixed and just append new rows corresponding to the new terms and documents. The D will not be recomputed, so the final D is given by

$$D = D^{(1)}.$$

To incorporate the documents (the first step), we want to find $Y^{(2)} \in \mathcal{S}^{n_2 \times k}$ such that

$$\begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \approx X^{(1)} D \begin{bmatrix} Y^{(1)} \\ Y^{(2)} \end{bmatrix}^T .$$

Let $k_{\max}$ be the rank of the decomposition desired; generally this is the same as the rank of the original decomposition. For each value of $k = 1, \ldots, k_{\max}$, we must find the vector y that solves

$$\min_{y \in \mathcal{S}^{n_2}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = A_{12} - X^{(1)}_{k-1} D_{k-1} (Y^{(2)}_{k-1})^T$, x is the kth column of $X^{(1)}$, and d is the kth diagonal element of D. We never access A_{11}, and this may be useful in some situations. The solution y becomes the kth column of $Y^{(2)}$. The final Y is given by

$$Y = \begin{bmatrix} Y^{(1)} \\ Y^{(2)} \end{bmatrix}.$$

To incorporate the terms, we want to find $X^{(2)} \in \mathcal{S}^{m_2 \times k}$ such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \approx \begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix} DY^T.$$

We find $X^{(2)}$ in a analogous way to finding $Y^{(2)}$. For each $k = 1, \ldots, k_{\max}$, we must find the vector x that solves

$$\min_{x \in \mathcal{S}^{m_2}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = \begin{bmatrix} A_{21} & A_{22} \end{bmatrix} - X^{(2)}_{k-1} D_{k-1} (Y_{k-1})^T$, y is the kth column of Y, and d is the kth diagonal element of D. Again, we never access A_{11} for this computation. The final X is given by

$$X = \begin{bmatrix} X^{(1)} \\ X^{(2)} \end{bmatrix}.$$

Method 2: Re-Compute Y and D, then X and D. Another possible method is to completely re-compute Y and D (holding X fixed) to incorporate the documents, and then recompute X and D, holding Y fixed.

Specifically, to incorporate the documents, we first want to find $D^{(2)}$ and Y such that

$$\begin{bmatrix} A_{11} & A_{12} \end{bmatrix} \approx X^{(1)} D^{(2)} Y,$$

where Y has no superscript because it will be the final Y.

To do this, let $k_{\max}$ be the rank of the decomposition desired. For each $k = 1, \ldots, k_{\max}$, we must find the d and y that solve

$$\min_{\substack{d>0 \\ y \in S^n}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = A - X^{(1)}_{k-1} D^{(2)}_{k-1} Y^T_{k-1}$ and x is the kth column of $X^{(1)}$. The solutions d and y become the kth diagonal element of $D^{(2)}$ and the kth column of Y respectively.

To incorporate the documents, we wish to find X and D such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \approx XDY^T.$$

This is similar to how we computed Y and $D^{(2)}$ in the first step. For each $k = 1, \ldots, k_{\max}$, we must find the d and x that solve

$$\min_{\substack{d>0 \\ x \in S^m}} \|A^{(c)} - dxy^T\|_F,$$

where $A^{(c)} = A - X_{k-1} D_{k-1} Y^T_{k-1}$ and y is the kth column of Y. The solutions d and x become the kth diagonal element of D and the kth column of X respectively.

Neither method has any inner iterations, and so both are fast. We tried each update method on a collection of tests derived from the MEDLINE data. We split the MEDLINE document collection into two groups. We did a decomposition on the first group of documents with $k = 100$, then added the second group of documents to the collection, and updated the decomposition via each of the two update methods. The results are summarized in Table 8.4. The second method is better, as should be expected since we are allowing more to change. For the second method, the decrease in mean average precision is not very great when we add only a small number of documents. As the proportion of new documents to old documents grows, however, performance worsens.

If we wish to delete terms or documents, we simply delete the corresponding rows in the X and Y matrices.

8.5.2 Iterative Improvement of the Decomposition

If we have an existing decomposition, perhaps resulting from adding and/or deleting documents and terms, we may wish to improve on this decomposition without actually re-computing it. We consider two approaches.

Documents		Decomp	Method 1		Method 2	
Old	New	Time (Sec)	Time (Sec)	Mean Avg Prec	Time (Sec)	Mean Avg Prec
1033	–	150.5	–	62.18	–	62.18
929	104	138.3	10.5	60.10	13.8	61.83
826	207	122.1	10.4	58.44	13.7	61.80
723	310	103.6	10.2	54.59	13.4	62.46
619	414	94.2	10.2	47.70	13.2	59.28
516	517	77.5	10.1	39.11	12.9	58.76
413	620	60.7	9.9	34.00	12.6	58.83
309	724	45.6	9.5	18.98	12.1	57.19
206	827	26.2	9.6	18.50	11.7	52.29
103	930	14.9	9.4	16.26	11.1	51.38

Table 8.4: Comparison of two update methods on the MEDLINE data set with $k = 100$.

Method 1: Partial Re-Computation In order to improve on this decomposition, we could reduce its rank by deleting 10% of the vectors and then recompute them using our original algorithm. This method's main disadvantage is that it can be expensive in time. If performed on the original decomposition, it has no effect.

Method 2: Fix and Compute. This method is derived from the second update method. We fix the current X and re-compute Y and D; we then fix the current Y and re-compute the X and D. This method is very fast because there are no inner iterations. This can be repeated to further improve the results. If applied to an original decomposition, it would change it.

We took the decompositions resulting from the second update method in the last subsection and applied the improvement methods to them. We have a rank-100 decomposition. For the first improvement method, we re-computed 10 dimensions. For the second improvement method, we applied the method once. The results are summarized in Table 8.5. If we have added only a few documents, the first method improves the precision while the second method worsens it. On the other hand, if we have added many documents, then the second method is much better. The first method could be improved by re-computing more dimensions, but this would quickly become too expensive. The second method greatly improves poor decompositions and is relatively inexpensive. It can be applied repeatedly to further improve the decomposition.

			Method 1		Method 2	
Documents		Prev Mean	Time	Mean	Time	Mean
Old	New	Avg Prec	(Sec)	Avg Prec	(Sec)	Avg Prec
1033	–	62.16	13.5	61.85	22.9	62.16
929	104	61.83	13.4	61.22	20.8	61.45
826	207	61.80	13.5	62.03	19.8	61.51
723	310	62.46	13.4	61.89	21.6	61.91
619	414	59.28	13.6	61.42	19.6	58.70
516	517	58.76	13.5	59.32	19.2	59.43
413	620	58.83	13.4	61.55	20.2	59.68
309	724	57.19	13.6	59.59	20.1	57.94
206	827	52.29	13.4	57.63	21.2	54.35
103	930	51.38	13.4	56.46	22.7	53.88

Table 8.5: Comparison of two improvement methods on the MEDLINE data set with $k = 100$.

Chapter 9

Conclusions

We have explored two applications of limited memory methods. In each case, we considered theoretical and experimental results.

For optimization problems, we characterized which limited-memory quasi-Newton methods fitting a general form (3.1) have the property of producing conjugate search directions on convex quadratics. We showed that limited-memory BFGS is the only Broyden Family member that has a limited-memory analog with this property. We considered update-skipping, something that may seem attractive in a parallel environment. We show that update skipping on quadratic problems is acceptable for full-memory Broyden Family members in that it only delays termination, but that we lose the property of finite termination if we both limit memory and skip updates. Then we introduced simple-to-implement modifications of the standard limited-memory BFGS algorithm that are promising on test problems.

There are a number of directions for future work in this area. The results that have been presented here apply to quasi-Newton methods that use an exact line search, but it would be useful to know what happens with methods that are not perfect. This extends further into the realm of possible limited-memory quasi-Newton methods for solving nonlinear equations. Little work has been done in this area although there is promise for exciting new methods. Other potential work includes developing the hybrid SR1-BFGS method previously mentioned and implementing the new limited-memory methods in parallel.

In the information retrieval application, we introduced a semi-discrete matrix decomposition for use in LSI. We showed that the approximation generated by the SDD converges linearly to the true matrix.

We showed how the SDD can be used to improve the performance of LSI. For equal query times, the SDD-LSI method performs as well as the original SVD-LSI method. The advantage of the SDD method is that the decomposition takes very little storage and the query times are faster; the disadvantage is that the initial time to form the decomposition is large. Since decomposition is a one-time expense, we believe that the SDD-LSI algorithm will be quite useful in application.

We also introduced methods to dynamically change the SDD decomposition if the document collection changes and methods to improve the decomposition if it is found to be inadequate.

The LSI-SDD method is an exciting new approach to information retrieval, and future work in this area would be to develop a full-scale system that can index gigabytes of text.

Appendix A

Line Search Parameters

Table A.1 give the line search parameters used for our code. Note that in the first iteration, the initial steplength is $\|g_0\|^{-1}$ rather than 1.0.

Variable	Value	Description
`STP`	1.0	Step length to try first.
`FTOL`	1.0×10^{-4}	Value of ω_1 in Wolfe conditions.
`GTOL`	0.9	Value of ω_2 in Wolfe conditions.
`XTOL`	1.0×10^{-15}	Relative width of interval of uncertainty.
`STPMIN`	1.0×10^{-15}	Minimum step length.
`STPMAX`	1.0×10^{15}	Maximum step length.
`MAXFEV`	20	Maximum number of function evaluations.

Table A.1: Line Search Parameters

Appendix B

Pseudo-Code

B.1 L-BFGS: Algorithm 0

The pseudo-code for the computation of $d_k = -H_k g_k$ at iteration k for L-BFGS is given in Figure B.2. The update of H is also handled implicitly in this computation.

B.2 Varying m iteratively: Algorithms 1–4

Suppose that m_k denotes the number of (s, y) pairs to be used in the kth update. Then simply chose `sze` as the minimum of `oldsze` + 1 and m_k before computing d_k.

B.3 Disposing of Old Information: Algorithm 5

If the disposal criterion is met at iteration k, set `oldsze` to zero and `sze` to one before computing d_k.

B.4 Backing Up in the Update to H: Algorithms 6-11

If we are to back-up at iterations k, set `oldsze` to the *one less* than the previous value of `sze` and set `sze` as the minimum of `oldsze` + 1 and m, as usual.

B.5 Merging s and y Information in the Update: Algorithms 12 and 13

Merging is the most complicated variation to handle. Before we determine the newest `sze` and before we compute d_k, we execute the pseudo-code given in Figure B.1. We then set `oldsze` to *one less* than the previous value of `sze` and set `sze` as the minimum of `oldsze` +

```
% Compute d_k = -H_k g_k
if (sze == 0)
  d = -g;
else
  % Step 0
  idx = 2 - (sze - oldsze);
  % Step 1
  S = [S(:,idx:oldsze),s]; Y = [Y(:,idx:oldsze),y];
  % This is needed for Step 3 before we overwrite Stg and Ytg
  Stoldg = [Stg(idx:oldsze); s'*oldg];
  Ytoldg = [Ytg(idx:oldsze); y'*oldg];
  % Step 2
  Stg = S'*g; Ytg = Y'*g;
  % Step 3
  Sty = Stg - Stoldg;
  Yty = Ytg - Ytoldg;
  % Step 4
  rho = 1.0/Sty(sze);
  invU = [invU(idx:oldsze,idx:oldsze) - ...
          rho*invU(idx:oldsze,idx:oldsze)*Sty(1:sze-1)
          zeros(1,sze-1) rho];
  % Step 5
  YtY = [YtY(idx:oldsze,idx:oldsze) Yty(1:sze-1)
         (Yty(1:sze-1))' Yty(sze)];
  % Step 6
  D = [D(idx:oldsze), Sty(sze)];
  % Step 7
  gamma = Sty(sze)/Yty(sze);
  % Step 8
  p1 = invU*Stg;
  p2 = invU*(gamma*YtY*p1 + diag(D)*p1 - gamma*Ytg);
  % Step 9
  d = gamma*Y*p1 - S*p2 - gamma*g;
end
```

Figure B.1: MATLAB pseudo-code for the computation of $d = Hg$ in L-BFGS. `sze` is the number of s vectors available for the update this iteration and `oldsze` is the number of s vectors that were available the previous iteration. For L-BFGS, `sze` is chosen as the minimum of `oldsze` $+ 1$ and m (the limited-memory constant).

1 and m, as usual. We are assuming we are at iteration k, but that the newest values of s and y have not yet been added to S and Y.

```
% Execute before choosing new value for sze
% and before computing d
S(:,sze-1) = S(:,sze) + S(:,sze-1);
Y(:,sze-1) = Y(:,sze) + Y(:,sze-1);
Stg(sze-1) = S(:,sze-1)'*g;
Ytg(sze-1) = Y(:,sze-1)'*g;
delta = S(:,sze-1)'*Y(:,sze-1);
rho = 1.0/delta;
invU = ...
  [invU(1:sze-2,1:sze-2) - ...
     rho*invU(1:sze-2,1:sze-2)*S(:,1:sze-2)'*Y(:,sze-1)
  zeros(1,sze-2) rho];
temp = YtY(1:sze-2,sze-1) + YtY(1:sze-2,sze);
YtY = [YtY(1:sze-2,1:sze-2) temp
      temp' Y(:,sze-1)'*Y(:,sze-1)];
D = [D(1:sze-2), delta];
```

Figure B.2: MATLAB pseudo-code for the merge variation. This fixes the values of the components that are used in the computation of d_k.

B.6 Skipping Updates to H: Algorithms 14–16

To skip the update at iteration k, set `sze` to `oldsze`. Compute `Stg` and `Ytg` before Step 0 and then skip to Step 8 and continue.

Bibliography

[1] M. Al-Baali. Descent property and global convergence of the Fletcher-Reeves method with inexact line search. *IMA Journal of Numerical Analysis*, 5:121–124, 1985.

[2] M. W. Berry and R. D. Fierro. Low-rank orthogonal decompositions for information retrieval applications. *Numerical Linear Algebra with Applications*, 1:1–27, 1996.

[3] Michael Berry, Theresa Do, Gavin O'Brien, Vijay Krishna, and Sowmini Varadhan. SVDPACKC (Version 1.0) Users' Guide. Technical Report CS-93-194, Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301, 1993.

[4] Michael W. Berry, Susan T. Dumais, and Gavin W. O'Brien. Using linear algebra for intelligent information retrieval. *SIAM Review*, 37:573–595, 1995.

[5] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. `ftp://thales.math.fundp.ac.be/pub/cute`.

[6] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. `http://www.rl.ac.uk/departments/ccd/numerical/cute/cute.html`.

[7] I. Bongartz, A. R. Conn, Nick Gould, and Ph. L. Toint. CUTE: Constrained and unconstrained testing environment. *ACM Transactions on Mathematical Software*, 21:123–160, 1995.

[8] C. G. Broyden. The convergence of a class of double-rank minimization algorithms, Parts I and II. *Journal of the Institute of Mathematics and Its Applications*, 6:76–90, 222–236, 1970.

[9] A. R. Buckley. Test functions for unconstrained minimization. Technical Report TR 1989CS-3, Mathematics, statistics and computing centre, Dalhousie University, Halifax (CDN), 1989. Cited in [5, 6, 7].

[10] Chris Buckley, Gerald Salton, James Allan, and Amit Singhal. Automatic query expansion using SMART: TREC 3. In D. K. Harman, editor, *Overview of the Third Text REtrieval Conference (TREC-3)*, NIST SP 500-225, pages 69–80, National Institute of Standards and Technology, Gaithersburg, MD, 1995. Also available at `http://potomac.ncsl.nist.gov/TREC`.

[11] R. H. Byrd, J. Nocedal, and Y. Yuan. Global convergence of a class of quasi-Newton methods on convex problems. *SIAM Journal on Numerical Analysis*, 24:1171–1190, 1987.

[12] Richard H. Byrd, Jorge Nocedal, and Robert B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. *Mathematical Programming*, 63:129–156, 1994.

[13] James P. Callan, Bruce Croft, and Stephen M. Harding. The INQUERY retrieval system. In *Proceedings of the Third International Conference on Database and Expert Systems Applications*, pages 78–83. Springer-Verlag, 1992.

[14] A. R. Conn, N. I. M. Gould, M. Lescrenier, and Ph. L. Toint. Performance of a multi-frontal scheme for partially separable optimization. Technical Report 88/4, Department of Mathematics, FUNDP, Namur, Belgium, 1988. Cited in [5, 6].

[15] W. B. Croft. Experiments with representation in a document retrieval system. *Information Technology: Research and Development*, 2:1–21, 1983. Cited in [27].

[16] W. C. Davidon. Variable metric methods for minimization. Technical Report ANL-5990, Argonne National Labs, 1959.

[17] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41:391–407, 1990.

[18] Ron S. Dembo and Trond Steihaug. Truncated-Newton algorithms for large-scale unconstrained optimization. *Mathematical Programming*, 26:190–212, 1983.

[19] J. E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Series in Computational Mathematics. Prentice Hall, 1983.

[20] L. C. W. Dixon. Quasi-Newton algorithms generate identical points. *Mathematical Programming*, 2:383–387, 1972.

[21] Susan Dumais. Improving the retrieval of infomation from external sources. *Behavior Research Methods, Instruments, & Computers*, 23:229–236, 1991.

[22] Susan Dumais. Latent sematic indexing (LSI): TREC-3 report. In D. K. Harman, editor, *Overview of the Third Text REtrieval Conference (TREC-3)*, NIST SP 500-225, pages 219–230, National Institute of Standards and Technology, Gaithersburg, MD, 1995. Also available at `http://potomac.ncsl.nist.gov/TREC`.

[23] R. Fletcher. A new approach to variable metric algorithms. *The Computer Journal*, 13:317–322, 1970.

[24] R. Fletcher. An optimal positive definite update for sparse Hessian matrices. Numerical Analysis NA/145, University of Dundee, 1992. Cited in [5, 6].

[25] R. Fletcher and M. J. D. Powell. A rapidly convergent descent method for minimization. *The Computer Journal*, 6:163–168, 1963.

[26] R. Fletcher and C. M. Reeves. Function minimization by conjugate gradients. *The Computer Journal*, 7:149–154, 1964.

[27] William B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[28] Philip E. Gill and Walter Murray. Conjugate-gradient methods for large-scale non-linear optimization. Technical Report SOL 79-15, Systems Optimization Laboratory, Department of Operations Research, Stanford University, Stanford, California, 94305, 1979.

[29] D. Goldfarb. A family of variational methods derived by variational means. *Mathematics of Computation*, 24:23–26, 1970.

[30] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[31] N. Gould. Private communication to authors of [7]. Cited in [5, 6].

[32] D. K. Harman, editor. *The First Text REtrieval Confernece (TREC-1)*, NIST SP 500-207, National Institute of Standards and Technology, Gaithersburg, MD, 1993.

[33] D. K. Harman, editor. *The Second Text REtrieval Confernece (TREC-2)*, NIST SP 500-215, National Institute of Standards and Technology, Gaithersburg, MD, 1994.

[34] D. K. Harman, editor. *Overview of the Third Text REtrieval Conference (TREC-3)*, NIST SP 500-225, National Institute of Standards and Technology, Gaithersburg, MD, 1995. Also available at `http://potomac.ncsl.nist.gov/TREC`.

[35] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[36] H. Khalfan, R. H. Byrd, and R. B. Schnabel. A theoretical and experimental study of the symmetric rank one update. Technical Report CU-CS-489-90, Department of Computer Science, University of Colorado at Boulder, 1990.

[37] Tamara Kolda, Dianne O'Leary, and Larry Nazareth. `http://www.cs.umd.edu/users/oleary/LBFGS/index.html`, 1996.

[38] Tamara G. Kolda and Dianne P. O'Leary. A semi-discrete matrix decomposition for latent semantic indexing in information retrieval. Technical Report CS-TR-3724/UMIACS-TR-96-92, Computer Science Department, University of Maryland, 1996.

[39] Tamara G. Kolda and Dianne P. O'Leary. Latent semantic indexing via a semi-discrete matrix decomposition. In George Cybenko, Dianne P. O'Leary, and Jorma Rissanen, editors, *The Mathematics of Information Coding, Extraction and Distribution*, IMA Volumes in Math. and Its Applics. Springer-Verlag, New York, 1997. To appear.

[40] Tamara G. Kolda, Dianne P. O'Leary, and Larry Nazareth. BFGS with update skipping and varying memory. *SIAM J. Optimization*, 1997. To appear.

[41] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

[42] David G. Luenberger. *Linear and Nonlinear Programming.* Addison Wesley, 2nd edition, 1984.

[43] Jorge J. Moré, Burton S. Garbow, and Kenneth E. Hillstrom. Testing unconstrained optimization software. *ACM Transactions on Matematical. Software*, 7:17–41, 1981.

[44] S. Nash. Newton-type minimization via the Lanczos process. *SIAM Journal on Numerical Analysis*, 21:770–788, 1984.

[45] Stephen G. Nash and Jorge Nocedal. A numerical study of the limited memory BFGS method and the truncated-Newton method for large scale optimization. *SIAM Journal on Optimization*, 1:358–372, 1991.

[46] Larry Nazareth. On the BFGS method. Univ. of California, Berkeley, 1981.

[47] Jorge Nocedal. Updating quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

[48] Jorge Nocedal. Theory of algorithms for unconstrained optimization. In *Acta Numerica (1991)*, pages 199–242. Cambridge Univ. Press, 1992.

[49] Gavin O'Brien. Information management tools for updating an SVD-encoded indexing scheme. Master's thesis, University of Tennessee, Knoxville, TN 37996-1301, 1994.

[50] Dianne P. O'Leary. A discrete Newton algorithm for minimizing a function of many variables. *Mathematical Programming*, 23:20–33, 1982.

[51] Dianne P. O'Leary and Shmuel Peleg. Digital image compression by outer product expansion. *IEEE Transactions on Communications*, 31:441–444, 1983.

[52] S. S. Oren. Self-scaling variable metric algorithms, Part II: Implementation and experiments. *Management Science*, 20:863–874, 1974. Cited in [5, 6].

[53] M. J. D. Powell. On the convergence of the variable metric algorithm. *Journal of the Institute of Mathematics and Its Applications*, 7:21–36, 1971.

[54] M. J. D. Powell. Quadratic termination properties of minimization algorithms I. Statement and discussion of results. *Journal of the Institute of Mathematics and Its Applications*, 10:333–342, 1972.

[55] M. J. D. Powell. Some global convergence properties of a variable metric algorithm for minimization without exact line searches. In *SIAM-AMS Proceedings*, volume 9, pages 53–72, 1976.

[56] Gerald Salton. Developments in automatic text retrieval. *Science*, 253:974–980, 1991.

[57] Gerald Salton. The state of retrieval system evaluation. Technical Report TR 91-1206, Department of Computer Science, Cornell University, Ithica, NY 14853-7501, 1991.

[58] Gerald Salton and Chris Buckley. Term weighting approaches in automatic text retrieval. *Information Processing and Management*, 24:513–523, 1988.

[59] Gerald Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[60] D. F. Shanno. Conditioning of quasi-Newton methods for function minimization. *Mathematics of Computation*, 24:647–657, 1970.

[61] Amit Singhal, Gerald Salton, Mandar Mitra, and Chris Buckley. Document length normalization. Technical Report TR 95-1529, Deparment of Computer Science, Cornell University, Ithica, NY 14853-7501, 1995.

[62] Ph. L. Toint. An error in specifying problem CHNROSNB. Cited in [5, 6].

[63] Ph. L. Toint. Some numerical results using a sparse matrix updating formula in unconstrained optimization. *Mathematics of Computation*, 32:839–852, 1978.

[64] Ph. L. Toint. Test problems for partially separable optimization and results for the routine PSPMIN. Technical Report 83/4, Department of Mathematics, FUNDP, Namur, Belgium, 1983. Cited in [5, 6, 7].

[65] The UMCP libraries' mission. `http://www.itd.umd.edu/UMS/UMCP/PUB/mission.html`, 1996.