On the Error Analysis and Implementation of Some Eigenvalue Decomposition and Singular Value Decomposition Algorithms

by

Huan Ren

B.S. (Peking University, P.R.China) 1991

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Applied Mathematics

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor James Demmel, Chair
Professor Beresford Parlett
Professor Phillip Colella

1996

Abstract

On the Error Analysis and Implementation of Some Eigenvalue Decomposition and Singular Value Decomposition Algorithms

by

Huan Ren

Doctor of Philosophy in Applied Mathematics

University of California at Berkeley

Professor James Demmel, Chair

Many algorithms exist for computing the symmetric eigendecomposition, the singular value decomposition and the generalized singular value decomposition. In this thesis, we present several new algorithms and improvements on old algorithms, analyzing them with respect to their speed, accuracy, and storage requirements.

We first discuss the variations on the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. We show the challenges in implementing a correct algorithm with floating point arithmetic. We show how reasonable looking but incorrect implementations can fail. We carefully define correctness, and present several implementations that we rigorously prove correct.

We then discuss a fast implementation of bisection using parallel prefix. We show many numerical examples of the instability of this algorithm, and then discuss its forward error and backward error analysis. We also discuss possible ways to stabilize it by using iterative refinement.

Finally, we discuss how to use a divide-and-conquer algorithm to compute the singular value decomposition and solve the linear least squares problem, and how to implement Van Loan's algorithm for the generalized singular value decomposition using this divide-and-conquer algorithm. We show how our implementations achieve good speedups over the previous implementations. For example, on an IBM RS6000/590, our implementation runs 50 times faster than LAPACK's implementation for computing the bidiagonal SVD, and 13 times faster for computing the dense SVD for 1600×1600 random matrices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The symmetric eigenvalue decomposition (SED) and the singular value decomposition (SVD) are two of the most common problems in numerical linear algebra. There are many algorithms to compute these two decompositions. We will discuss two popular algorithms which are suitable for serial computers as well as parallel computers: the bisection algorithm and the divide-and-conquer algorithm.

In this chapter, we introduce some basic concepts and then give an overview of the thesis.

1.2 Basic Concepts

The *eigendecomposition* of an $n \times n$ real symmetric matrix A is

$$A = U \Lambda U^T,$$

where U is an $n \times n$ orthogonal matrix ($U^T U = I$)and $\Lambda \in \mathcal{R}^{n \times n}$ is a diagonal matrix. The columns of U are the *eigenvectors* of A and the diagonal elements of Λ are the *eigenvalues* of A. We assume the eigenvalues are in increasing order:

$$\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n.$$

The *singular value decomposition* (SVD) of an $m \times n$ real matrix B is

$$B = U \Sigma V^T,$$

where $U \in \mathcal{R}^{m\times m}$ and $V \in \mathcal{R}^{n\times n}$ are orthogonal matrices, and $\Sigma \in \mathcal{R}^{m\times n}$ is a nonnegative diagonal matrix. The columns of U are the *left singular vectors* of B, the columns of V are the *right singular vectors*, and the diagonal elements of $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$ are the *singular values*.

There are important relationships between the singular value decomposition of a matrix B and the eigendecomposition of B^TB, BB^T, and $\begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix}$ [43]. In fact, if the SVD of $B \in \mathcal{R}^{m\times n}(m \geq n)$ is given by $B = U\Sigma V^T$, then

$$V^T(B^TB)V = \Sigma^T\Sigma = \mathrm{diag}\ (\sigma_1^2, \ldots, \sigma_n^2) \in \mathcal{R}^{n\times n}$$

and

$$U^T(BB^T)U = \mathrm{diag}\ (\sigma_1^2, \ldots, \sigma_n^2, 0, \ldots, 0) \in \mathcal{R}^{m\times m}.$$

Moreover, if

$$U = [U_1 \quad U_2]$$

where $U_1 \in \mathcal{R}^{m\times n}$ and $U_2 \in \mathcal{R}^{m\times(m-n)}$, and we define the $(m+n) \times (m+n)$ orthogonal matrix Q by

$$Q = \frac{1}{\sqrt{2}} \begin{bmatrix} V & V & 0 \\ U_1 & -U_1 & \sqrt{2}U_2 \end{bmatrix}$$

then

$$Q^T \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} Q = \mathrm{diag}\ (\sigma_1, \ldots, \sigma_n, -\sigma_1, \ldots, -\sigma_n, 0, \ldots, 0).$$

The *linear least squares problem* is to compute the x which minimizes

$$||Ax - b||_2$$

where $A \in \mathcal{R}^{m\times n}$, $b \in \mathcal{R}^m$ and $x \in \mathcal{R}^n$. If $m > n$, we have more equations than unknowns, and the system is *overdetermined.* If $m < n$, the system is *underdetermined.*

The *generalized singular value decomposition* of two matrices, $A \in \mathcal{R}^{m\times n}$ and $B \in \mathcal{R}^{p\times n}$, is a pair of factorizations:

$$A = U\Sigma_1[0 \quad R]Q^T \quad \text{and} \quad B = V\Sigma_2[0 \quad R]Q^T,$$

where $U \in \mathcal{R}^{m\times m}$, $V \in \mathcal{R}^{p\times p}$ and $Q \in \mathcal{R}^{n\times n}$ are orthogonal matrices. R is an $r \times r$ nonsingular and upper triangular matrix, where $r \leq n$ is the rank of $\begin{bmatrix} A \\ B \end{bmatrix}$. Σ_1 is an $m \times r$

diagonal matrix, Σ_2 is a $p \times r$ diagonal matrix, the diagonal elements of both matrices are nonnegative, and they satisfy

$$\Sigma_1^T \Sigma_1 + \Sigma_2^T \Sigma_2 = I.$$

1.3 Floating Point Arithmetic

Each arithmetic operation is generally affected by **roundoff error** because the machine hardware can only represent a subset of real numbers which are called **floating point numbers**. In more detail, let $\otimes$ be one of the operations $+, -, *, /$. When the true value of an operation $a \otimes b$ can not be represented exactly as a floating point number, it must be approximated by a nearby floating point number before it can be stored in memory. We denote this approximation by $fl(a \otimes b)$, and the difference

$$a \otimes b - fl(a \otimes b)$$

is the roundoff error. If $fl(a \otimes b)$ is the nearest floating point number to $a \otimes b$, we say the arithmetic *rounds correctly*. IEEE arithmetic [4, 5] has this attractive property. When rounding correctly and $a \otimes b$ does not overflow or underflow, we can write

$$fl(a \otimes b) = (a \otimes b)(1 + \delta), \tag{1.3.1}$$

where $|\delta|$ is bounded by *machine precision* ε. In IEEE single precision, $\varepsilon = 2^{-24} \approx 6 \cdot 10^{-8}$; in IEEE double precision, $\varepsilon = 2^{-53} \approx 1.1 \cdot 10^{-16}$. The IEEE standard for binary arithmetic [4] is used on SUN, DEC, HP and IBM workstations and all PCs. Exceptions include Cray vector computers, so to accommodate error analysis on a Cray-2, Cray-YMP, or Cray C-90 and Cray T-90, we have to modify our model (1.3.1) to $fl(a \pm b) = a(1 + \delta_1) \pm b(1 + \delta_2)$, $fl(a * b) = (a * b)(1 + \delta_3)$ and $fl(a/b) = (a/b)(1 + \delta_4)$ with $|\delta_i| \leq c \cdot \varepsilon$ where c is a small integer [27, 68].

A *flop* is any floating point operation $a \otimes b$, where a and b are floating point numbers.

1.4 Algorithms to Compute the Symmetric Eigendecomposition

In general, algorithms for computing the symmetric eigendecomposition proceed in three steps [43, 27] except Jacobi methods which is slow.

- **Step 1: Tridiagonalization**

  Given a symmetric matrix $A \in \mathcal{R}^{n \times n}$, find an orthogonal matrix Q such that

  $$QAQ^T = T$$

  where T is a symmetric tridiagonal matrix and Q is the product of Householder matrices.

- **Step 2: Compute Symmetric Tridiagonal Eigendecomposition**

  Use an algorithm to get the eigendecomposition of T

  $$T = U^T \Lambda U.$$

- **Step 3: Computer Eigenvector**

  Compute eigenvector matrix $V = UQ$ of A.

Therefore, the eigendecomposition of A is given by:

$$A = (UQ)^T \Lambda (UQ) = V^T \Lambda V.$$

It can be shown that the computed symmetric tridiagonal matrix $\hat{T}$ from Step 1 satisfies

$$\hat{T} = \hat{Q}^T (A + E) \hat{Q},$$

where $\hat{Q}$ is exactly orthogonal and E is a symmetric matrix satisfying $\|E\|_F \leq c\varepsilon \|A\|_F$ where c is a small constant [97, 43].

Step 2 can be accomplished by many algorithms like tridiagonal QR iteration [80, 27], bisection [43, 27, 80, 30], divide-and-conquer [22, 72, 84, 52, 89], etc. In chapter 2, we will review bisection and discuss several implementations of bisection algorithm. Our goal in chapter 2 is to prove *rigorously* the correctness of those implementations under certain assumptions, e.g. models of floating-point arithmetic.

Bisection can be accelerated by a fast algorithm called *parallel prefix* [26, 28, 32, 94, 74], reducing the time that bisection takes from $O(n)$ to $O(\log_2 n)$ on a machine with n processors and sufficient fast communications. However, it can be very unstable [74, 30]. In chapter 3 and 4, we will introduce the parallel prefix algorithm and show many numerical examples of its instability. We will also analyze its forward and backward stability.

1.5 Singular Value Decomposition and Least Squares Problem

The algorithm to compute singular value decomposition of $A \in \mathcal{R}^{m\times n}$ also takes three steps [41]. The first step is to use orthogonal matrices U and V to reduce A to a bidiagonal matrix B,

$$A = UBV^T,$$

and then compute the SVD of B:

$$B = Q\Sigma W^T.$$

Finally, compute the singular vectors $\tilde{U} = UQ$ and $\tilde{V} = VW$. The SVD of A is then computed as

$$A = (UQ)\Sigma(VW)^T = \tilde{U}\Sigma\tilde{V}.$$

We will describe this process in details in chapter 5. There are also many algorithms to compute the SVD of bidiagonal matrix, like QR-iteration [33, 41, 42], QD-iteration [38, 83] and divide-and-conquer [22, 51, 46, 6, 63].

We will describe the bidiagonal SVD using divide-and-conquer in chapter 5. We will discuss the implementation of the algorithm and present its performance on a high performance workstation, the IBM RS6000/590. For a 1600×1600 random matrix, our implementation achieves a 50-fold speedup over LAPACK's QR-iteration based SVD for step 2, the SVD of a bidiagonal matrix, and a 13-fold speedup for the overall dense SVD.

Given the SVD of A, $A = U\Sigma V^T$, we can solve the least squares problem [43, 27, 2]

$$\min_{x\in\mathcal{R}^n} \|Ax - b\|_2$$

by

$$w = U^T b, \quad y = \begin{bmatrix} \Sigma_1^{-1} & \\ & 0 \end{bmatrix} w, \quad x = Vy,$$

where $\Sigma_1 = \text{diag}\ (\sigma_1, \sigma_2, \ldots, \sigma_r)$ and r is the rank of A. This is the most reliable way to solve the rank deficient linear least squares problem. The other methods like QR factorization and Rank-Revealing QR run faster than the SVD solver, but they are not as reliable when A is rank deficient. We will introduce various least squares solvers in chapter 5 and show their performance on an IBM RS6000/590. We will show that our implementation of a least squares solver using the divide-and-conquer SVD achieves a 28-fold speedup over LAPACK's `DGELSS`, the solver based on the SVD with QR-iteration for a 1600×1600 random matrix.

1.6 Generalized Singular Value Decomposition

We will discuss two ways to compute the generalized singular value decomposition in chapter 6, one based on Paige's algorithm [77] and one on Van Loan's algorithm [96].

Paige's algorithm first reduces the matrix pair A and B to upper triangular form by QR factorization, and then use Jacobi rotations to compute the GSVD of the triangular matrices [77, 10, 2].

Van Loan's algorithm first computes the QR factorization of $\begin{bmatrix} A \\ B \end{bmatrix}$,

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R,$$

and then computes the Cosine-Sine Decomposition (CSD) of $\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix}$,

$$\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 \\ \Sigma_2 \end{bmatrix} V^T.$$

It uses two SVDs in the process of computing the CSD. We implemented Van Loan's algorithm using our implementation of the divide-and-conquer SVD algorithm. We achieve a 54-fold speedup over LAPACK's implementation of Paige's algorithm on an IBM RS6000/590, for 500×500 random matrices. More performance data will be presented in chapter 6.

Chapter 2

Solving the Symmetric Tridiagonal Eigenproblem Using Bisection

2.1 Introduction

Bisection is a well known method for finding eigenvalue of symmetric tridiagonal matrices [43, 27, 80, 66]. We will review it briefly below. Bisection relies on a function which counts the eigenvalues less than x; we call it $\mathrm{Count}(x)$. Our goal in this chapter is to show how an incorrect implementation of $\mathrm{Count}(x)$ can cause failure of the bisection algorithm, and prove the *correctness* of several implementations of $\mathrm{Count}(x)$.

2.2 Review of Bisection

Given an $n \times n$ symmetric tridiagonal matrix

$$T = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-1} & b_{n-1} \\ & & & b_{n-1} & a_n \end{bmatrix}, \tag{2.2.1}$$

the **symmetric tridiagonal eigenproblem** is to find the eigendecomposition:

$$T = U \Lambda U^T,$$

where Λ is a diagonal matrix and U is an orthogonal matrix. The diagonal elements of Λ are the eigenvalues of T, and the columns of U are the corresponding eigenvectors. This is a basic problem in numerical linear algebra [27, 43, 80, 91, 97]. In this chapter, we discuss the **bisection** algorithm [27, 43, 80] to solve this problem.

Before discussing the bisection algorithm, we introduce a classical result upon which it is based.

Definition 2.2.1 The **inertia** of a symmetric matrix A is a triple of integers: $\text{Inertia}(A) \equiv (\mu, \zeta, \pi)$, where μ is the number of negative eigenvalues of A, ζ is the number of zero eigenvalues of A, and π is the number of positive eigenvalues of A.

Theorem 2.2.1 *(***Sylvester's Law of Inertia [40]***) Let A be a symmetric matrix and X be a nonsingular matrix. Then A and X^TAX have the same inertia.*

Suppose $A = A^T$ and one does Gaussian Elimination to get $A - xI = LDL^T$ where L is lower triangular and nonsingular, D is diagonal and I is identity matrix. By Sylvester's Law of Inertia, $\text{Inertia}(A-xI) = \text{Inertia}(D)$. Therefore, $\text{Inertia}(A-xI)$ is trivial to compute since D is diagonal:

$$\begin{aligned}
\text{Inertia}(A - xI) &= (\#\text{ negative eigenvalues of } A - xI, \#\text{ zero eigenvalues of } A - xI, \\
&\qquad \#\text{ positive eigenvalues of } A - xI) \\
&= (\#\text{ eigenvalues of } A < x, \#\text{ eigenvalues of } A = x, \\
&\qquad \#\text{ eigenvalues of } A > x) \\
&= \text{Inertia}(D) \\
&= (\#\ d_{ii} < 0, \#\ d_{ii} = 0, \#\ d_{ii} > 0)
\end{aligned}$$

Suppose $x_1 < x_2$ and we compute $\text{Inertia}(A - x_1 I)$ and $\text{Inertia}(A - x_2 I)$. Then the number of eigenvalues in the interval $[x_1, x_2)$ equals

$$(\#\text{eigenvalues of } A < x_2) - (\#\text{eigenvalues of } A < x_1).$$

By defining the following function(also called Negcount)

$$\text{Count}(x) = \#\text{ eigenvalues of } A < x,$$

we can introduce the following algorithm:

Algorithm 2.2.1 Bisection: Compute all the eigenvalues of A in the interval $[left, right)$ to the desired accuracy τ, given the initial task $(left, right, n_{left}, n_{right})$ where $n_{left} = \mathrm{Count}(left)$, $n_{right} = \mathrm{Count}(right)$.

```
if (n_left = n_right or left > right) return; /* no eigenvalues in interval */
enqueue (left, right, n_left, n_right) on Worklist;
while (Worklist is not empty)
    dequeue (α, β, n_α, n_β) from Worklist;
    if ( β − α small enough ) then
        print "Eigenvalue (α + β)/2 has multiplicity n_β − n_α";
    else
        mid = (α + β)/2;
        n_mid = Count(mid);
        if (n_mid > n_α) then
        /* bottom half of interval (α, β) contains eigenvalues */
            enqueue (α, mid, n_α, n_mid) on Worklist;
        end if
        if (n_mid < n_β) then
        /* top half of interval (α, β) contains eigenvalues */
            enqueue (mid, β, n_mid, n_β) on Worklist;
        end if
    end if
end while
```

In general, we say $\beta - \alpha$ small enough if

$$\beta - \alpha < \min(\tau, \max(|\alpha|, |\beta|)\varepsilon),$$

where τ is used to bound absolute error and $\max(|\alpha|, |\beta|)\varepsilon$ is used to bound relative error.

Let $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ be the eigenvalues of $n \times n$ matrix A. The same idea can be used to compute λ_j for $j = j_0, j_0 + 1, \ldots, j_1$. This is because we know $\lambda_{n_{left}}$ through $\lambda_{n_{right}-1}$ lie in the interval $[left, right)$. Also, by using the Gershgorin Disk Theorem, it is easy to see that all eigenvalues must lie in the interval $[left, eight]$ where

$$left = \min_{1 \leq i \leq n}(A_{ii} - \sum_{j \neq i} |A_{ij}|) \quad \text{and} \quad right = \max_{1 \leq i \leq n}(A_{ii} + \sum_{j \neq i} |A_{ij}|),$$

so that if we let the initial task on the $Worklist$ be $(left, right, 0, n)$, then bisection will compute all the eigenvalues of A.

Theorem 2.2.2 (Gershgorin Disk Theorem [27, 43]) *Let A be an $n \times n$ arbitrary matrix. Then the eigenvalues λ of A are located in the union of the n disks.*

$$|\lambda - A_{kk}| \leq \sum_{j \neq k} |A_{kj}|$$

If A were dense, we could implement Count(x) by doing symmetric Gaussian Elimination [16] with pivoting. But this would cost $O(n^3)$ flops and thus not be cost effective. On the other hand, Count(x) is quite simple to compute for symmetric tridiagonal matrix T:

$$\begin{aligned} T - xI &= \begin{bmatrix} a_1 - x & b_1 & & & \\ b_1 & a_2 - x & b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-1} - x & b_{n-1} \\ & & & b_{n-1} & a_n - x \end{bmatrix} = LDL^T \\ &= \begin{bmatrix} 1 & & & & \\ l_1 & 1 & & & \\ & \ddots & \ddots & & \\ & & l_{n-2} & 1 & \\ & & & l_{n-1} & 1 \end{bmatrix} \begin{bmatrix} d_1 & & & & \\ & d_2 & & & \\ & & \ddots & & \\ & & & d_{n-1} & \\ & & & & d_n \end{bmatrix} \begin{bmatrix} 1 & l_1 & & & \\ & 1 & l_2 & & \\ & & \ddots & & \\ & & & 1 & l_{n-1} \\ & & & & 1 \end{bmatrix} \end{aligned}$$

Therefore, $d_1 = a_1 - x$, $d_1 l_1 = b_1$ and thereafter $l_{i-1}^2 d_{i-1} + d_i = a_i - x$, $d_i l_i = b_i$.

Substituting $l_i = b_i / d_i$ yields:

$$\begin{aligned} d_1 &= a_1 - x \\ d_i &= a_i - x - \frac{b_{i-1}^2}{d_{i-1}} \end{aligned}$$

Thus, given an n-by-n real symmetric tridiagonal matrix T with diagonals $a_1, ..., a_n$ and offdiagonals $b_1, ..., b_{n-1}$ (we let $b_0 \equiv 0$), and let $\lambda_1 \leq \cdots \leq \lambda_n$ be T's eigenvalues, then the following algorithm computes the function Count(x) which returns the number of eigenvalues of T that are less than x (for all but the finite number of x resulting in a divide by zero, which we call *singular points*) :

Algorithm 2.2.2 Count(x) returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.

Count $= 0$;
$d = 1$;
for $i = 1$ *to* n
  $d = a_i - x - b_{i-1}^2/d$
  if $d < 0$ **then**
    Count $=$ Count $+ 1$
  end if
end for

(If we wish to emphasize that T is the argument, we will write Count(x, T) *instead.)*

Remark 2.2.1 We define Count(x) at singular points to be the number of eigenvalues less than x. This means that for all x, Count(x) is left continuous at x.

The cost of a single call to Algorithm 2.2.2 is $4n$. Therefore, the overall cost to find m eigenvalues is $O(mn)$. To compute the corresponding eigenvectors, we can use inverse iteration [43, 80, 81, 61].

2.3 Our Goals in This Chapter

The logic of bisection algorithm seems to depend on the simple fact that Count(x) is a monotonic increasing step function of x, since the number of the eigenvalues in the half-open interval $[\sigma_1, \sigma_2)$ is Count(σ_2) $-$ Count(σ_1). If its computer implementation, call it FloatingCount(x), were not also monotonic, so that one could find $\sigma_1 < \sigma_2$ with FloatingCount(σ_1) $>$ FloatingCount(σ_2), then the computer implementation might well report that the interval $[\sigma_1, \sigma_2)$ contains a negative number of eigenvalues, namely FloatingCount(σ_2) $-$FloatingCount(σ_1). This result is clearly incorrect. In section 2.5 below, we will see that this can indeed occur using the the EISPACK routine `bisect` (using IEEE floating point standard arithmetic [4, 5], and without over/underflow). There are at least four reasons why FloatingCount(x) might not be monotonic [30]:

1. the floating point arithmetic is too inaccurate,

2. over/underflow occurs, or is avoided improperly,

3. FloatingCount(x) is implemented using a fast parallel algorithm called parallel prefix, or

4. heterogeneity — processors in a parallel environment may have differing floating point arithmetics, or may just compile code slightly differently.

In this chapter, we discuss the first two challenges to implementing bisection correctly. In next chapter, we discuss parallel prefix. For heterogeneity and other parallel issues, we refer to [30].

We first give an example to show how monotonicity can fail, and cause incorrect eigenvalues to be computed; see sections 2.5 and 2.6.

We then show that as long as the floating point arithmetic is monotonic (we define this in section 2.4.1), and FloatingCount(x) is implemented on a single processor in a reasonable way, then FloatingCount(x) is also monotonic. A sufficient condition for floating point to be monotonic is that it be correctly rounded or correctly chopped; thus IEEE floating point arithmetic is monotonic. This result was first proven but not published by Kahan in 1966 for symmetric tridiagonal matrices [66]; here we extend this result to *symmetric acyclic matrices*, a larger class including tridiagonal matrices, arrow matrices, and exponentially many others [31]; see section 2.6.

Finally, we review the roundoff error analysis of FloatingCount(x), and how to account for over/underflow, thus rigorously prove several implementations of FloatingCount function are *correct* (we will define *correctness* in section 2.4.3), which is a necessary condition for an implementation of bisection algorithm (serial or parallel) to be correct [30]; part of this material may also be found in [31, 66]; see section 2.7.

2.4 Definitions and Assumptions

Section 2.4.1 defines the kinds of matrices whose eigenvalue problems we will consider, what monotonic arithmetic is, and what "jump points" of the functions Count(x) and FloatingCount(x) are. Section 2.4.2 presents our (mild) assumptions about floating point arithmetic and the input matrices our algorithms will accept. Section 2.4.3 lists the criteria an implementation of FloatingCount must satisfy to be correct.

2.4.1 Preliminary Definitions

In this section, we define some fundamental concepts which we will use through this chapter.

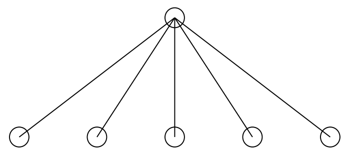
Symmetric Acyclic Matrices

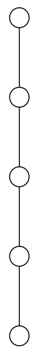
Algorithm 2.2.2 was recently extended to the larger class of *symmetric acyclic matrices* [31], i.e. those matrices whose graphs are acyclic (trees). The undirected graph $G(T)$ of a symmetric n-by-n matrix T is defined to have n nodes and an edge (i, j), $i < j$, if and only if $T_{ij} \neq 0$. A symmetric tridiagonal matrix is one example of a symmetric acyclic matrix; its graph is a chain. An "arrow matrix" which is nonzero only on the diagonal, in the last row and in the last column, is another example; its graph is a star.

$$\begin{bmatrix} \times & & & & & \times \\ & \times & & & & \times \\ & & \times & & & \times \\ & & & \times & & \times \\ & & & & \times & \times \\ \times & \times & \times & \times & \times & \times \end{bmatrix} \qquad \begin{bmatrix} \times & \times & & & & \\ \times & \times & \times & & & \\ & \times & \times & \times & & \\ & & \times & \times & \times & \\ & & & \times & \times & \times \\ & & & & \times & \times \end{bmatrix}$$

Arrow Matrix

Tridiagonal Matrix





From now on, we will assume T is a symmetric acyclic matrix unless we state explicitly otherwise. Also we will number the rows and columns of T in preorder such that node 1 is the root of the tree and so accessed first; node j is called a *child* of node i if $T_{ij} \neq 0$ and node j is visited after node i by the algorithm (see Algorithm 2.6.1 in section 2.6, TreeCount, for details). We let C denote the maximum number of children of any node in the acyclic graph $G(T)$(C is never larger than the degree of $G(T)$).

Monotonic Floating Point Arithmetic

To describe the monotonicity of FloatingCount(x), we need to define *monotonic arithmetic*: An implementation of floating point arithmetic is monotonic if, whenever a, b, c and d are floating point numbers, $\otimes$ is any binary operation, and the floating point results $fl(a \otimes b)$ and $fl(c \otimes d)$ do not overflow, then $a \otimes b \geq c \otimes d$ implies $fl(a \otimes b) \geq fl(c \otimes d)$. This is satisfied by any arithmetic that rounds or truncates correctly such as IEEE floating point arithmetic [4] but not by the floating point arithmetic of Cray XMP or YMP [68]. In section 2.6, we will prove that the FloatingCount function (Floating TreeCount) for a symmetric acyclic matrix is monotonic if the floating point arithmetic is monotonic.

Jump Points

We now define a *jump-point* of the function Count(x). λ_i is the i^{th} jump-point of the function Count(x) if

$$\lim_{x \to \lambda_i^-} \text{Count}(x) < i \leq \lim_{x \to \lambda_i^+} \text{Count}(x)$$

Note that λ_i is actually an eigenvalue of the input matrix T for Count(x). Analogous to the above definition, we define an i^{th} jump-point of a possibly nonmonotonic function FloatingCount(x) as a floating point number λ_i'' such that

$$\text{FloatingCount}(\lambda_i'') < i \leq \text{FloatingCount}(\textit{nextafter}(\lambda_i''))$$

where $\textit{nextafter}(\lambda_i'')$ is the smallest floating point number greater than λ_i''. For a nonmonotonic FloatingCount(x) function, there may be more than one such jump-point.

2.4.2 Assumptions Required to Prove Correctness of Bisection

In order to prove correctness of the algorithms, we need to make some assumptions about the computer arithmetic and the inputs. The following is a list of all the assumptions we will make; not all our results require all the assumptions, so we must be explicit about which assumptions we need.

The first set of assumptions, Assumption 1, concerns the floating point arithmetic. Not all parts of Assumption 1 are necessary for all later results, so we will later refer to Assumptions 1A, 1B, etc. Assumption 2 is about the input matrix, and includes a mild restriction on its size, and an easily enforceable assumption on its scaling.

Assumption 1 (Properties of Floating Point Arithmetic)

1A. Barring overflow, the usual expression for roundoff may be extended to include underflow as follows [25]:

$$fl(a \otimes b) = (a \otimes b)(1 + \delta) + \eta \tag{2.4.2}$$

where $\otimes$ is a binary arithmetic operation, $|\delta|$ is bounded by machine precision ε, $|\eta|$ is bounded by a tiny number $\bar{\omega}$, typically the underflow threshold ω (the smallest normalized number which can safely participate in, or be a result of, any floating point operation)[1] and at most one of δ and η can be nonzero. In IEEE arithmetic, gradual underflow lets us further assert that $\bar{\omega} = \varepsilon\omega$, and that if $\otimes$ is addition or subtraction, then η must be zero. We denote the overflow threshold of the computer (the largest number which can safely participate in, or be a result of, any floating point operation) by Ω.

In this paper, we will consider the following three variations on this basic floating point arithmetic model:

Model 1. $fl(a \otimes b) = (a \otimes b)(1 + \delta) + \eta$ as above, and overflows terminate, i.e. the machine stops executing the running program.

Model 2. IEEE arithmetic with $\pm\infty$, ± 0 and NaN, and with gradual underflow.

Model 3. IEEE arithmetic with $\pm\infty$, ± 0 and NaN, but with underflow flushing to zero instead of gradual underflow.

1B. $\sqrt{\omega} \le \varepsilon \le 1 \le 1/\varepsilon \le \sqrt{\Omega}$. This mild assumption is satisfied by all commercial floating point arithmetics.

1C. Floating point arithmetic is monotonic. This is true of IEEE arithmetic (Models 2 and 3) but is not true of all arithmetics satisfying Model 1. We don't know any commercial machine which satisfies Model 1 but violates this assumption.

"Indeed, the builder of any machine which failed to satisfy this assumption should be ashamed of himself." — Kahan [66].

Assumption 2 (Properties of the input matrix)

[1] These caveats about "safe participation in *any* floating point operation" take machines like some Crays into account, since they have "partial overflow". On Cray, there are numbers for which addition by 1 does not cause overflow although multiplication by 1 does [68].

Table 2.1: Parameters for Different Arithmetics

Parameters	IEEE				Cray
	Single	Single Extended	Double	Double Extended	
ε	$5.96 \cdot 10^{-8}$	$\leq 2.33 \cdot 10^{-10}$	$1.11 \cdot 10^{-16}$	$\leq 5.42 \cdot 10^{-20}$	$3.55 \cdot 10^{-15}$
ω	$1.18 \cdot 10^{-38}$	$\leq 2.23 \cdot 10^{-308}$	$2.23 \cdot 10^{-308}$	$\leq 3.36 \cdot 10^{-4932}$	$3.36 \cdot 10^{-4932}$
Ω	$3.40 \cdot 10^{38}$	$\geq 1.79 \cdot 10^{308}$	$1.79 \cdot 10^{308}$	$\geq 1.19 \cdot 10^{4932}$	$1.19 \cdot 10^{4932}$

2A. Assumption on the problem size n: $n\varepsilon \leq 0.1$. For example, in IEEE double precision, this limits us to matrices of dimension less than $4.5 \cdot 10^{14}$, or 450 trillions. Virtually all numerical algorithms share a restriction like this.

2B. Assumptions on the scaling of the input matrix. Let $\bar{B} \equiv \min_{i \neq j} T_{ij}^2$ and $\bar{M} \equiv \max_{i,j} |T_{ij}|$.

i. $\bar{M} \leq \sqrt{\Omega}$ (largest matrix entry not too large).

ii. $\bar{B} \geq \omega$ (smallest offdiagonal matrix entry not too small).

These assumptions may be achieved by explicitly scaling the input matrix (multiplying it by an appropriate scalar) to adjust $\bar{M}$, and then setting small off-diagonal elements $T_{ij}^2 < \omega$ to zero and so splitting the matrix into *unreduced* blocks [30, 8], each of which satisfies $\bar{B} \geq \omega$. By Weyl's Theorem [80], this may introduce a tiny error of amount no more than $\sqrt{\omega}$ in the computed eigenvalues.

2C. More assumptions on the scaling of the input matrix (that the largest entry is not too small). These are used to get refined error bounds in section 2.7.

i. $\bar{M} \geq \omega/\varepsilon$.

ii. $\bar{M} \geq 1/(\varepsilon\Omega)$.

To end this section, we show a table of ε, ω and Ω for some arithmetics (table 2.1) [4, 68].

2.4.3 Definition of Correctness of FloatingCount

When we say that an implementation of FloatingCount function is *correct*, we assert that it terminates and the following hold:
Let $\lambda_i^{(1)''}, \lambda_i^{(2)''}, \ldots, \lambda_i^{(k)''}$ be the i^{th} jump-points of FloatingCount(x) and λ_i be the i^{th} jump-points of Count(x). We assume that FloatingCount(x) satisfies the error bound,

$$|\lambda_i^{(j)''} - \lambda_i| \leq \xi_i, \quad \forall j = 1, \ldots, k$$

for some small $\xi_i \geq 0$ (usually we require ξ_i to be $O(\varepsilon)$).

We have permitted that FloatingCount(x) to have a bounded region of possible nonmonotonicity, where the error bound ξ_i is also a bound on the nonmonotonicity around eigenvalue λ_i. Different implementations of FloatingCount(x) result in different values of ξ_i (see section 2.7). For some of the practical FloatingCount functions in use, we prove in section 2.7 that they satisfy the correctness property.

We say that an implementation of FloatingCount function is *incorrect* when the above fails to hold.

2.5 An Incorrect Implementation of Bisection

We give an example of the failure of EISPACK's `bisect` routine [88] which implements a nonmonotonic FloatingCount(x). Suppose we use IEEE standard double precision floating point arithmetic [4, 5] with $\varepsilon = 2^{-53} \approx 1.1 \cdot 10^{-16}$ and we want to find the eigenvalues of the following 2×2 matrix:

$$A = \begin{pmatrix} 0 & \varepsilon \\ \varepsilon & 1 \end{pmatrix}.$$

A has eigenvalues near 1 and $-\varepsilon^2 \approx -1.23 \cdot 10^{-32}$. But `bisect` reports that the interval $[-10^{-32}, 0)$ contains -1 eigenvalues. No overflow or underflow occurs in this case. The reason for this is `bisect`'s incorrect provision against division by zero (See Algorithm 2.7.1 in section 2.7.2). In section 2.6, in the proof of Theorem 2.6.1, we will show that this cannot happen for the LAPACK routine `dstebz` even for more general symmetric acyclic matrices.

2.6 Proof of Monotonicity of FloatingCount(x)

In 1966 Kahan proved but did not publish the following result [66]: if the floating point arithmetic is monotonic, then FloatingCount(x) is a monotonically increasing function of x for symmetric tridiagonal matrices. That monotonic floating point arithmetic is *necessary* for FloatingCount(x) to be monotonic is easily seen by considering 1-by-1 matrices: if addition fails to be monotonic so that $x < x'$ but $fl(a_1 - x) < 0 < fl(a_1 - x')$, then $\text{FloatingCount}(x) = 1 > 0 = \text{FloatingCount}(x')$. In this section, we will extend this proof of monotonicity of FloatingCount(x) to symmetric acyclic matrices.

As we mentioned before, Algorithm 2.2.2 was recently extended to the symmetric acyclic matrices. In [31] an implementation of Count(x) for acyclic matrices was given, see Algorithm 2.6.1 below. The algorithm refers to the tree $G(T)$ which is the graph of the $n \times n$ symmetric matrix T, where node 1 is chosen (arbitrarily) as the root of the tree, and node j is called a *child* of node i if $T_{ij} \neq 0$ and node j has not yet been visited by the algorithm. We are also explicit about where roundoff occurs in the algorithm.

Algorithm 2.6.1 TreeCount(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

call $\text{TreeCount}(1, x, d_1, s_1)$
$\text{Count} = s_1$

procedure $\text{TreeCount}(i, x, d_i, s_i)$
    /* *i and x are inputs, d_i and s_i are outputs* */
    $d_i = fl(T_{ii} - x)$
    $s_i = 0$
    for *all children j of i* do
        call $\text{TreeCount}(j, x, d_j, s_j)$
        $d_i = fl(d_i - fl(T_{ij}^2/d_j))$
        $s_i = s_i + s_j$
    endfor
    if $d_i < 0$ then
        $s_i = s_i + 1$
    end if

end TreeCount

Without loss of generality, from now on we ignore roundoff in computing T_{ij}^2 since we may as well consider T_{ij}^2 as the input data (see assumption 2B).

Clearly, s_i is the total number of negative d_j in the subtree rooted at i (including d_i). We may summarize Algorithm 2.6.1 more briefly by

$$d_i = fl(fl(T_{ii} - x) - fl(\sum_{j \in C(i)} fl(\frac{T_{ij}^2}{d_j}))) \tag{2.6.3}$$

$$s_i = \sum_{j \in C(i)} s_j + \begin{cases} 0 & \text{if } d_i \geq 0 \\ 1 & \text{if } d_i < 0 \end{cases} \tag{2.6.4}$$

where the sums are over the set $C(i)$ of all children of i.

Let x be a floating point number, and let x' denote the *next floating point number larger than* x. To distinguish the results of Algorithm 2.6.1 for different x we will s_i and d_i as functions $s_i(x)$ and $d_i(x)$. The theorem we wish to prove is:

Theorem 2.6.1 *If the floating point arithmetic used to implement Algorithm 2.6.1 is monotonic, then $s_i(x) \leq s_i(x')$, i.e.* TreeCount(x) *is monotonic.*

We introduce some more definitions. In these definitions, y is always a floating point number.

Definition 2.6.1 Zeros and Poles:

i. The number y is a *zero* of d_i if $d_i(y) \geq 0 > d_i(y')$.

ii. The number y is a *pole* of d_i if $d_i(y) < d_i(y')$.

- It is called a *positive pole* if in addition to being a pole $d_i(y)d_i(y') > 0$ or $d_i(y) = 0$.
- It is called a *negative pole* if in addition to being a pole $d_i(y)d_i(y') < 0$ or $d_i(y') = 0$.

Now we suppose that for some i, $s_i(x) > s_i(x')$ is decreasing, we want to find a contradiction.

Lemma 2.6.1 *Let m be the largest m such that s_m ever decreases. This means that for some y, $s_m(y) > s_m(y')$. Then in fact $d_m(y) < 0 \leq d_m(y')$, i.e. y is a negative pole of d_m.*

Proof. Since m is the largest integer for which s_m is decreasing, we must have $s_k(y) \leq s_k(y')$ for all children k of m. Now write

$$\begin{aligned} 0 &> s_m(y') - s_m(y) \\ &= \{s_m(y') - \sum_{k \in C(m)} s_k(y')\} + \{\sum_{k \in C(m)} s_k(y') - \sum_{k \in C(m)} s_k(y)\} \\ &+ \{\sum_{k \in C(m)} s_k(y) - s_m(y)\} \\ &\equiv t_1 + t_2 + t_3 \ . \end{aligned}$$

From (2.6.4) we we conclude $t_1 \geq 0$ and $t_3 \geq -1$. From the definition of m we conclude $t_2 \geq 0$. These inequalities have one solution, namely $t_1 = t_2 = 0$ and $t_3 = -1$. From $t_1 = 0$ we conclude that $d_m(y') \geq 0$, and from $t_3 = -1$ we conclude $d_m(y) < 0$. In particular, this means y is a *negative pole* of d_m. ∎

Lemma 2.6.2 *If y is a pole of d_i, then i must have a child j for which y is either a positive pole or a zero.*

Proof. If y is a pole of d_i, then for some child j of i we must have

$$fl(\frac{T_{ij}^2}{d_j(y)}) > fl(\frac{T_{ij}^2}{d_j(y')}) \tag{2.6.5}$$

Otherwise all children would satisfy

$$fl(\frac{T_{ij}^2}{d_j(y)}) \leq fl(\frac{T_{ij}^2}{d_j(y')})$$

and so by the monotonicity of arithmetic

$$fl(\sum_{j \in C(i)} fl(\frac{T_{ij}^2}{d_j(y)})) \leq fl(\sum_{j \in C(i)} fl(\frac{T_{ij}^2}{d_j(y')}))$$

Arithmetic monotonicity further implies

$$fl(T_{ii} - y) \geq fl(T_{ii} - y')$$

and finally

$$fl(fl(T_{ii} - y) - fl(\sum_{j \in C(i)} fl(\frac{T_{ij}^2}{d_j(y)}))) \geq fl(fl(T_{ii} - y') - fl(\sum_{j \in C(i)} fl(\frac{T_{ij}^2}{d_j(y')})))$$

or $d_i(y) \geq d_i(y')$, contradicting the assumption that y is a pole. Applying arithmetic monotonicity to (2.6.5) we conclude

$$\frac{T_{ij}^2}{d_j(y)} > \frac{T_{ij}^2}{d_j(y')} \quad .$$

This means either $d_j(y) \geq 0 > d_j(y')$ (i.e. y is a zero of d_j) or $d_j(y) < d_j(y')$ and $d_j(y) \cdot d_j(y') > 0$ (y is a positive pole of d_j) or $0 = d_j(y) < d_j(y')$ (y is a positive pole of d_j). ∎

Remark 2.6.1 The proof of the last lemma does not depend on the order in which the additions and subtractions of T_{ii}, y, and T_{ij}^2/d_j are carried out. It is also not damaged by inserting the line "if $|d_i| < tol$ then $d_i = -tol$" just before "if $d_i < 0$ then $s_i = s_i + 1$" in Algorithm 2.6.1 since we can simply modify the definition of zero to: y is a zero of d_i if $d_i(y) \geq tol$ *and* $d_i(y') \leq -tol$ (in fact, this is the definition in [66] to prove the monotonicity of Count(x) for symmetric tridiagonal matrix), and the proof will follow through. This is done in practice to avoid overflow and division by zero; see Algorithm 2.7.3 and [3, 66]. However, the proof does *not* work for the algorithm used to avoid overflow in the subroutine `bisect` [88]. This is because `bisect` tests if a computed d_j is exactly zero, and increases if it is; this can increase $d_i(y')$ past $d_i(y)$ even if inequalities (2.6.5) are not satisfied. The example in section 2.5 shows that monotonicity can indeed fail in practice.

Lemma 2.6.3 *If y is a pole of d_i, then there must be a node l in the subtree rooted at i such that y is a zero of d_l and for all d_j on the path from i to l, y is a positive pole of d_j.*

Proof. We can apply Lemma 2.6.2 to i to find a child l which is either a zero or a positive pole. If it is a zero we are done, and otherwise we apply Lemma 2.6.2 again to l. This process must end in a zero since the leaves are of the form $d_l(x) = fl(T_{ll} - x)$ and so can only be zeros by arithmetic monotonicity. ∎

Proof of **Theorem** *2.6.1:* We use proof by contradiction, assume that some $s_i(x)$ is not monotonic, we now use Lemma 2.6.1 to conclude that there is a largest m such that y is a negative pole of d_m, and

$$\sum_{k \in C(m)} s_k(y) = \sum_{k \in C(m)} s_k(y') \quad . \tag{2.6.6}$$

Use Lemma 2.6.3 to conclude that there is some l in the tree rooted at m for which y is a zero. This means $d_l(y) \geq 0 > d_l(y')$, so that d_l contributes one more to the right hand side of (2.6.6) than to the left hand side. So to maintain (2.6.6) there must be another p in the tree rooted at m with $d_p(y) < 0 \le d_p(y')$, i.e. y is a negative pole of d_p. By Lemma 2.6.3, p cannot lie on the path from m to l, since only positive poles lie on this path. Therefore, again by Lemma 2.6.3, there must be a $q \neq l$ in the tree rooted at p such that y is a zero of d_q. But this means d_p and d_q together contribute equally to both sides of (2.6.6), and so cannot balance d_l. By the same argument, any other negative pole which would balance d_l has a counterbalancing zero. Therefore (2.6.6) cannot be satisfied. This contradiction proves Theorem 2.6.1. ∎

2.7 Roundoff Error Analysis

In last section, we introduced Algorithm 2.6.1 which can be used to compute the eigenvalues of a symmetric acyclic matrix. In [31] it is shown that barring over/underflow, the floating point version of Algorithm 2.6.1 has the same attractive backward error analysis as the floating point version of Algorithm 2.2.2. We reproduce the error analysis in [31]. Results are summarized in tables 2.4 and 2.5.

Let FloatingCount(x, T) denote the value of Count(x, T) computed in floating point arithmetic. Then FloatingCount(x, T) = Count(x, T'), where T' differs from T only slightly:

$$|T_{ij} - T'_{ij}| \le f(C/2+2, \varepsilon)|T_{ij}| \ \text{ if } \ i \neq j \ \text{ and } \ T_{ii} = T'_{ii}, \tag{2.7.7}$$

where ε is the machine precision, C is the maximum number of children of any node in the graph $G(T)$ and $f(n, \varepsilon)$ is defined by

$$f(n, \varepsilon) = (1+\varepsilon)^n - 1.$$

By Assumption 2A ($n\varepsilon \le .1$), we have [97]:

$$f(n, \varepsilon) \le 1.06 n\varepsilon.$$

(Strictly speaking, the proof of this bound is a slight modification of the one in [31], and requires that d be computed exactly as shown in TreeCount. The analysis in [31] makes no assumption about the order in which the sum for d is evaluated, whereas the bound (2.7.7) for TreeCount assumes the parentheses in the sum for d are respected. Not respecting the parentheses weakens the bounds just slightly, and complicates the discussion below, but does not change the overall conclusion.)

This tiny componentwise backward error permits us to compute the eigenvalues accurately, as we now discuss. Suppose the backward error in (2.7.7) can change eigenvalue λ_k by at most ξ_k. For example, Weyl's Theorem [80] implies that $\xi_k \le \|T - T'\|_2 \le 2f(C/2+2,\varepsilon)\|T\|_2$, i.e. that each eigenvalue is changed by an amount small compared with the largest eigenvalue. If $T_{ii} = 0$ for all i, then $\xi_k \le ((1-(C+4)\varepsilon)^{1-n} - 1)|\lambda_k|$, i.e. each eigenvalue is changed by an amount small relative to itself. See [12, 33, 66] for more such bounds.

Now suppose that at some point in the algorithm we have an interval $[x, y)$, $x < y$, where

$$i = \mathrm{FloatingCount}(x, T) < \mathrm{FloatingCount}(y, T) = j \ . \qquad (2.7.8)$$

Let T'_x be the equivalent matrix for which $\mathrm{FloatingCount}(x, T) = \mathrm{Count}(x, T'_x)$, and T'_y be the equivalent matrix for which $\mathrm{FloatingCount}(y, T) = \mathrm{Count}(y, T'_y)$, Thus $x \le \lambda_{i+1}(T'_x) \le \lambda_{i+1}(T) + \xi_{i+1}$, or $x - \xi_{i+1} \le \lambda_{i+1}(T)$. Similarly, $y > \lambda_j(T'_y) \ge \lambda_j(T) - \xi_j$, or $\lambda_j(T) < y + \xi_j$. Altogether,

$$x - \xi_{i+1} \le \lambda_{i+1}(T) \le \lambda_j(T) < y + \xi_j \ . \qquad (2.7.9)$$

If $j = i + 1$, we get the simpler result

$$x - \xi_j \le \lambda_j(T) < y + \xi_j \ . \qquad (2.7.10)$$

This means that by making x and y closer together, we can compute $\lambda_j(T)$ with an accuracy of at best about $\pm\xi_j$; this is when x and y are adjacent floating point numbers and $j = i+1$ in (2.7.8). Thus, in principle $\lambda_j(T)$ can be computed nearly as accurately as the inherent uncertainty ξ_j permits.

We now describe the impact of over/underflow, including division by zero. We denote the pivot d computed when visiting node i by d_i. We first discuss the way division by zero is avoided in EISPACK's `bisect` routine [88], then the superior method in LAPACK's `dstebz` routine [3, 66], and finally how our alternative Algorithm 2.7.2 (Flcnt_IEEE) works (Algorithm 2.7.2 assumes IEEE arithmetic). The difficulty arises because if d_j is tiny or zero, the division T_{ij}^2/d_j can overflow. In addition, T_{ij}^2 can itself over/underflow.

The floating point operations performed while visiting node i are

$$d_i = fl((T_{ii} - x) - (\sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{d_j})). \qquad (2.7.11)$$

To analyze this formula, we will let subscripted ε's and η's denote independent quantities bounded in absolute value by ε (machine precision) and $\bar{\omega}$ (underflow threshold). We will also make standard substitutions like $\prod_{i=1}^{n}(1+\varepsilon_i) \to (1+\tilde{\varepsilon})^n$ where $|\tilde{\varepsilon}| \le \varepsilon$, and $(1+\varepsilon_i)^{\pm 1}\eta_j \to \eta_j$.

2.7.1 Model 1: Barring Overflow, Acyclic Matrix

Barring overflow, (2.7.11) and Assumption 2B(ii) leads to

$$d_i = \{(T_{ii}-x)(1+\varepsilon_a^i) + \eta_{1i} - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{d_j}(1+\tilde{\varepsilon}_{ij})^{C+1} - (2C-1)\eta_{2i}\}(1+\varepsilon_b^i) + \eta_{3i}.$$

or

$$\frac{d_i}{1+\varepsilon_b^i} = (T_{ii}-x)(1+\varepsilon_a^i) - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{d_j}(1+\tilde{\varepsilon}_{ij})^{C+1} + 2C\cdot\eta'_{2i} + \eta'_{3i}.$$

or

$$\frac{d_i}{(1+\varepsilon_c^i)^2} = T_{ii} - x - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{d_j}(1+\hat{\varepsilon}_{ij})^{C+2} + (2C+1)\eta_i,$$

where $(1+\varepsilon_c^i)^2 \equiv (1+\varepsilon_a^i)(1+\varepsilon_b^i)$. Let $\tilde{d}_i = d_i/(1+\varepsilon_c^i)^2$, finally,

$$\tilde{d}_i = T_{ii} + (2C+1)\eta_i - x - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{\tilde{d}_j}(1+\varepsilon_{ij})^{C+4}. \tag{2.7.12}$$

Remark 2.7.1 Under Model 2, IEEE arithmetic with gradual underflow, the underflow error $(2C+1)\eta_i$ of the above equation can be replaced by $C\eta_i$ because addition and subtraction never underflow.

If there is no underflow during the computations of d_i either, then (2.7.12) simplifies to:

$$\tilde{d}_i = T_{ii} - x - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{\tilde{d}_j}(1+\varepsilon_{ij})^{C+4}.$$

This proves (2.7.7), since the $\tilde{d}_i$ are the exact pivots corresponding to T' where T' satisfies (2.7.7) and $\operatorname{sign}(\tilde{d}_i) = \operatorname{sign}(d_i)$.

Remark 2.7.2 We need to bar overflow *in principle* for symmetric acyclic matrix with IEEE arithmetic, because if in (2.7.11), there are two children j_1 and j_2 of i such that $T_{ij_1}^2/d_{j_1}$ overflows to ∞ and $T_{ij_2}^2/d_{j_2}$ overflows to $-\infty$; then d_i will be NaN, not even well-defined.

2.7.2 Models 2 and 3: EISPACK's FlCnt_bisect, Tridiagonal Matrix

EISPACK's FlCnt_bisect can overflow for symmetric tridiagonal or acyclic matrices with Model 1 arithmetic, and return NaN's for symmetric acyclic matrices and IEEE arithmetic since it makes no provision against overflow (see Remark 2.7.2). In this section, we assume T is a symmetric tridiagonal matrix, whose graph is just a chain, i.e. $C = 1$. Therefore, to describe the error analysis for FlCnt_bisect, we need the following assumptions:

Assumption 2B(ii): $\bar{M} \equiv \max_{i,j} |T_{ij}| \leq \sqrt{\Omega}$, and one of

Assumption 1A: Model 2. Full IEEE arithmetic with ∞ and NaN arithmetic, and with gradual underflow, or

Assumption 1A: Model 3. Full IEEE arithmetic with ∞ and NaN arithmetic, but with underflow flushing to zero.

Algorithm 2.7.1 EISPACK **FlCnt_bisect**. FloatingCount(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.**

FloatingCount $= 0$;
$d_0 = 1$;
for $i = 1$ *to* n
    if $(d_{i-1} = 0)$ then
        $v = |b_{i-1}|/\varepsilon$
    else
        $v = b_{i-1}^2/d_{i-1}$
    endif
    $d_i = a_i - x - v$

if $d_i < 0$ **then**
  $\text{FloatingCount} = \text{FloatingCount} + 1$
endif
endfor

Under Models 2 and 3, our error expression (2.7.12) simplifies to

$$\tilde{d}_i = a_i + 3\eta_i - x - \frac{b_{i-1}^2(1+\varepsilon_{ij})^5}{\tilde{d}_{i-1}}.$$

where $a_i = T_{ii}$ and $b_{i-1} = T_{i-1,i}$.

However, FlCnt_bisect's provision against division by zero can drastically increase the backward error bound (2.7.7). When $d_j = 0$ for some j in (2.7.11), it is easy to see that what `bisect` does is equivalent to perturbing a_j by $\varepsilon|b_j|$. This backward error is clearly small in norm, i.e. at most $\varepsilon\|T\|_2$, and so by Weyl's Theorem, can perturb computed eigenvalue by no more than $\varepsilon\|T\|_2$. If one is satisfied with absolute accuracy, this is sufficient. However, it can clearly destroy any componentwise relative accuracy, because $\varepsilon|b_j|$ may be much larger than $|a_j|$.

Furthermore, suppose there is some k such that d_k overflows, i.e. $|d_k| \geq \Omega$. Since $\bar{M} \leq \sqrt{\Omega}$, it must be b_{k-1}^2/d_{k-1} that overflows. So $\tilde{d}_k$ is $-\text{sign}(b_{k-1}^2/d_{k-1}) \cdot \infty$ which has the same sign as the exact pivot corresponding to T'. But this will contribute an extra uncertainty to a_{k+1} of at most $\bar{M}^2/\Omega$, since $|b_k^2/d_k| \leq \bar{M}^2/\Omega$.

Therefore we get the following backward error for FlCnt_bisect:

$$|T_{ij} - T'_{ij}| \leq f(2.5, \varepsilon)|T_{ij}| \;\text{ if }\; i \neq j.$$

and

$$|T_{ii} - T'_{ii}| \leq \varepsilon\|T\|_2 + \frac{\bar{M}^2}{\Omega} + \begin{cases} \varepsilon\omega & \text{Model 2} \\ 3\omega & \text{Model 3} \end{cases}.$$

2.7.3 Models 2 and 3: FlCnt_IEEE, Tridiagonal Matrix

The following code can work only for *unreduced* symmetric tridiagonal matrices under Models 2 and 3 for the same reason as FlCnt_bisect: otherwise we could get $T_{ij_1}^2/d_{j_1} + T_{ij_2}^2/d_{j_2} = \infty - \infty = \text{NaN}$. So in this section, we again assume T is a symmetric tridiagonal matrix. By using IEEE arithmetic, we can eliminate all tests in the inner loop, and so make it faster on many architectures [34]. To describe the error analysis, we again make

Assumptions 1A(Model 2 or Model 3) and 2B(ii), as in section 2.7.2, and Assumption 2B(i), which is $\bar{B} \equiv \min_{i \neq j} T_{ij}^2 \geq \omega$.

The function `SignBit` is defined as in IEEE floating point arithmetic, i.e., `SignBit`(x) is 0 if $x > 0$ or $x = +0$, and 1 if $x < 0$ or $x = -0$.

Algorithm 2.7.2 FlCnt_IEEE. FloatingCount(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.**

FloatingCount $= 0$;
$d_0 = 1$;
for $i = 1$ *to* n
    / note that there is no provision against overflow and division by zero */*
    $d_i = (a_i - x) - b_{i-1}^2 / d_{i-1}$
    FloatingCount $=$ FloatingCount $+$ `SignBit`(d_i)
endfor

By Assumption 2B(i), b_i^2 never underflows. Therefore when some d_i underflows, we do not have the headache of dealing with 0/0 which is NaN.

Algorithm 2.7.2 is quite similar to FlCnt_bisect except division by zero is permitted to occur, and the `SignBit`(± 0) function ($= 0$ or 1) is used to count eigenvalues [30]. More precisely, if $d_i = +0$, d_{i+1} would be $-\infty$, so after two steps, Count will increase by 1. On the other hand, if $d_i = -0$, d_{i+1} would be $+\infty$, hence Count also increases by 1 after two steps. Therefore, we can simply change any $d_i = -0$ to $d_i = +0$, and $d_{i+1} = +\infty$ to $d_{i+1} = -\infty$, to eliminate -0 from the analysis. Then using an analysis analogous to the last section, if we use Model 2(gradual underflow), T' differs from T by

$$|T_{ij} - T'_{ij}| \leq f(2.5, \varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } |T_{ii} - T'_{ii}| \leq \frac{\bar{M}^2}{\Omega} + \varepsilon\omega.$$

Using Model 3(flush to zero), we have the slightly weaker results that

$$|T_{ij} - T'_{ij}| \leq f(2.5, \varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } |T_{ii} - T'_{ii}| \leq \frac{\bar{M}^2}{\Omega} + 3\omega.$$

Since $\bar{M} \leq \sqrt{\Omega}$, so

$$\frac{\bar{M}^2/\Omega}{\bar{M}} \leq \frac{1}{\sqrt{\Omega}} \ll \varepsilon.$$

which tells us that $\bar{M} \leq \sqrt{\Omega}$ is a good scaling choice.

2.7.4 Models 1, 2 and 3: LAPACK's FlCnt_stebz routine, Acyclic Matrix

In contrast to EISPACK's FlCnt_bisect and FlCnt_IEEE, LAPACK's FlCnt_stebz can work in principle for general symmetric acyclic matrices under all three models (although its current implementation only works for tridiagonal matrices). So in this section, T is a symmetric acyclic matrix. Let $B = \max_{i \neq j}(1, T_{ij}^2) \leq \Omega$, and $\hat{p} = 2C \cdot B/\Omega$ ($\hat{p}$ is called *pivmin* in `dstebz`). In this section, we need the Assumptions 1A (Model 1, 2 or 3) and 2B(ii). Because of the Gerschgorin Disk Theorem, we can restrict our attention to those shifts x such that $|x| \leq (n+1)\sqrt{\Omega}$.

Algorithm 2.7.3 LAPACK **Flcnt_stebz.** FloatingCount(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

```
call TreeCount(1, x, d_1, s_1)
FloatingCount = s_1

procedure TreeCount(i, x, d_i, s_i)   /* i and x are inputs, d_i and s_i are outputs */
    d_i = fl(T_ii - x)
    s_i = 0
    for all children j of i do
        call TreeCount(j, x, d_j, s_j)
        sum = sum + T_ij^2 / d_j
        s_i = s_i + s_j
    endfor
    d_i = (T_ii - x) - sum
    if (|d_i| <= p̂) then
        d_i = -p̂
    endif
    if d_i < 0 then
        s_i = s_i + 1
    endif
end TreeCount
```

It is clear that $|d_i| \geq \hat{p}$ for each node i, so

$$|T_{ii}| + |x| + \sum_{\substack{\text{all children} \\ j \text{ of } i}} |\frac{T_{ij}^2}{d_j}| \leq (n+2)\sqrt{\Omega} + C \cdot \frac{B}{\hat{p}} \leq \frac{\Omega}{2} + C\frac{B}{2C \cdot B/\Omega} = \Omega.$$

This tells us that FlCnt_stebz never overflows and it works under all three models. For all these models, the assignment $d_i = -\hat{p}$ when $|d_i|$ is small can contribute an extra uncertainty to T_{ii} of no more than $2 \cdot \hat{p}$. Thus we have the following backward error:

$$|T_{ij} - T'_{ij}| \leq f(C/2+2, \varepsilon)|T_{ij}| \ \text{ if } \ i \neq j.$$

and

$$|T_{ii} - T'_{ii}| \leq 2 \cdot \hat{p} + \begin{cases} (2C+1)\bar{\omega} & \text{Model 1} \\ C\varepsilon\omega & \text{Model 2} \\ (2C+1)\omega & \text{Model 3} \end{cases}.$$

The driver routine which calls `dstebz` scales the input matrix (which is reduced to tridiagonal T before calling `dstebz`) such that $B = O(\omega^{1/2}\Omega)$, therefore, $\hat{p} = 2C \cdot B/\Omega = O(\sqrt{\omega})$.

2.7.5 Models 1,2 and 3: FlCnt_Best_Scaling, Acyclic Matrix

Following Kahan[66], let $\xi = \omega^{1/4}\Omega^{-1/2}$ and $M = \xi \cdot \Omega = \omega^{1/4}\Omega^{1/2}$. The following code assumes that the initial data has been scaled so that

$$\bar{M} \leq \frac{M}{\sqrt{2C}} \ \text{ and } \ \bar{M} \approx \frac{M}{\sqrt{2C}}.$$

This code can be used to compute the eigenvalues of general symmetric acyclic matrix, so in this section, T is a symmetric acyclic matrix. To describe the error analysis, we only need Assumption 1A. Again because of the Gerschgorin Disk Theorem, the shifts are restricted to those x such that $|x| \leq (n+1)M$.

Algorithm 2.7.4 FlCnt_Best_Scaling. FloatingCount(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

call $\textsc{TreeCount}(1, x, d_1, s_1)$
FloatingCount $= s_1$

```
procedure TreeCount(i, x, d_i, s_i)   /* i and x are inputs, d_i and s_i are outputs */
    d_i = fl(T_ii - x)
    s_i = 0
    for all children j of i do
        call TreeCount(j, x, d_j, s_j)
        sum = sum + T_ij^2 / d_j
        s_i = s_i + s_j
    endfor
    d_i = (T_ii - x) - sum
    if (|d_i| <= sqrt(ω)) then
        d_i = -sqrt(ω)
    endif
    if d_i < 0 then
        s_i = s_i + 1
    endif
end TreeCount
```

Similar to FlCnt_stebz, $|d_i| \geq \sqrt{\omega}$ for any node i, so

$$|T_{ii}| + |x| + \sum_{\substack{\text{all children} \\ j \text{ of } i}} |\frac{T_{ij}^2}{d_j}| \leq (n+1)M + \frac{M}{\sqrt{2C}} + C \cdot \frac{M^2/2C}{\omega^{1/2}} \leq \frac{\Omega}{2} + C \cdot \frac{\omega^{1/2}\Omega/2C}{\omega^{1/2}} = \Omega$$

which tells us overflow **never** happens and the code can work fine under all the models we mentioned. For all the models, The backward error bound becomes,

$$|T_{ij} - T'_{ij}| \leq f(C/2 + 2, \varepsilon)|T_{ij}| \text{ if } i \neq j.$$

and

$$|T_{ii} - T'_{ii}| \leq 2\sqrt{\omega} + \begin{cases} (2C+1)\bar{\omega} & \text{Model 1} \\ C\varepsilon\omega & \text{Model 2} \\ (2C+1)\omega & \text{Model 3} \end{cases}.$$

2.7.6 Error Bounds For Eigenvalues

We need the following lemma to give error bounds for the computed eigenvalues.

Table 2.2: Backward Error Bounds for Symmetric Tridiagonal Matrices

Algorithms	Model 1		Model 2		Model 3	
	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β
FlCnt_bisect	—	—	$f(2.5,\varepsilon)$	$\varepsilon\|T\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$\varepsilon\|T\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$
FlCnt_stebz	$f(2.5,\varepsilon)$	$2\hat{p}+3\bar{\omega}$	$f(2.5,\varepsilon)$	$2\hat{p}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$2\hat{p}+3\omega$
Flcnt_Best_Scal	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+3\bar{\omega}$	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+3\omega$
FlCnt_IEEE	—	—	$f(2.5,\varepsilon)$	$\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$\frac{\bar{M}^2}{\Omega}+3\omega$

Lemma 2.7.1 *Assume T is an acyclic matrix and* FloatingCount(x,T) = Count$(x,\ T')$, *where T' differs from T only slightly:*

$$|T_{ij}-T'_{ij}| \leq \alpha(\varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } |T_{ii}-T'_{ii}| \leq \beta.$$

where $\alpha(\varepsilon) \geq 0$ is a function of ε and $\beta \geq 0$. Then this backward error can change the eigenvalues λ_k by at most ξ_k where

$$\xi_k \leq 2\alpha(\varepsilon) \parallel T \parallel_2 + \beta. \tag{2.7.13}$$

Proof. By Weyl's Theorem [80],

$$\xi_k \leq \|T-T'\|_2 \leq \||T-T'|\|_2 \leq \|\alpha(\varepsilon)|T-\Lambda|+\beta I\|_2 \leq \alpha(\varepsilon)\||T-\Lambda|\|_2+\beta.$$

and

$$\||T-\Lambda|\|_2 = \|T-\Lambda\|_2 \leq \|T\|_2+\|\Lambda\|_2 \leq 2\|T\|_2.$$

where $\Lambda = \mathrm{diag}(d_i)$ which is the diagonal part of T. Therefore,

$$\xi_k \leq 2\alpha(\varepsilon)\|T\|_2+\beta.$$

∎

In Tables 2.2 through 2.5, we present the backward errors $\alpha(\varepsilon)$ and β, and the corresponding error bounds ξ_k for the various algorithms under different models of arithmetic.

2.7.7 Correctness of the Gerschgorin Bound

In section 2.1, we mentioned that to compute all the eigenvalues of an $n \times n$ symmetric tridiagonal matrix T, we need to find an interval $[left, right)$, such that

$$\text{FloatingCount}(left) = 0 \quad \text{and FloatingCount}(right) = n.$$

Table 2.3: Error Bounds ξ_k of Eigenvalues for Symmetric Tridiagonal Matrices

Algorithms	Model 1	Model 2	Model 3
FlCnt_bisect	—	$[2f(2.5,\varepsilon)+\varepsilon]\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$[2f(2.5,\varepsilon)+\varepsilon]\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$
FlCnt_stebz	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\hat{p}+3\bar{\omega}$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\hat{p}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\hat{p}+3\omega$
Flcnt_Best_Scal	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+3\bar{\omega}$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+3\omega$
FlCnt_IEEE	—	$2f(2.5,\varepsilon)\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$

Table 2.4: Backward Error Bounds for Symmetric Acyclic Matrices

Algorithms	Model 1		Model 2		Model 3	
	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β
FlCnt_bisect	—	—	—	—	—	—
FlCnt_stebz	$f(C/2+2,\varepsilon)$	$2\hat{p}+(2C+1)\bar{\omega}$	$f(C/2+2,\varepsilon)$	$2\hat{p}+C\varepsilon\omega$	$f(C/2+2,\varepsilon)$	$2\hat{p}+(2C+1)\omega$
Flcnt_Best_S	$f(C/2+2,\varepsilon)$	$2\sqrt{\omega}+(2C+1)\bar{\omega}$	$f(C/2+2,\varepsilon)$	$2\sqrt{\omega}+C\varepsilon\omega$	$f(C/2+2,\varepsilon)$	$2\sqrt{\omega}+(2C+1)\omega$
FlCnt_IEEE	—	—	—	—	—	—

Table 2.5: Error Bounds ξ_k of Eigenvalues for Symmetric Acyclic Matrices, $g(\varepsilon) = f(C/2+2,\varepsilon)$

Algorithms	Model 1	Model 2	Model 3
FlCnt_bisect	—	—	
FlCnt_stebz	$2g(\varepsilon)\|\|T\|\|_2+2\hat{p}+(2C+1)\bar{\omega}$	$2g(\varepsilon)\|\|T\|\|_2+2\hat{p}+C\varepsilon\omega$	$2g(\varepsilon)\|\|T\|\|_2+2\hat{p}+(2C+1)\omega$
Flcnt_Best_Scal	$2g(\varepsilon)\|\|T\|\|_2+2\sqrt{w}+(2C+1)\bar{\omega}$	$2g(\varepsilon)\|\|T\|\|_2+2\sqrt{w}+C\varepsilon\omega$	$2g(\varepsilon)\|\|T\|\|_2+2\sqrt{w}+(2C+1)\omega$
FlCnt_IEEE	—	—	

function COMPUTE_GERSCHGORIN(n,T) /* returns the Gerschgorin Interval [gl,gu] */

$gl = \min_{i=1}^{n}(T_{ii} - \sum_{j\neq i}|T_{ij}|)$; /* Gerschgorin left bound */

$gu = \max_{i=1}^{n}(T_{ii} + \sum_{j\neq i}|T_{ij}|)$; /* Gerschgorin right bound */

$bnorm = \max(|gl|, |gu|)$;

$gl = gl - bnorm \cdot 2n\varepsilon - \xi_0$; $gu = gu + bnorm \cdot 2n\varepsilon + \xi_n$; /* see Table 2.6 */

$gu = \max(gl, gu)$;

return(gl,gu);

end function

Figure 2.1: COMPUTE_GERSCHGORIN computes the Gerschgorin interval for T

In this section, we will prove the correctness of the Gerschgorin interval returned by the function COMPUTE_GERSCHGORIN [30] (see figure 2.1). We will need assumptions 1A and the correctness property of FloatingCount(x).

In exact arithmetic,

$$glexact = \min_i(T_{ii} - \sum_{j\neq i}|T_{ij}|); \;\; guexact = \max_i(T_{ii} + \sum_{j\neq i}|T_{ij}|).$$

So, $bnorm = \max(|glexact|, |guexact|) = \|T\|_\infty$. Notice that

$$fl((T_{ii} - \sum_{j\neq i}|T_{ij}|)) = (T_{ii}(1+\delta_i)^{k_i} - \sum_{j\neq i}|T_{ij}|(1+\delta_j)^{k_j}).$$

Therefore,

$$|fl(glexact) - glexact| \leq f(C,\varepsilon)\|T\|_\infty \leq 2n\varepsilon\|T\|_\infty = 2n\varepsilon \cdot bnorm.$$

Similarly, $|fl(guexact) - guexact| \leq 2n\varepsilon \cdot bnorm$. With correctness property of FloatingCount(x), this proves that if we let

$$gl = fl(glexact) - 2n\varepsilon \cdot bnorm - \xi_0; \;\; gu = fl(guexact) + 2n\varepsilon \cdot bnorm + \xi_n.$$

then we can claim

$$\text{FloatingCount}(gl) = 0; \;\; \text{FloatingCount}(gu) = n.$$

Table 2.6: Upper Bounds for ξ_k for Different Algorithms under Different Models

Algorithms	Matrix	Additional Assumptions	Bounds of ξ_k
FlCnt_bisect	Tridiagonal	Assumption 2C(i)	$11\varepsilon \cdot bnorm$
FlCnt_stebz	Acyclic	Assumptions 2C(i), 2C(ii)	$(8n+6)\varepsilon \cdot bnorm$
Flcnt_Best_Scaling	Acyclic	—	$(4n+8)\varepsilon \cdot bnorm$
FlCnt_IEEE	Tridiagonal	Assumption 2C(i)	$10\varepsilon \cdot bnorm$

For the algorithms we mentioned in the previous sections, we can obtain the upper bounds for ξ_k under certain additional appropriate assumptions, which enable us to give more specific and explicit Gerschgorin bounds computed by the routine COMPUTE_GERSCHGORIN (see Table 2.6). For instance, the error bound of FlCnt_bisect for symmetric tridiagonal matrices is at most $[2f(2.5,\varepsilon)+\varepsilon]\|T\|_2+\bar{M}^2/\Omega+3\omega$, with Assumption 2C(i): $\bar{M}\geq\omega/\varepsilon$, we have

$$\begin{aligned}&[2f(2.5,\varepsilon)+\varepsilon]\|T\|_2+\bar{M}^2/\Omega+3\omega \leq (2\cdot 2.5\cdot 1.06\varepsilon+\varepsilon)\|T\|_2+\frac{\bar{M}}{\Omega}\bar{M}+3\varepsilon\bar{M}\\ \leq\; & 7\varepsilon\cdot bnorm+\varepsilon\cdot bnorm+3\varepsilon\cdot bnorm=11\varepsilon\cdot bnorm.\end{aligned}$$

According to Table 2.6, if we let

$$gl=fl(glexact)-(10n+6)\varepsilon\cdot bnorm;\;\; gu=fl(guexact)+(10n+6)\varepsilon\cdot bnorm. \quad (2.7.14)$$

Then we have

$$\text{FloatingCount}(gl)=0;\;\; \text{FloatingCount}(gu)=n$$

in all situations, which shows the Gerschgorin Bound (2.7.14) is correct for EISPACK's FlCnt_bisect, LAPACK's FlCnt_stebz, FlCnt_IEEE and Flcnt_Best_Scal.

2.8 Summary

To end this chapter, we present two tables to summarize all the different implementations of FloatingCount(x) we introduced and all the results we concluded. Tables 2.7 describes the algorithms, and Tables 2.8 describe their properties. For each implementation of FloatingCount, Table 2.8 lists which parts of Assumptions 1–2 are needed for correctness property of FloatingCount, and possibly monotonicity, to hold.

Table 2.7: Different implementations of FloatingCount

Algorithms	Description	Where
FlCnt_bisect	algorithm used in EISPACK's `bisect` routine; most floating point exceptions avoided by tests and branches	See section 2.7.2 and [88]
FlCnt_IEEE	IEEE standard floating point arithmetic used to accommodate possible exceptions; tridiagonals only	See section 2.7.3 and [8, 66]
FlCnt_stebz	algorithm used in LAPACK's `dstebz` routine; floating point exceptions avoided by tests and branches	See section 2.7.4 and [3]
Flcnt_Best_Scaling	like FlCnt_stebz, but prescales for optimal error bounds	See section 2.7.5 and [8, 66]

Table 2.8: Results of Roundoff Error Analysis and Monotonicity

Assumptions about Arithmetic and Input Matrix	Results	Proofs
T is symmetric tridiagonal $\wedge$ (1A(Model 2) $\vee$ 1A(Model 3)) $\wedge$ 1B $\wedge$ 2A $\wedge$ 2B(ii)	For FlCnt_bisect, Correctness Property holds but FloatingCount(x) can be nonmonotonic	See section 2.7.2
T is symmetric tridiagonal $\wedge$ (1A(Model 2) $\vee$ 1A(Model 3)) $\wedge$ 1B $\wedge$2A $\wedge$ 2B	For FlCnt_IEEE, Correctness Property holds and FloatingCount(x) is monotonic	See section 2.6 and section 2.7.3
T is symmetric acyclic $\wedge$ 1A $\wedge$ 1B $\wedge$ 1C $\wedge$ 2A $\wedge$ 2B(ii)	For FlCnt_stebz, Correctness Property holds and FloatingCount(x) is monotonic	See section 2.6 and section 2.7.4
T is symmetric acyclic $\wedge$ 1A $\wedge$ 1B $\wedge$ 1C $\wedge$ 2A	For Flcnt_Best_Scaling, Correctness Property holds and FloatingCount(x) is monotonic	See section 2.6 and section 2.7.5

Chapter 3

The Instability and Nonmonotonicity of FloatingCount Implemented Using Parallel Prefix

3.1 Introduction

THE *Parallel Prefix* operation is very useful to parallelize many numerical linear algebra algorithms [26, 28, 32]. The bisection algorithm is one of its many applications. In this chapter, we will present numerical examples to show that when FloatingCount(x) is implemented using parallel prefix, it can be nonmonotonic and very unstable.

3.1.1 Another Way to Count Eigenvalues Less Than x

Let T_k be the leading $k \times k$ principal submatrix (sometimes called *leading principal minor*) of the symmetric tridiagonal matrix T in (2.2.1) and define the polynomials $p_k(x) = \det(T_k - xI)$ where I is an $k \times k$ identity matrix, for $k = 1 : n$. Since T is tridiagonal, it can be easily shown that the sequence $p_k(x)$ satisfies the following three term recurrence [43]:

$$p_k(x) = (a_k - x)p_{k-1}(x) - b_{k-1}^2 p_{k-2}(x) \tag{3.1.1}$$

where we let $p_0(x) = 1$.

We state the following classical result [43, 97]:

Theorem 3.1.1 (Sturm Sequence Property) *If the symmetric tridiagonal matrix T in (2.2.1) is unreduced, then the eigenvalues of T_{k-1} strictly separate the eigenvalues of T_k:*

$$\lambda_1(T_k) < \lambda_1(T_{k-1}) < \lambda_2(T_k) < \ldots < \lambda_{k-1}(T_k) < \lambda_{k-1}(T_{k-1}) < \lambda_k(T_k). \tag{3.1.2}$$

Moreover, if $s(x)$ denotes the number of sign changes in the sequence (which we call a **Sturm sequence***)*

$$\{p_0(x), p_1(x), \ldots, p_n(x)\},$$

then $s(x)$ equals the number of T's eigenvalues that are less than x. Here the polynomials $p_k(x)$ are defined by (3.1.1) and we have the convention that $p_k(x)$ has the opposite sign of $p_{k-1}(x)$ if $p_k(x) = 0$.

The function Count(x) can be computed in a different way from Algorithm 2.2.2 as follows:

Algorithm 3.1.1 Count(x) returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.

```
Count = 0;
p_0 = 1;
p_{-1} = 0;
b_0 = 0;
for i = 1 to n
    p_i = (a_i - x) p_{i-1} - b_{i-1}^2 p_{i-2}
    if (p_i p_{i-1} < 0 or (p_{i-1} ≠ 0 and p_i = 0)) then
        Count = Count + 1
    end if
end for
```

(If we wish to emphasize that T is the argument, we will write Count(x, T) *instead.)*

Remark 3.1.1 There is a simple relationship between Algorithm 2.2.2 and Algorithm 3.1.1:

$$p_k = d_1 d_2 \ldots d_k.$$

Proc #	0	1	2	3	4	5	6	7
Step 0	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
Step 1		$x_{0:1}$		$x_{2:3}$		$x_{4:5}$		$x_{6:7}$
Step 2				$x_{0:3}$				$x_{4:7}$
Step 3								$x_{0:7}$
Step 4						$x_{0:5}$		
Step 5			$x_{0:2}$		$x_{0:4}$		$x_{0:6}$	

Figure 3.1: Parallel Prefix on 8 data items

This relationship can be easily verified by using equation (2.2.2) and noticing that

$$\det(T_k) = \det(L_k D_k L_k^T) = \det(L_k)\det(D_k)\det(L_k^T)$$

and

$$p_k = \det(T_k), \quad \det(L_k) = 1, \quad \det(D_k) = d_1 d_2 \dots d_k,$$

where T_k, L_k and D_k are the leading principal submatrices of T, L and D respectively.

3.1.2 Parallel Prefix

The parallel prefix operation [26, 28, 32, 94], also called *scan*, is defined as follows:

Definition 3.1.1 Parallel Prefix Operation(Scan) Let $x_0, x_1, \dots, x_n$ be data items, and $\otimes$ any associative operation. Then the *scan* of these n data items yields another n data items defined by

$$y_0 = x_0, y_1 = x_0 \otimes x_1, \dots, y_i = x_0 \otimes x_1 \otimes \dots \otimes x_i, \dots, y_n = x_0 \otimes x_1 \otimes \dots \otimes x_n. \qquad (3.1.3)$$

We also say y_i is the *reduction* of x_0 through x_i.

The attraction of this operation, other than its usefulness, is its ease of implementation using a simple tree of processors [28, 32]. We illustrate in figure 3.1.2 for $n = 8$; in the figure we denote $x_i \cdots x_j$ by $x_{i:j}$, i-th column contains all the data held by i-th processor, and only the data that changes are indicated.

Parallel prefix can be used to solve linear recurrence relations For example, to evaluate $z_{i+1} = a_i z_i + b_i, i \geq 0, z_0 = 0$, we do the following operations [32]:

Compute $p_i = a_0 \cdots a_i$ using parallel prefix multiplication

Compute $\beta_i = b_i / p_i$ in parallel

Compute $s_i = \beta_0 + \cdots + \beta_{i-1}$ using parallel prefix addition

Compute $z_i = s_i \cdot p_{i-1}$ in parallel

This approach extends to n term linear recurrences $z_{i+1} = \sum_{j=0}^{n-2} a_{i,j} z_{i-j} + b_i$, but the associative operation becomes $n-1$ by $n-1$ matrix multiplication.

Similarly, we can use parallel prefix to evaluate certain rational recurrences $z_{i+1} = (a_i z_i + b_i)/(c_i z_i + d_i)$ by writing $z_i = u_i / v_i$ and reducing to linear recurrence for u_i and v_i [32]:

$$\begin{bmatrix} u_{i+1} \\ v_{i+1} \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \cdot \begin{bmatrix} u_i \\ v_i \end{bmatrix}. \tag{3.1.4}$$

We may ask more generally about evaluating the scalar rational recurrence $z_{i+1} = f_i(z_i)$ in parallel. Let deg be the maximum of the degrees of the numerators and denominators of the rational functions f_i. Kung [69] has shown that z_i can be evaluated faster than linear time (i.e. z_i can be evaluated in $o(i)$ steps) if and only if $deg \leq 1$; in this case the problem reduces to 2×2 matrix multiplication parallel prefix in (3.1.4). Basic linear algebra operations which can be solved in this way include tridiagonal Gauss elimination(three term recurrence), solving bidiagonal linear systems(two term recurrence), Sturm sequence evaluation for the symmetric tridiagonal eigenproblem(three term recurrence), and the bidiagonal dqds algorithm for singular values(three term recurrence)[38].

The numerical stability of these algorithms is not completely understood. For some application, it is easy to see that the error bounds are rather worse than the those of the sequential implementation [85].

For the case of Sturm sequence evaluation for the symmetric tridiagonal eigenproblem, instead of computing p_i as $p_i = (a_i - x)p_{i-1} - b_{i-1}^2 p_{i-2}$, we can do this as follows:

$$[p_i, \;\; p_{i-1}] = [p_{i-1}, \;\; p_{i-2}] \cdot \begin{bmatrix} a_i - x & 1 \\ -b_{i-1}^2 & 0 \end{bmatrix} = [p_{i-1}, \;\; p_{i-2}] \cdot M_i.$$

Therefore,

$$[p_i, \;\; p_{i-1}] = [p_0, \;\; p_{-1}] M_1 M_2 \cdots M_i,$$

where $p_0 = 1, p_{-1} = 0$ and $b_0 = 0$. So we can compute $M_1 M_2 \cdots M_i$, for $i = 1, \ldots, n$, by parallel prefix in time $(O(\log_2 n))$ on n processors. The $(1,1)$ element of the product

$M_1 M_2 \cdots M_i$ will give us the p_i. From now on, we will denote the product $M_i M_{i-1} \cdots M_j$ by $M_{i:j}$. Thus we have the following algorithm, which we call **Count_Prefix**, and computes Count(x) using parallel prefix:

Algorithm 3.1.2 Count_Prefix. Count(x) returns the number of eigenvalues of a real symmetric tridiagonal matrix T that are less than x.

Count $= 0$;
$p_0 = 1$;
$p_{-1} = 0$;
$b_0 = 0$;
for $i = 1$ *to* n
    $M_i = \begin{bmatrix} a_i - x & 1 \\ -b_{i-1}^2 & 0 \end{bmatrix}$
end for
Compute $M_{1:i}$ *for* $i = 1, \ldots, n$ *using parallel prefix.*
for $i = 1$ *to* n
    $p_i = M_{1:i}(1,1)$
    if $(p_i p_{i-1} < 0 \text{ or } (p_{i-1} \neq 0 \text{ and } p_i = 0))$ then
        Count $=$ Count $+ 1$
    end if
end for

Several work has been done concerning the numerical stability of the above algorithm [94, 74]. In some cases, the reasonably accurate results can be obtained in practice [94]. However, Mathias [74] has shown that even for positive definite matrices, the relative error in the computed Sturm sequence can be as large as $\varepsilon\kappa^3$, where ε is machine precision and κ is the condition number for the problem of computing the eigenvalues of T. Therefore, if we can not find a cheap way to detect the instability or to correct the wrong results, the parallel prefix algorithm is not a reliable algorithm to use in practice.

In this chapter, we discuss the backward error analysis for general matrices, and present some numerical experiment results to show the instability and nonmonotonicity of FloatingCount(x) implemented using parallel prefix.

3.2 An Example of Instability of Count_Prefix

In this section, we show that when we use parallel prefix algorithm (Algorithm 3.1.2) to compute Count(x) instead of using a serial algorithm like Algorithm 3.1.1 or Algorithm 2.2.2, the computed result can be very inaccurate.

We implement both Algorithm 3.1.2 and Algorithm 3.1.1 in MATLAB[1], and use them to compute the counts for the following matrix:

$$T_{\text{Wilkinson}} = \begin{pmatrix} 32 & 1 & & & & & & \\ 1 & 31 & 1 & & & & & \\ & \ddots & \ddots & \ddots & & & & \\ & & 1 & 1 & 1 & & & \\ & & & 1 & 1 & 1 & & \\ & & & & \ddots & \ddots & \ddots & \\ & & & & & 1 & 31 & 1 \\ & & & & & & 1 & 32 \end{pmatrix}_{64\times 64} . \tag{3.2.5}$$

(Matrix $T_{\text{Wilkinson}}$ is very similar to the well known Wilkinson W_{21}^{+} matrix [97], except that there are no zeros on the diagonal.)

When we computed the counts for $T_{\text{Wilkinson}}$ at a few shifts $x = 14$, 15, 16, 17, 18, 19, the results are very inaccurate. For example, the computed Count(14) is 14 while the true count should be 27. These shifts are very close to the double eigenvalues of $T_{\text{Wilkinson}}$(see table 3.2).

To explain why this loss of accuracy happens, we consider the shift $x = 14$ only, the other shifts can be explained in a similar way. We plotted the computed function Count(x) in a tiny interval centered at 14, see figure 3.2. We can observe that the curve of the computed Count(x) has a very sharp spike at 14, which is clearly incorrect. We can also look at table 3.2 to see how different the results computed by parallel prefix algorithm are from those computed by serial algorithm(for parallel prefix, $p_i = 0$ for $i \geq 32$ while serial algorithm gets completely different results).

As we mentioned earlier in section 3.1.2, when we use parallel prefix to compute the Sturm sequence, we first generate a sequence of 2×2 matrices, which are denoted by $M_1, M_2, \ldots, M_{64}$, we also use $M_{i:j}$ to denote the product $M_i M_{i+1} \ldots M_j$.

[1]IEEE double precision floating point arithmetic, with machine precision $\varepsilon = 2^{-53} \approx 1.1 \times 10^{-16}$

order	eigenvalues	order	eigenvalues	order	eigenvalues
1	-0.4641321726904621	23	11.99999999999999	45	23.00000000000062
2	0.7126628425131144	24	12.00000000000000	46	23.00000000000063
3	1.578164406734629	25	12.99999999999999	47	24.00000000005449
4	2.154730092254134	26	13.00000000000001	48	24.00000000005450
5	2.900771751580554	27	13.99999999999999	49	25.00000000380813
6	3.113723558709666	28	14.00000000000001	50	25.00000000380815
7	3.986274955522141	29	15.00000000000000	51	26.00000020507043
8	4.017639311060099	30	15.00000000000000	52	26.00000020507044
9	4.998968620932772	31	16.00000000000000	53	27.00000815867295
10	5.001192469653752	32	16.00000000000001	54	27.00000815867296
11	5.999953844648227	33	17.00000000000000	55	28.00022568018515
12	6.000050238625582	34	17.00000000000000	56	28.00022568018517
13	6.999998623277990	35	18.00000000000000	57	29.00395200266536
14	7.000001455954138	36	18.00000000000001	58	29.00395200266538
15	7.999999970478815	37	19.00000000000000	59	30.03894111930644
16	8.000000030730062	38	19.00000000000000	60	30.03894111930648
17	8.999999999521551	39	20.00000000000000	61	31.21067864733305
18	9.000000000493216	40	20.00000000000000	62	31.21067864733305
19	9.999999999993923	41	21.00000000000000	63	32.74619418290332
20	10.00000000000622	42	21.00000000000000	64	32.74619418290336
21	10.99999999999993	43	22.00000000000001		
22	11.00000000000006	44	22.00000000000001		

Table 3.1: Eigenvalues of 64×64 $T_{\mathrm{Wilkinson}}$

	serial	prefix		serial	prefix
p_1	1.8000e+01	1.8000e+01	p_{33}	6.7763e+04	0
p_2	3.0500e+02	3.0500e+02	p_{34}	-8.8578e+05	0
p_3	4.8620e+03	4.8620e+03	p_{35}	9.6758e+06	0
p_4	7.2625e+04	7.2625e+04	p_{36}	-9.5872e+07	0
p_5	1.0119e+06	1.0119e+06	p_{37}	8.5318e+08	0
p_6	1.3082e+07	1.3082e+07	p_{38}	-6.7295e+09	0
p_7	1.5597e+08	1.5597e+08	p_{39}	4.6254e+10	0
p_8	1.7026e+09	1.7026e+09	p_{40}	-2.7079e+11	0
p_9	1.6870e+10	1.6870e+10	p_{41}	1.3077e+12	0
p_{10}	1.5013e+11	1.5013e+11	p_{42}	-4.9600e+12	0
p_{11}	1.1842e+12	1.1842e+12	p_{43}	1.3572e+13	0
p_{12}	8.1389e+12	8.1389e+12	p_{44}	-2.2185e+13	0
p_{13}	4.7649e+13	4.7649e+13	p_{45}	8.6124e+12	0
p_{14}	2.3011e+14	2.3011e+14	p_{46}	2.2185e+13	0
p_{15}	8.7278e+14	8.7278e+14	p_{47}	1.3572e+13	0
p_{16}	2.3882e+15	2.3882e+15	p_{48}	4.9600e+12	0
p_{17}	3.9037e+15	3.9037e+15	p_{49}	1.3077e+12	0
p_{18}	1.5155e+15	1.5155e+15	p_{50}	2.7079e+11	0
p_{19}	-3.9037e+15	-3.9037e+15	p_{51}	4.6254e+10	0
p_{20}	2.3882e+15	2.3882e+15	p_{52}	6.7295e+09	0
p_{21}	-8.7278e+14	-8.7278e+14	p_{53}	8.5318e+08	0
p_{22}	2.3011e+14	2.3011e+14	p_{54}	9.5872e+07	0
p_{23}	-4.7649e+13	-4.7649e+13	p_{55}	9.6758e+06	0
p_{24}	8.1389e+12	8.1389e+12	p_{56}	8.8578e+05	0
p_{25}	-1.1842e+12	-1.1842e+12	p_{57}	6.7763e+04	0
p_{26}	1.5013e+11	1.5013e+11	p_{58}	-7.2625e+04	0
p_{27}	-1.6870e+10	-1.6870e+10	p_{59}	-1.0119e+06	0
p_{28}	1.7026e+09	1.7026e+09	p_{60}	-1.4094e+07	0
p_{29}	-1.5597e+08	-1.5606e+08	p_{61}	-2.1040e+08	0
p_{30}	1.3082e+07	1.4006e+07	p_{62}	-3.3522e+09	0
p_{31}	-1.0119e+06	-1.2010e+07	p_{63}	-5.6778e+10	0
p_{32}	7.2625e+04	0	p_{64}	-1.0186e+12	0

Table 3.2: Comparison of Computed Sturm Sequences of 64×64 $T_{\text{Wilkinson}}$ Matrix by using serial and parallel prefix algorithms at $x = 14$

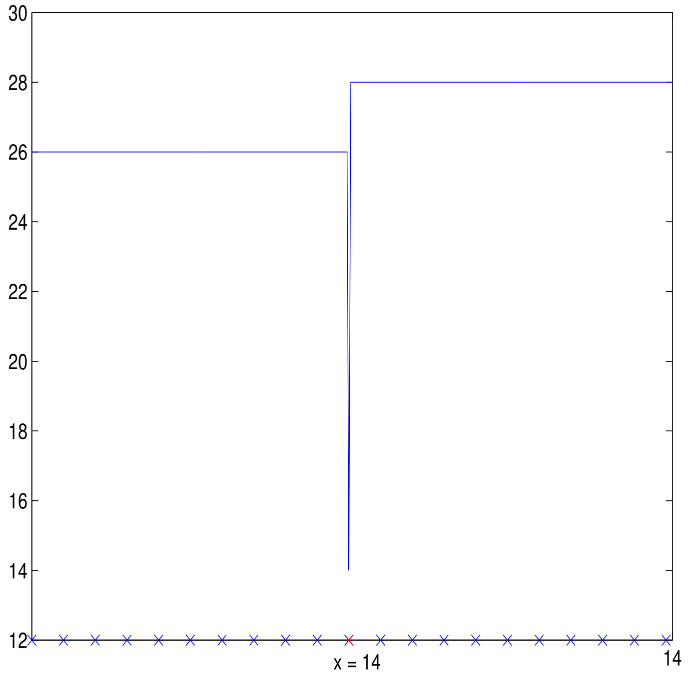


Figure 3.2: Computed Count(x) for 64×64 $T_{\text{Wilkinson}}$ Matrix in interval $[14(1-200\varepsilon), 14(1+200\varepsilon)]$ by Parallel Prefix Algorithm

Since each element of $T_{\text{Wilkinson}}$ is an integer, therefore all the elements of M_i are integers, so are the elements of any matrix product $M_{i:j}$. As we compute the products $M_{i:j}$ by parallel prefix, we find that until $M_{1:16}$, $M_{17:32}$, $M_{33:48}$, $M_{49:64}$, the computed $M_{i:j}$ are exactly the same as the corresponding true products(the products computed using exact arithmetic). However, when we multiply $M_{1:16}$ and $M_{17:32}$ together, due to the limited precision, the computed result of $M_{1:32}$ is *exactly* a zero matrix, and so is $M_{33:64}$. As a consequence, all of the computed $M_{1:i}$'s, when $i \geq 32$, are zero matrices. Therefore, the computed Sturm sequence p_i is zero when $i \geq 32$. Since we lost the information for a huge part(more than half) of the Sturm sequence, it is not surprising that the count dramatically decreases at the shift $x = 14$.

More precisely, we have

$$M_{1:16} = \begin{pmatrix} 217586071308601 & 133116815989000 \\ 0 & 0 \end{pmatrix},$$

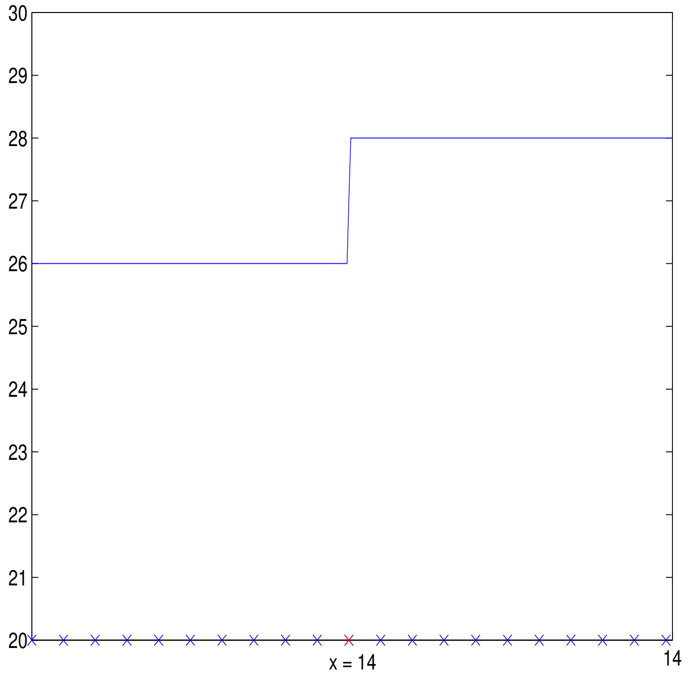


Figure 3.3: Computed Count(x) for 64×64 $T_{\text{Wilkinson}}$ Matrix in interval $[14(1-200\varepsilon), 14(1+200\varepsilon)]$ by Serial Algorithm

and

$$M_{17:32} = \begin{pmatrix} 33043478329 & -2373373752 \\ -54011212472 & 3879397705 \end{pmatrix}.$$

When we use IEEE double precision in MATLAB, $M_{1:16}\times M_{17:32}$ is a zero matrix; on the other hand, the true matrix product $M_{1:32}$ is the following matrix(we use MATHEMATICA with infinite precision, i.e. exact arithmetic):

$$\begin{pmatrix} -271 & 4048 \\ 0 & 0 \end{pmatrix}.$$

To compare the result of parallel prefix algorithm, we also use the serial algorithm to compute the Count(x) in the same tiny interval around 14. The results are plotted in figure 3.3. The computed function Count(x) is a monotonicly increasing function and the count at $x = 14$ is 27.

To conclude this section, there is an interesting phenomenon we want to mention. We computed the counts by parallel prefix for the floating-point numbers right before and after 14, the results turn out to be quite correct: the count for floating-point number right before 14 is 26 and after 14 is 28.

3.3 Examples of Nonmonotonicity of Count_Prefix

We have found many matrices for which parallel prefix algorithm computes a nonmonotonic Count(x). In this section, we show the computed Count(x) for two classes of test matrices.

- Glued random matrices. These matrices are generated by gluing 4×4 matrices whose independent random entries are uniformly distributed in $[-1, 1]$. More precisely, we generate a series of 4×4 random symmetric diagonal matrices, $T_1, T_2, \ldots, T_k$, the glued matrix is:

$$\begin{bmatrix} T_1 & \delta_1 & & & \\ \delta_1 & T_2 & \delta_2 & & \\ & \delta_2 & \ddots & \ddots & \\ & & \ddots & T_{k-1} & \delta_{k-1} \\ & & & \delta_{k-1} & T_k \end{bmatrix}$$

  where δ_i are small numbers.

- Glued positive definite matrices. These matrices are very similar to the examples in [74]. They are generated by gluing following 2×2 matrices:

$$\begin{bmatrix} 1 & 1-\alpha \\ 1-\alpha & 1 \end{bmatrix}.$$

We use parallel prefix algorithm to compute Count(x) in a small interval around the cluster of eigenvalues. Figure 3.4 plots the computed Count(x), by both parallel prefix(dotted red line) and serial algorithm(solid blue line) of a 32×32 glued random matrix in an interval contains a cluster of 7 eigenvalues; the magnitude of the eigenvalues are close to 0.8963, the length of the interval is $4 \cdot 10^{-6}$ and we sampled 400 floating point numbers in the interval. Similarly, figure 3.5 plots the computed Count(x) of a 32×32 glued positive definite matrix in an interval contains a cluster of 15 eigenvalues; the magnitude of the eigenvalues are close to 2, the length of the interval is $2 \cdot 10^{-5}$, we also sampled 400 floating point numbers in the interval.

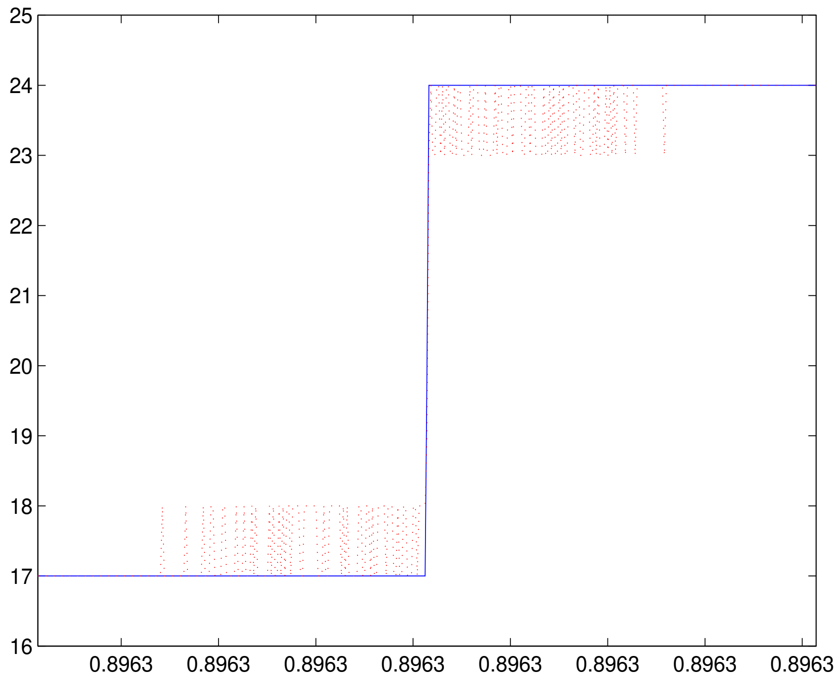


Figure 3.4: Computed Count(x) for 32×32 Glued Random Matrix by Serial (solid blue line) and Parallel Prefix (dotted red line) Algorithms

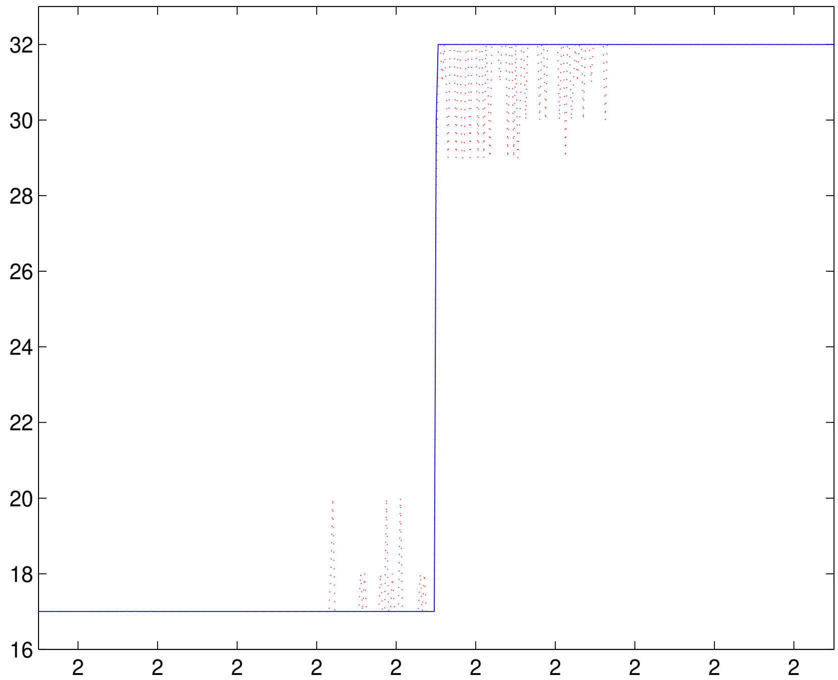


Figure 3.5: Computed Count(x) for 32×32 Glued Positive Definite Matrix by Serial (solid blue line) and Parallel Prefix (dotted red line) Algorithms

3.4 Backward Error Analysis

Our goal in this section is to analyze the backward stability of the parallel prefix algorithm, from which to see whether we can use the backward error to get some information of Count(x) even when the computed count is not correct.

When we compute Count(x) for T by either parallel prefix or serial algorithm, in exact arithmetic, the Sturm sequence p_i $(i = 1, 2, \ldots, n)$ is essentially computed by the three term recurrence (3.1.1)(without loss of generality, we assume shift $x = 0$):

$$p_i = a_i p_{i-1} - b_{i-1}^2 p_{i-2} \quad i = 1, 2, \ldots, n,$$

where $p_0 = 1$, $p_{-1} = 0$ and $b_0 = 0$.

We denote the computed Sturm sequence by $\hat{p}_i$ $(i = 1, 2, \ldots, n)$, such that $\hat{p}_i$ is the exact Sturm sequence of a perturbed symmetric tridiagonal matrix $\hat{T}$. We denote the perturbation matrix by δT so that $\hat{T} = T + \delta T$ where

$$\delta T = \begin{pmatrix} \delta a_1 & \delta b_1 & & & \\ \delta b_1 & \delta a_2 & \delta b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \delta b_{n-2} & \delta a_{n-1} & \delta b_{n-1} \\ & & & \delta b_{n-1} & \delta a_n \end{pmatrix}.$$

Thus, for $i = 1, 2, \ldots, n$, we have

$$\hat{p}_i = (a_i + \delta a_i)\hat{p}_{i-1} - (b_{i-1} + \delta b_{i-1})^2 \hat{p}_{i-2},$$

let $\xi_i = \max\{|\delta a_i|, |\delta b_{i-1}|\}$ and for $i = 1, 2, \ldots, n$,

$$\hat{P}_i = a_i \hat{p}_{i-1} - b_{i-1}^2 \hat{p}_{i-2}.$$

Then the residuals can be bounded as follows by ignoring the second order term δb_{i-1}^2:

$$\begin{aligned} |\hat{p}_i - \hat{P}_i| &= |\delta a_i \hat{p}_{i-1} - 2b_{i-1}\delta b_{i-1}\hat{p}_{i-2}| \\ &\leq \xi_i(|\hat{p}_{i-1}| + 2|b_{i-1}||\hat{p}_{i-2}|), \end{aligned}$$

where the equality can be attained when $\delta a_i = \xi_i \cdot \mathrm{sign}\ (\hat{p}_{i-1})$ and $\delta b_{i-1} = -\xi_i \cdot \mathrm{sign}\ (b_{i-1}\hat{p}_{i-2})$. Therefore,

$$\xi_i = \frac{|\hat{p}_i - \hat{P}_i|}{|\hat{p}_{i-1}| + 2|b_{i-1}||\hat{p}_{i-2}|}.$$

Let $\xi = \max_i\{\xi_i\}$, which is a componentwise bound for backward error, we have

$$\|\delta T\|_1 = \max_i(|\delta a_i| + |\delta b_i| + |\delta b_{i-1}|) \leq 3 \max_i \xi_i \leq 3\xi.$$

Since δT is a symmetric tridiagonal matrix, therefore $\|\delta T\|_2 \leq \|\delta T\|_1$ [80]. Hence,

$$\|\delta T\|_2 \leq \|\delta T\|_1 \leq 3\xi.$$

Since ξ can be estimated very cheaply, we like to know what kind of extra information ξ can offer, such that we are able to get the correct count even though parallel prefix might give us an incorrect one. We discuss two cases: when the computed counts at two shifts are equal, and when they are not equal.

3.4.1 When Computed Counts at Two Different Shifts are Equal

Assume that we have computed the counts at two different shifts: x and y, with $x < y$. The absolute backward error bounds, ξ, for x and y are denoted by ξ_x and ξ_y respectively. Also we assume $\hat{T}_x = T + \delta T_x$ and $\hat{T}_y = T + \delta T_y$, where δT_x and δT_y are perturbation matrices of T at x and y. Therefore,

$$\|\delta T_x\|_2 \leq 3\xi_x$$

and

$$\|\delta T_y\|_2 \leq 3\xi_y.$$

For $i = 1, 2, \ldots, n$, let $\hat{\lambda}_i^x$ and $\hat{\lambda}_i^y$ be the eigenvalues of $\hat{T}_x$ and $\hat{T}_y$ and λ_i be the exact eigenvalues of T. By Weyl's Theorem [80], we know that for each i, we have the following bounds:

$$\lambda_i - 3\xi_x \leq \lambda_i - \|\delta T_x\|_2 \leq \hat{\lambda}_i^x \leq \lambda_i + \|\delta T_x\|_2 \leq \lambda_i + 3\xi_x. \quad (3.4.6)$$

$$\lambda_i - 3\xi_y \leq \lambda_i - \|\delta T_y\|_2 \leq \hat{\lambda}_i^y \leq \lambda_i + \|\delta T_y\|_2 \leq \lambda_i + 3\xi_y. \quad (3.4.7)$$

As before, we denote the computed count by FloatingCount, and the exact count by Count. Assume FloatingCount(x) = FloatingCount$(y) = k$, so

$$\hat{\lambda}_1^x \leq \hat{\lambda}_2^x \leq \ldots \leq \hat{\lambda}_k^x < x.$$

From (3.4.6), we know that for $i = 1, 2, \ldots, k$,

$$\lambda_i \leq \hat{\lambda}_i^x + 3\xi_x < x + 3\xi_x.$$

which implies

$$\text{Count}(x + 3\xi_x) \geq k.$$

On the other hand, since FloatingCount$(y) = k$, thus

$$\hat{\lambda}^y_{k+1} \geq y.$$

From (3.4.7), we can conclude that

$$\lambda_{k+1} \geq \hat{\lambda}^y_{k+1} - 3\xi_y \geq y - 3\xi_y,$$

which implies

$$\text{Count}(y - 3\xi_y) \leq k.$$

If ξ_x and ξ_y are small enough, then we can assume that $x + 3\xi_x < y - 3\xi_y$, therefore, we have

$$k \leq \text{Count}(x + 3\xi_x) \leq \text{Count}(y - 3\xi_y) \leq k.$$

Equivalently,

$$\text{Count}(x + 3\xi_x) = \text{Count}(y - 3\xi_y) = k.$$

This implies that there is no eigenvalue in the interval $(x + 3\xi_x, y - 3\xi_y)$.

3.4.2 When Computed Counts at Two Different Shifts are Unequal

We make the same assumptions as those in last subsection except we assume that the computed counts at x and y are k_1 and k_2 respectly, and $k_2 - k_1 = k$.

Since FloatingCount$(x) = k_1$, thus

$$\hat{\lambda}^x_{k_1+1} \geq x.$$

From (3.4.6), we have

$$x - 3\xi_x \leq \hat{\lambda}^x_{k_1+1} - 3\xi_x \leq \lambda_{k_1+1}.$$

Therefore,

$$\text{Count}(x - 3\xi_x) \leq k_1.$$

On the other hand, since FloatingCount$(y) = k_2$, so

$$\hat{\lambda}^y_1 \leq \hat{\lambda}^y_2 \leq \ldots \leq \hat{\lambda}^y_{k_2} < y.$$

From (3.4.7), we know that for $i = 1, 2, \ldots, k_2$,

$$\lambda_i \leq \hat{\lambda}_i^y + 3\xi_y < y + 3\xi_y.$$

Therefore,

$$\text{Count}(y + 3\xi_y) \geq k_2.$$

Hence, we know that there are at least k eigenvalues of T in the interval $(x - 3\xi_x, y + 3\xi_y)$.

However, the numerical experiments show that ξ is often too large to be useful. We show the numerical experiments in next subsection.

3.4.3 Numerical Experiments

Besides considering ξ, an absolute backward error bound, we also consider the relative backward error. Again, let $\hat{T} = T + \delta T$, but we express δT in a different way:

$$\delta T = \begin{pmatrix} a_1 \cdot \triangle a_1 & b_1 \cdot \triangle b_1 & & & \\ b_1 \cdot \triangle b_1 & a_2 \cdot \triangle a_2 & b_2 \cdot \triangle b_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} \cdot \triangle b_{n-2} & a_{n-1} \cdot \triangle a_{n-1} & b_{n-1} \cdot \triangle b_{n-1} \\ & & & b_{n-1} \cdot \triangle b_{n-1} & a_n \cdot \triangle a_n \end{pmatrix}.$$

Thus,

$$\hat{p}_i = a_i(1 + \triangle a_i)\hat{p}_{i-1} - b_{i-1}^2(1 + \triangle b_{i-1})^2 \hat{p}_{i-2},$$

let $\eta_i = \max\{|\triangle a_i|, |\triangle b_{i-1}|\}$ and $\hat{P}_i = a_i \hat{p}_{i-1} - b_{i-1}^2 \hat{p}_{i-2}$, then,

$$\begin{aligned} |\hat{p}_i - \hat{P}_i| &= |a_i \triangle a_i \hat{p}_{i-1} - 2b_{i-1}^2 \triangle b_{i-1} \hat{p}_{i-2} - b_{i-1}^2 \triangle b_{i-1}^2 \hat{p}_{i-2}| \\ &= |a_i \triangle a_i \hat{p}_{i-1} - 2b_{i-1}^2 \triangle b_{i-1} \hat{p}_{i-2}| \\ &\leq \eta_i (|a_i||\hat{p}_{i-1}| + 2b_{i-1}^2 |\hat{p}_{i-2}|), \end{aligned}$$

by ignoring the second order term $\triangle b_{i-1}^2$. Therefore,

$$\eta_i = \frac{|\hat{p}_i - \hat{P}_i|}{|a_i||\hat{p}_{i-1}| + 2b_{i-1}^2 |\hat{p}_{i-2}|}.$$

Finally, the relative backward error bound η can be expressed as follows:

$$\eta = \max_i \eta_i = \max_i \frac{|\hat{p}_i - \hat{P}_i|}{|a_i||\hat{p}_{i-1}| + 2b_{i-1}^2 |\hat{p}_{i-2}|}.$$

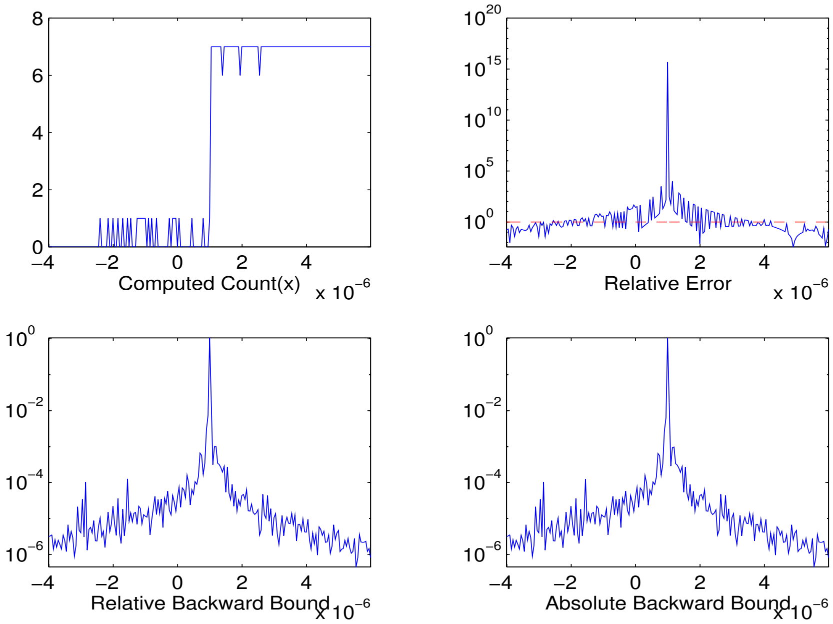


Figure 3.6: Computed Count(x) and Backward Errors for 16×16 Glued Positive Definite Matrix, length of interval $= 10^{-5}$

We present the numerical experiments for four kinds of test matrices: glued positive definite matrix, glued random matrix, glued Wilkinson-like matrix which is glued by two 32×32 Wilkinson-like matrices, and random matrix with entries independently and uniformly distributed on $[-1, 1]$. Each figure contains four plots, the computed Count(x) by parallel prefix, the *relative error* of computed Count(x), the relative and absolute backward error bounds η and ξ. The relative error of computed Count(x) by parallel prefix means relative to the serial algorithm. More precisely, let $\hat{p}_i$ be the Sturm sequence computed by parallel prefix, and let $\hat{p}_i^{\text{serial}}$ be the Sturm sequence computed by the serial algorithm. Then the relative error is expressed by the following formula:

$$\textbf{Relative Error} \quad = \max_i \frac{|\hat{p}_i^{\text{serial}} - \hat{p}_i|}{|\hat{p}_i^{\text{serial}}|}. \tag{3.4.8}$$

Figure 3.6 plots the results of a 16×16 glued random matrix; figure 3.7 plots the results of a 32×32 glued random matrix; figure 3.8 plots the results of a 64×64 glued Wilkinson-like matrix; and figure 3.9 plots the results of a 64×64 random matrix. Each plot is sampled at 400 floating point numbers.

Clearly, for the error analysis in the two previous subsections to be useful, the

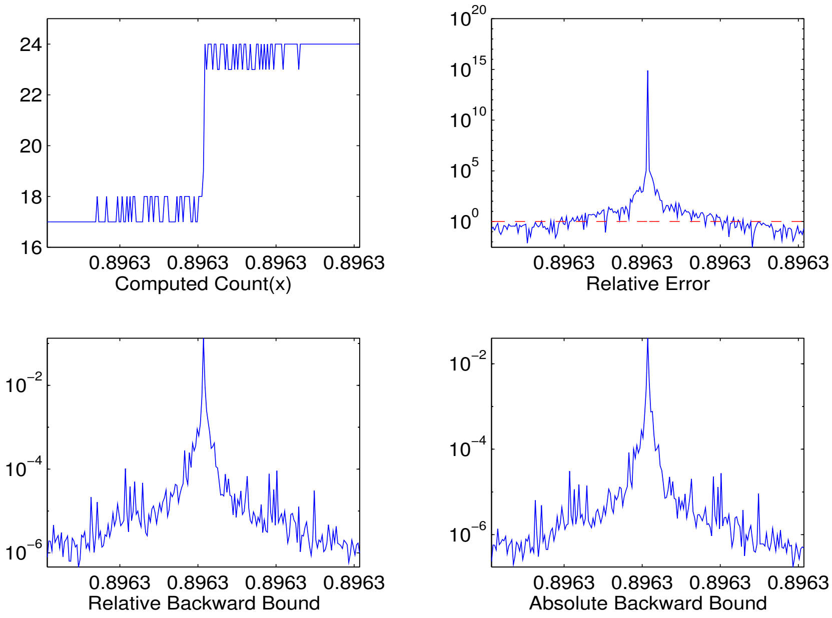


Figure 3.7: Computed Count(x) and Backward Errors for 32×32 Glued Random Matrix, length of interval $= 4 \cdot 10^{-6}$

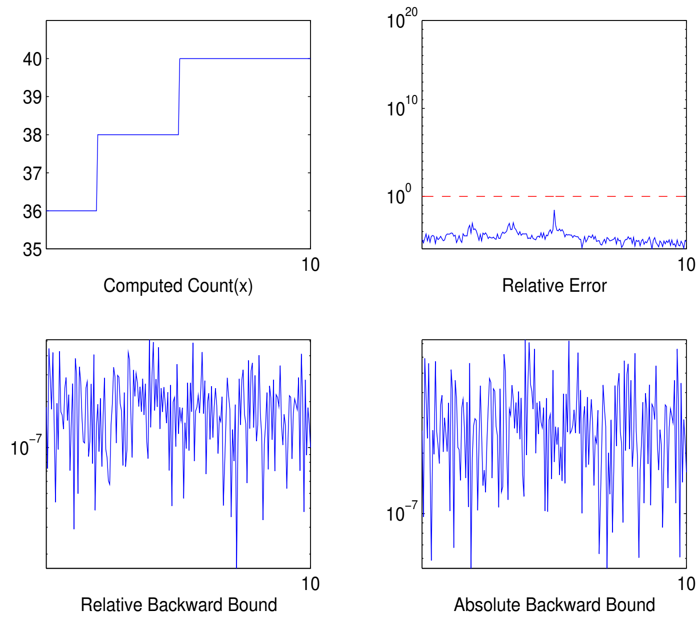


Figure 3.8: Computed Count(x) and Backward Errors for 64×64 Glued Wilkinson-like Matrix, length of interval $= 4 \cdot 10^{-11}$

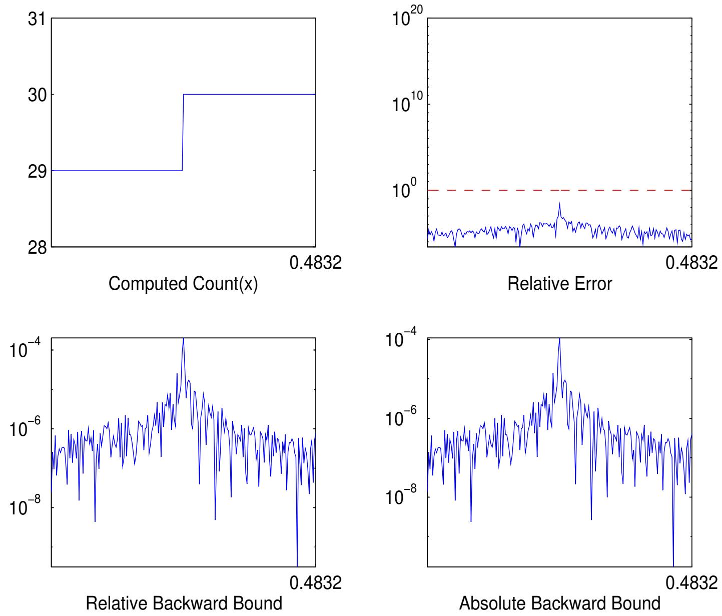


Figure 3.9: Computed Count(x) and Backward Errors for 64×64 Random Matrix, length of interval $= 2 \cdot 12^{-12}$

backward error bounds ξ and η have be small. However, for the figures we present here, the backward error bounds are frequently too large to use. For example, in figure 3.9, even if parallel prefix computes a correct count, the backward error bounds are still large.

One can argue that we can use ξ and η as a criteria to determine whether the parallel prefix computes an incorrect count. But, from the numerical experiments, we have no idea what kind of magnitude we should set for a tolerance τ such that if ξ or η is larger than τ, then we know parallel prefix might compute an incorrect count.

Chapter 4

Forward Error Analysis and Iterative Refinement

4.1 Introduction

In this chapter, we analyze the instability of the parallel prefix algorithm by using forward error analysis, and also discuss using iterative refinement with parallel prefix. We first review some results for symmetric positive definite tridiagonal matrix [74]. Then we present a rather complicated error bound for a general symmetric tridiagonal matrix, and show some examples of computed forward error bounds, and finally discuss iterative refinement.

4.2 Previous Work for Symmetric Positive Definite Tridiagonal Matrix

We review some analysis developed for symmetric positive definite tridiagonal matrices by Mathias [74], and compare the results with the results obtained for conventional algorithms [12, 35, 33]. Let $\hat{M}_{i:j}$ be the computed product $M_{i:j}$, where $M_{i:j} = M_i \cdot M_{i+1} \cdots M_j$, and $|M_{i:j}|$ be the matrix whose elements are the absolute values of the elements of $M_{i:j}$. Suppose that during the process of computing $M_{1:r}$, $1 \leq r \leq n$, all the matrix multiplications are computed exactly except multiplication of $M_{i+1:j}$ and $M_{j+1:k}$. Denote the corresponding error by $E_{i,j,k}$, so

$$E_{i,j,k} = fl(M_{i+1:j}M_{j+1:k}) - M_{i+1:j}M_{j+1:k} = \hat{M}_{i+1:k} - M_{i+1:k}.$$

Therefore,

$$\hat{M}_{1:r} = M_{1:i}\hat{M}_{i+1:k}M_{k+1:r} = M_{1:i}(M_{i+1:k} + E_{i,j,k})M_{k+1:r} = M_{1:r} + M_{1:i}E_{i,j,k}M_{k+1:r}.$$

Thus,

$$|E_{1:r}| = |\hat{M}_{1:r} - M_{1:r}| = |M_{1:i}E_{i,j,k}M_{k+1:r}| \leq |M_{1:i}||E_{i,j,k}||M_{k+1:r}|.$$

Assume we use inner(or outer) products to perform matrix multiplication. From [43], we know that

$$|E_{i,j,k}| = |fl(M_{i+1:j}M_{j+1:k}) - M_{i+1:j}M_{j+1:k}| \leq 2\varepsilon|M_{i+1:j}||M_{j+1:k}|.$$

Hence,

$$|E_{1:r}| \leq 2\varepsilon|M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:r}|.$$

We should mention that the indices i, j, k can not be chosen arbitrarily: i and k depend on j and n. Finally, taking into account the error made for each matrix multiplication during the process of computing $M_{1:r}$, and ignoring the second and higher order terms, we have the following forward error bound:

$$|\hat{M}_{1:r} - M_{1:r}| \leq 2\varepsilon \sum_{j=1}^{r-1} |M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:r}|. \tag{4.2.1}$$

In particular,

$$|\hat{M}_{1:n} - M_{1:n}| \leq 2\varepsilon \sum_{j=1}^{n-1} |M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:n}|. \tag{4.2.2}$$

Since the Sturm sequence component p_r is the $(1,1)$ element of the matrix $M_{1:r}$, the error $|p_r - \hat{p}_r|$ in the computed Sturm sequence component $\hat{p}_r$ is bounded by the $(1,1)$ element of the matrix $2\varepsilon\sum_{j=1}^{r-1}|M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:r}|$.

Definition 4.2.1 Given a symmetric tridiagonal $n \times n$ matrix T and indices i_j satisfying $1 \leq i_1 < i_2 < \cdots < i_k \leq n-1$,

i. For indices $1 \leq i_j \leq n-1$, $j = 1, 2, \ldots, k$, $T[i_1, i_2, \ldots, i_k]$ is a symmetric tridiagonal matrix such that

$$T_{pq}[i_1, i_2, \ldots, i_k] = T_{pq} \quad \text{for all } p \text{ and } q,$$

except

$$T_{i_j i_{j+1}}[i_1, i_2, \ldots, i_k] = 0 \quad \text{for } 1 \le j \le k.$$

ii. For $1 \le i < j \le n$, $T(i:j)$ is the principal submatrix of rows and columns $i, i+1, \ldots, j$ of T.

We use "det" to denote the determinant of a matrix. By using the properties of symmetric positive definite tridiagonal matrix, the following inequalities can be proved:

Lemma 4.2.1 (Mathias [74]) *Let T be a symmetric positive definite tridiagonal matrix, and let $i_1, \ldots, i_k \in \{1, 2, \ldots, n-1\}$. Then*

$$\det(T) \le \det(T[i_1, i_2, \ldots, i_k]) \le \frac{\lambda_n(A)^k}{\lambda_1(A) \cdots \lambda_k(A)} \det(T)$$

where $A = DTD$ and D is any $n \times n$ nonsingular diagonal matrix,

Lemma 4.2.2 (Mathias [74]) *Let T be a symmetric positive definite tridiagonal matrix and let*

$$M_i = \begin{bmatrix} a_i & 1 \\ -b_{i-1}^2 & 0 \end{bmatrix}.$$

Let $1 \le i \le j \le k \le n$,

$$P = |M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:n}|,$$

and

$$D = \det(T(1:i)) \det(T(i+1:j)) \det(T(j+1:k)) \det(T(k+1:n)).$$

Then

$$D \le P_{11} \le 8D,$$

where P_{11} is the $(1,1)$ element of matrix P.

By using these two lemmas and some other properties of symmetric positive definite tridiagonal matrix, the forward error bounds can be obtained.

Theorem 4.2.1 (Mathias [74]) *Let T be a symmetric positive definite tridiagonal matrix. Let $\hat{p}_i$ be the computed Sturm sequence by parallel prefix, then for $1 \le r \le n$,*

$$\begin{aligned} \left|\frac{p_r - \hat{p}_r}{p_r}\right| &\le 16\varepsilon \sum_{j=1}^{r-1} \frac{\det(T[i,j,k^{(r)}])}{\det(T(1:r))} \\ &\le 16\varepsilon \sum_{j=1}^{n-1} \frac{\det(T[i,j,k])}{\det(T(1:n))}. \end{aligned}$$

Corollary 4.2.1 (Mathias [74]) *Let T be a symmetric positive definite tridiagonal matrix. Let $\hat{p}_i$ be the computed Sturm sequence by parallel prefix. Let $A = DTD$, where D is diagonal and chosen so that the main diagonal entries of A are all 1. Then for $1 \le r \le n$,*

$$|\frac{p_r - \hat{p}_r}{p_r}| \le 16(r-1)\frac{\varepsilon \cdot \lambda_n(A)^3}{\lambda_1(A)\lambda_2(A)\lambda_3(A)}.$$

Moreover, if the smallest eigenvalues of A are close to each other, i.e. $\lambda_1(A) \approx \lambda_2(A) \approx \lambda_3(A)$, then

$$|\frac{p_r - \hat{p}_r}{p_r}| \le 16(r-1)\varepsilon\kappa^3,$$

where $\kappa = \lambda_n(A)/\lambda_1(A)$ is the condition number.

In contrast, if we use the conventional serial algorithm, we can get high relative accuracy results for computing the eigenvalues and singular values. The following theorems can be found in [12, 33, 35]. Here we present two of them.

Theorem 4.2.2 (Demmel and Veselić [35]) *Let $H = DAD$ be a symmetric positive definite matrix, and $D = diag(H_{ii}^{1/2})$ so $A_{ii} = 1$. Let $\delta H = D\delta AD$ be a perturbation such that $\|\delta A\|_2 \equiv \eta < \lambda_{\min}(A)$. Let λ_i be the ith eigenvalue of H and $\hat{\lambda}_i$ be the ith eigenvalue of $H + \delta H$. Then*

$$|\frac{\lambda_i - \hat{\lambda}_i}{\lambda_i} \le \frac{\eta}{\lambda_{\min}(A)}| \le \kappa(A) \cdot \eta,$$

where $\kappa(A)$ is the condition number of A.

For singular values, we have

Theorem 4.2.3 *(**(Barlow and Demmel [12], Demmel and Kahan [33])** Let B and $B + \delta B$ be bidiagonal with singular values $\sigma_1(B) \le \cdots \le \sigma_n(B)$ and $\sigma_1(B + \delta B) \le \cdots \le \sigma_n(B + \delta B)$, respectively. If for all nonzero entries B_{ij},*

$$\tau^{-1} \le |\frac{(B + \delta B)_{ij}}{B_{ij}}| \le \tau$$

for some $\tau \ge 1$, then

$$\frac{1}{\tau^{2n-1}} \le \frac{\sigma_i(B + \delta B)}{\sigma_i(B)} \le \tau^{2n-1}.$$

Thus, relative perturbations of at most τ in the entries of B can cause relative perturbations of at most τ^{2n-1} in its singular values. If $\tau = 1 + \eta$ is close to 1, so is $\tau^{2n-1} \approx 1 + (2n-1)\eta$.

If we use the serial algorithm, the errors in the computed Sturm sequence can be bounded as follows.

Theorem 4.2.4 (Mathias [74]) *Let T be a symmetric positive definite tridiagonal matrix. Let $\hat{p}_i$ be the serially computed Sturm sequence values. Let $A = DTD$, where D is diagonal and chosen so that the main diagonal entries of A are all 1. If $\lambda_1(A) \geq 4\varepsilon$, then for $1 \leq r \leq n$,*

$$|\frac{p_r - \hat{p}_r}{p_r}| \leq \frac{4r\varepsilon}{\lambda_1(A)}. \tag{4.2.3}$$

The theorems we have presented explain why Count(x) computed by parallel prefix is so inaccurate, at least when the symmetric tridiagonal matrix is positive definite. The forward relative error for computed Sturm sequence $\hat{p}_i$ by parallel prefix can be as large as $\varepsilon\kappa^3$, whereas it can be bounded by $\varepsilon\kappa$ when computed serially.

Since in general we need to compute the Sturm sequence of $T - xI$, for x inside the spectrum of T, $T - xI$ will not be positive definite. Therefore, to fully explain the inaccuracy of computed Count(x) by parallel prefix, we need to extend the forward analysis to the general symmetric tridiagonal matrix. In the next several sections, we will present such a general analysis.

4.3 Numerical Experiments

We first present several figures which plot the Count(x) and the corresponding forward error bound

$$2\varepsilon \sum_{j=1}^{n-1} |M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:n}| \tag{4.3.4}$$

in some intervals containing some eigenvalue or clusters of eigenvalues. As we did for backward error bounds, we present the numerical experiments for four kinds of test matrices: glued positive definite matrix, glued random matrix, glued Wilkinson-like matrix which is glued by two 32×32 Wilkinson-like matrices, and random matrix with entries independently and uniformly distributed on $[-1, 1]$. Each figure contains three plots, the computed Count(x) by parallel prefix, the relative error (see definition in (3.4.8)) and the forward relative error bound 4.3.4.

Figure 4.1 plots the results of a 16×16 glued random matrix; figure 4.2 plots the results of a 32×32 glued random matrix; figure 4.3 plots the results of a 64×64 glued Wilkinson-like matrix; and figure 4.4 plots the results of a 64×64 random matrix.

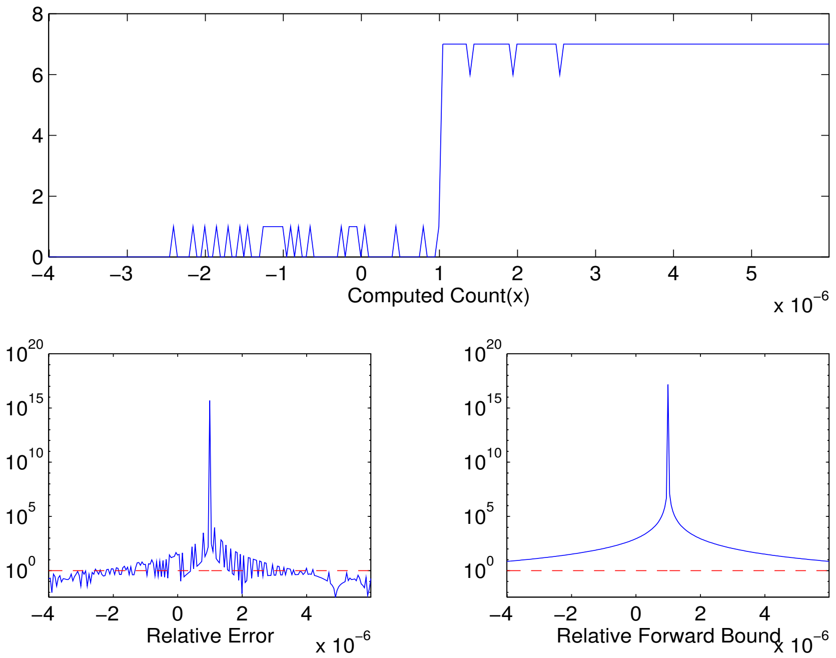


Figure 4.1: Computed Count(x) and Forward Error Bound for 16×16 Glued Positive Definite Matrix, length of interval $= 10^{-5}$

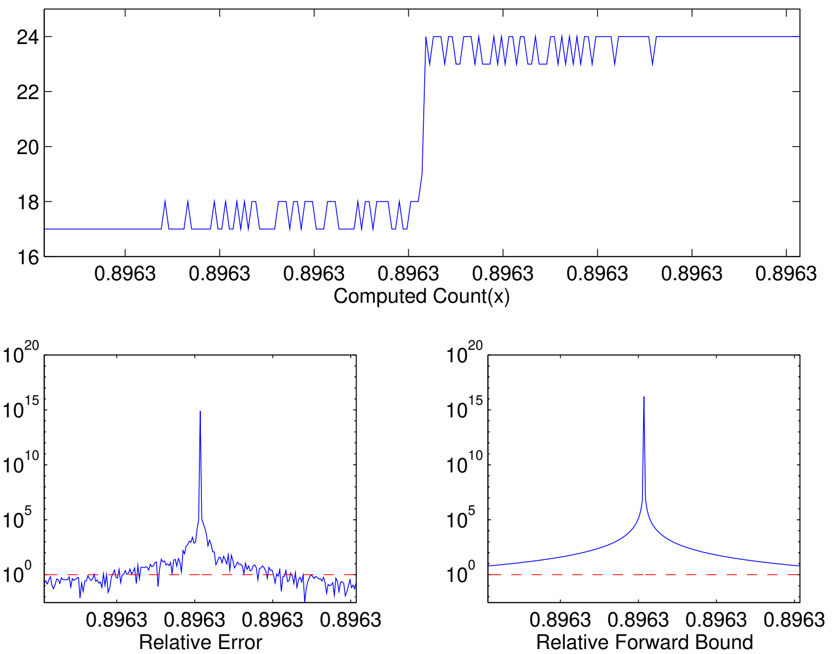


Figure 4.2: Computed Count(x) and Forward Error Bound for 32×32 Glued Random Matrix, length of interval $= 4 \cdot 10^{-6}$

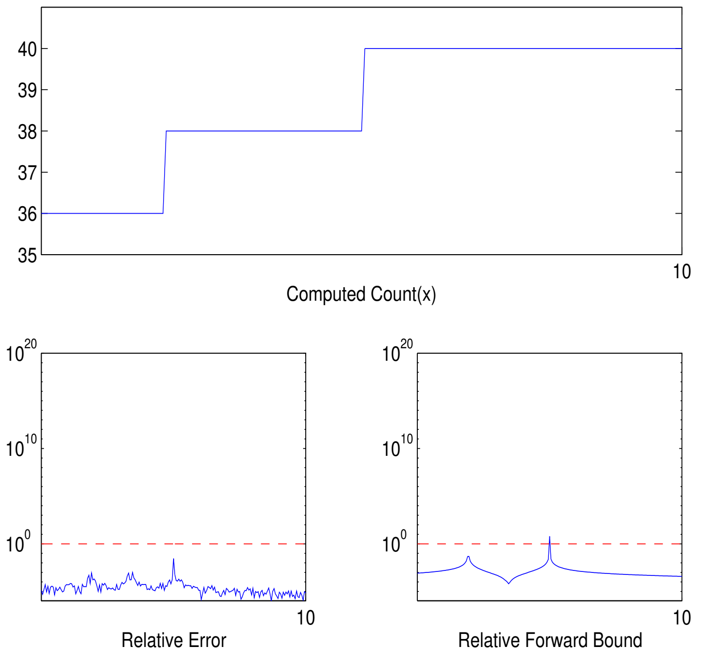


Figure 4.3: Computed Count(x) and Forward Error Bound for 64×64 Glued Wilkinson-like Matrix, length of interval $= 4 \cdot 10^{-11}$

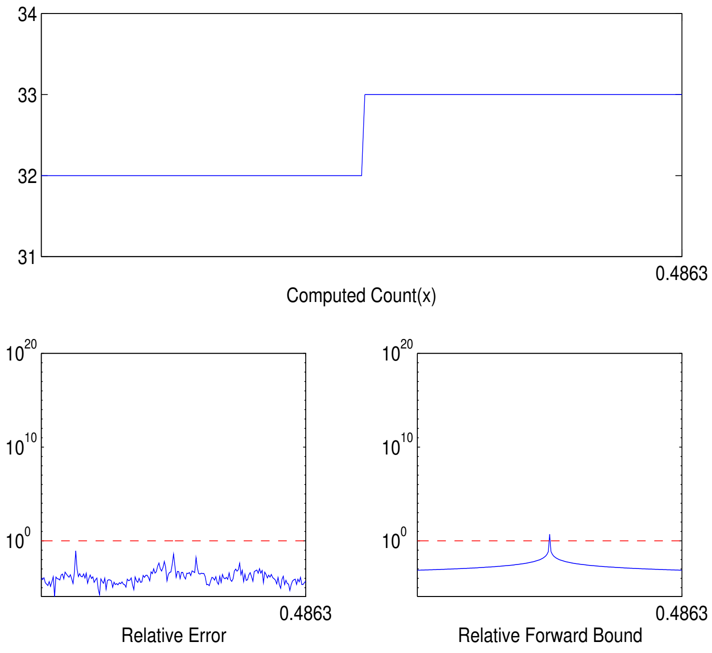


Figure 4.4: Computed Count(x) and Forward Error Bound for 64×64 Random Matrix, length of interval $= 2 \cdot 12^{-12}$

4.4 Some Properties of Symmetric Tridiagonal Matrices

In this section we introduce some interesting relations between singular values of a symmetric tridiagonal matrix T and $T([i_1, i_2, \ldots, i_k])$.

Lemma 4.4.1 *Let T be an $n \times n$ symmetric tridiagonal matrix and let i_1, i_2, ..., $i_k \in \{1, 2, \ldots, n-1\}$. Then*

$$\sigma_i(T[i_1, i_2, \ldots, i_k]) \leq \sigma_n(T) \quad i = 1, 2, \ldots, n,$$

where $\sigma_1 \leq \sigma_2 \leq \cdots \leq \sigma_n$ are the singular values.

Proof. By the Cauchy-Interlace Theorem [80], the eigenvalues of each diagonal block T_i of

$$T[i_1, i_2, \ldots, i_k] = \begin{pmatrix} T_1 & & & \\ & T_2 & & \\ & & \ddots & \\ & & & T_{k+1} \end{pmatrix}$$

satisfies

$$\lambda_1(T) \leq \lambda(T_i) \leq \lambda_n(T).$$

So for every eigenvalue λ of $T[i_1, i_2, \ldots, i_k]$, it must satisfy:

$$\lambda_1(T) \leq \lambda(T[i_1, i_2, \ldots, i_k]) \leq \lambda_n(T),$$

and

$$|\lambda(T[i_1, i_2, \ldots, i_k])| \leq \max(|\lambda_1(T)|, |\lambda_n(T)|).$$

Therefore,

$$\sigma_i(T[i_1, i_2, \ldots, i_k]) \leq \sigma_n(T).$$

∎

Lemma 4.4.2 *Let T be an $n \times n$ symmetric tridiagonal matrix, let i_1, i_2, ..., $i_k \in \{1, 2, \ldots, n-1\}$, then*

$$\sigma_{i-2k}(T[i_1, i_2, \ldots, i_k]) \leq \sigma_i(T) \quad i = 2k+1, 2k+2, \ldots, n.$$

Proof. When $k = 1$,

$$T[i_1] = \begin{pmatrix} T_1 & \\ & T_2 \end{pmatrix} = T - \begin{pmatrix} 0 & & & & & \\ & \ddots & & & & \\ & & 0 & b & & \\ & & b & 0 & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix}.$$

Without loss of generality, we can assume $b > 0$. Notice that

$$\begin{aligned} T[i_1] &= T + \frac{1}{2}\begin{pmatrix} 0 & & & & & \\ & \ddots & & & & \\ & & b & -b & & \\ & & -b & b & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix} - \frac{1}{2}\begin{pmatrix} 0 & & & & & \\ & \ddots & & & & \\ & & b & b & & \\ & & b & b & & \\ & & & & \ddots & \\ & & & & & 0 \end{pmatrix} \\ &= T + B_1 - B_2. \end{aligned}$$

Since B_1 and B_2 are both semidefinite, rank-1 matrices, by Weyl's Monotonicity Theorem [80],

$$\lambda_{i-1}(T) \leq \lambda_i(T - B_2) \leq \lambda_i(T[i_1]) \leq \lambda_i(T + B_1) \leq \lambda_{i+1}(T), \tag{4.4.5}$$

i.e., $\lambda_i(T[i_1]) \in [\lambda_{i-1}(T), \lambda_{i+1}(T)]$.

It is well known that the singular values of a symmetric matrix are simply the absolute values of the eigenvalues. Assume that $\sigma_l(T) = |\lambda_i(T)|$ for some i. If $\lambda_i(T) > 0$, we argue that $i \geq l$. In fact, there are $n - i$ of $\lambda_p(T)$ which are larger than $\lambda_i(T)$, namely $\lambda_{i+1}, \lambda_{i+2}, \ldots, \lambda_n$. Therefore, there are at least $n-i$ singular values are larger than $\sigma_l(T) = \lambda_i(T)$, hence $i \geq l$.

By (4.4.5), we know that $\lambda_{i-1}(T[i_1]) \leq \lambda_i(T)$, therefore, there are at most $n-i+1$ of $\lambda_j(T[i_1])$ which are larger than $\lambda_i(T)$, namely $\lambda_i(T[i_1]), \lambda_{i+1}(T[i_1]), \ldots, \lambda_n(T[i_1])$.

Since $\sigma_l(T) = \lambda_i(T)$, there must be $l - 1$ eigenvalues of T(those eigenvalues whose absolute values are the singular values $\sigma_1(T), \sigma_2(T), \ldots, \sigma_{l-1}(T)$) in the interval $(-\lambda_i(T), \lambda_i(T))$; otherwise, it contradicts with the fact that $\lambda_i(T)$ is the l-th singular value of T. Therefore,

$$\lambda_{i-l}(T) \leq -\lambda_i(T) < \lambda_{i-l+1}(T).$$

By (4.4.5), $\lambda_{i-l+1}(T) \le \lambda_{i-l+2}(T[i_1])$, so we conclude that there are at most $i-l+1$ of $\lambda_j(T[i_1])$ which are less than $-\lambda_i(T)$. Therefore, there are at most $(n-i+1)+(i-l+1) = n-l+2$ of $\sigma_j(T[i_1])$ which are greater than $\sigma_l(T)$, which implies

$$\sigma_{l-2}(T[i_1]) \le \sigma_l(T).$$

By doing a similar count, we can prove the above inequality for $\lambda_i(T) < 0$.

By induction, if the lemma is true for k, then for $k+1$, we have

$$\sigma_i(T[i_1, i_2, \ldots, i_k, i_{k+1}]) \le \sigma_{i-2k}(T[i_{k+1}]) \le \sigma_{i-2(k+1)}(T).$$

∎

Lemma 4.4.3 *Let T be an $n \times n$ symmetric tridiagonal matrix and i_1, i_2, …, $i_k \in \{1, 2, \ldots, n-1\}$. Then*

$$|\det(T[i_1, i_2, \ldots, i_k])| \le \frac{\sigma_n^{2k}(T)}{\sigma_1(T)\ldots\sigma_{2k}(T)} |\det(T)|.$$

Proof.

$$\begin{aligned} |\det(T[i_1, i_2, \ldots, i_k])| &= \prod_{i=1}^{n-2k} \sigma_i(T[i_1, i_2, \ldots, i_k]) \prod_{i=n-2k+1}^{n} \sigma_i(T[i_1, i_2, \ldots, i_k]) \\ &\le \sigma_n^{2k}(T) \prod_{i=1}^{n-2k} \sigma_{i+2k}(T) \quad \text{by Lemma 4.4.2} \\ &= \sigma_n^{2k}(T) \prod_{i=2k+1}^{n} \sigma_i(T) \\ &= \frac{\sigma_n^{2k}(T)}{\sigma_1(T)\ldots\sigma_{2k}(T)} |\det(T)| \end{aligned}$$

∎

Lemma 4.4.4 *Let H be the $n-1 \times n-1$ principal submatrix of rows and columns $1, 2, \ldots, n-1$ or $2, 3, \ldots, n$ of the symmetric tridiagonal matrix T, Then*

$$\sigma_i(H) \le \sigma_{i+1}(T) \quad i = 1, 2, \ldots, n-1.$$

Proof. When H is the leading principal matrix,

$$T = \begin{pmatrix} H & \beta \\ \beta^T & \alpha \end{pmatrix},$$

where β is a vector and α is a real number. Notice that

$$T^2 = T^T \cdot T = \begin{pmatrix} H^2 + \beta\beta^T & \gamma \\ \gamma^T & \mu \end{pmatrix},$$

where γ is a vector and μ is a real number. By the Cauchy-Interlace Theorem and Weyl's Monotonicity Theorem [80], we know that for $i = 1, 2, \ldots, n-1$,

$$\sigma_i^2(H) = \lambda_i(H^2) \leq \lambda_i(H^2 + \beta\beta^T) \leq \lambda_{i+1}(T^2) = \sigma_{i+1}^2(T).$$

which implies $\sigma_i(H) \leq \sigma_{i+1}(T)$.

Similarly, we can prove the same result when H is the principal matrix of rows and columns $2, 3, \ldots, n$. ∎

Lemma 4.4.5 *Let H be the $(n-1) \times (n-1)$ principal submatrix of rows and columns $1, 2, \ldots, n-1$ or $2, 3, \ldots, n$ of the symmetric tridiagonal matrix T, Then*

$$|\det(H)| \leq \frac{|\det(T)|}{\sigma_1}.$$

Proof.

$$\begin{aligned} |\det(H)| &= \prod_{i=1}^{n-1} \sigma_i(H) \leq \prod_{i=1}^{n-1} \sigma_{i+1}(T) \\ &= \prod_{i=2}^{n} \sigma_i(T) = \frac{|\det(T)|}{\sigma_1(T)}. \end{aligned}$$

∎

4.5 Forward Error Bound for Symmetric Tridiagonal Matrix

In the previous section, we introduced a general forward error bound (4.2.1). In this section, we discuss in particular how to relate the error bound (4.2.2)

$$|\hat{M}_{1:n} - M_{1:n}| \leq 2\varepsilon \sum_{j=1}^{n-1} |M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:n}|$$

to the singular values of the symmetric tridiagonal matrix T, i.e. the case of $r = n$ in (4.2.1). The similar discussion can also be applied to $1 \leq r < n$.

There is an explicit formula for the matrix product $M_{i:j} = M_i M_{i+1} \cdots M_j$ [74]:

$$M_{i:j} = \begin{bmatrix} \det(T(i:j)) & \det(T(i:j-1)) \\ -b_{i-1}^2 \det(T(i+1:j)) & -b_{i-1}^2 \det(T(i+1:j-1)) \end{bmatrix}.$$

Therefore, we can express the following products explicitly:

$$\begin{aligned} |M_{1:i}| &= \begin{bmatrix} |\det(T(1:i))| & |\det(T(1:i-1))| \\ 0 & 0 \end{bmatrix}, \\ |M_{i+1:j}| &= \begin{bmatrix} |\det(T(i+1:j))| & |\det(T(i+1:j-1))| \\ b_i^2|\det(T(i+2:j))| & b_i^2|\det(T(i+2:j-1))| \end{bmatrix}, \\ |M_{j+1:k}| &= \begin{bmatrix} |\det(T(j+1:k))| & |\det(T(j+1:k-1))| \\ b_j^2|\det(T(j+2:k))| & b_j^2|\det(T(j+2:k-1))| \end{bmatrix}, \\ |M_{k+1:n}| &= \begin{bmatrix} |\det(T(k+1:n))| & |\det(T(k+1:n-1))| \\ b_k^2|\det(T(k+2:n))| & b_k^2|\det(T(k+2:n-1))| \end{bmatrix}. \end{aligned}$$

So

$$|M_{1:i}||M_{i+1:j}| = \left[\begin{matrix} |\det(T(1:i))\det(T(i+1:j))| + b_i^2|\det(T(1:i-1))\det(T(i+2:j))| \\ 0 \end{matrix}\right.$$

$$\left.\begin{matrix} |\det(T(1:i))\det(T(i+1:j-1))| + b_i^2|\det(T(1:i-1))\det(T(i+2:j-1))| \\ 0 \end{matrix}\right].$$

and $|M_{j+1:k}||M_{k+1:n}| =$

$$\left[\begin{matrix} |\det(T(j+1:k))\det(T(k+1:n))| + b_k^2|\det(T(j+1:k-1))\det(T(k+2:n))| \\ b_j^2|\det(T(j+2:k))\det(T(k+1:n))| + b_j^2 b_k^2|\det(T(j+2:k-1))\det(T(k+2:n))| \end{matrix}\right.$$

$$\left.\begin{matrix} |\det(T(j{+}1{:}k))\det(T(k{+}1{:}n{-}1))| + b_k^2|\det(T(j{+}1{:}k{-}1))\det(T(k{+}2{:}n{-}1))| \\ b_j^2|\det(T(j{+}2{:}k))\det(T(k{+}1{:}n{-}1))| + b_j^2 b_k^2|\det(T(j{+}2{:}k{-}1))\det(T(k{+}2{:}n{-}1))| \end{matrix}\right].$$

Let $P = |M_{1:i}||M_{i+1:j}||M_{j+1:k}||M_{k+1:n}|$, and let P_{11} be the $(1,1)$ element of P, which is the inner product of the first row of of $|M_{1:i}||M_{i+1:j}|$ and the first column of $|M_{j+1:k}||M_{k+1:n}|$. We denote the first row of $|M_{1:i}||M_{i+1:j}|$ by u^T and first column of $|M_{j+1:k}||M_{k+1:n}|$ by v.

By Lemma 4.4.5,

$$|\det(T(1:i-1))| \leq |\det(T(1:i))| / \sigma_{\min}(T(1:i)).$$

and

$$|\det(T(i+2:j))| \leq |\det(T(i+1:j))|/\sigma_{\min}(T(i+1:j)).$$

Therefore, we can bound u by

$$u^T \leq \{(1+\frac{b_i^2}{\sigma_{\min}(T(1:i))\sigma_{\min}(T(i+1:j))})|\det(T(1:i))\det(T(i+1:j))|,$$

$$(1+\frac{b_i^2}{\sigma_{\min}(T(1:i))\sigma_{\min}(T(i+1:j-1))})|\det(T(1:i))\det(T(i+1:j-1))|\},$$

and bound v by

$$v^T \leq \{(1+\frac{b_k^2}{\sigma_{\min}(T(j+1:k))\sigma_{\min}(T(k+1:n))})|\det(T(j+1:k))\det(T(k+1:n))|,$$

$$b_j^2(1+\frac{b_k^2}{\sigma_{\min}(T(j+2:k))\sigma_{\min}(T(k+1:n))})|\det(T(j+2:k))\det(T(k+1:n))|\}.$$

Therefore,

$$\begin{aligned}
P_{11} &= u^T \cdot v \\
&= (1+\frac{b_i^2}{\sigma_{\min}(T(1:i))\sigma_{\min}(T(i+1:j))})(1+\frac{b_k^2}{\sigma_{\min}(T(j+1:k))\sigma_{\min}(T(k+1:n))}) \times \\
&\quad |\det(T(1:i))\det(T(i+1:j))\det(T(j+1:k))\det(T(k+1:n))| \\
&+ b_j^2(1+\frac{b_i^2}{\sigma_{\min}(T(1:i))\sigma_{\min}(T(i+1:j-1))}) \times \\
&\quad (1+\frac{b_k^2}{\sigma_{\min}(T(j+2:k))\sigma_{\min}(T(k+1:n))}) \times \\
&\quad |\det(T(1:i))\det(T(i+1:j-1))\det(T(j+2:k))\det(T(k+1:n))|.
\end{aligned}$$

Again by Lemma 4.4.5,

$$|\det(T(i+1:j-1))| \leq \frac{|\det(T(i+1:j))|}{\sigma_{\min}(T(i+1:j))}$$

and

$$|\det(T(j+2:k)| \leq \frac{|\det(T(j+1:k))|}{\sigma_{\min}(T(j+1:k))},$$

$$\begin{aligned}
P_{11} &\leq |\det(T[i,j,k])| \times \\
&\quad \{(1+\frac{b_i^2}{\sigma_{\min}(T(1:i))\sigma_{\min}(T(i+1:j))})(1+\frac{b_k^2}{\sigma_{\min}(T(j+1:k))\sigma_{\min}(T(k+1:n))}) \\
&+ \frac{b_j^2}{\sigma_{\min}(T(i+1:j))\sigma_{\min}(T(j+1:k))}(1+\frac{b_i^2}{\sigma_{\min}(T(1:i))\sigma_{\min}(T(i+1:j-1))}) \times \\
&\quad (1+\frac{b_k^2}{\sigma_{\min}(T(j+2:k))\sigma_{\min}(T(k+1:n))})\} \\
&\equiv |\det(T[i,j,k])| \cdot F(n,i,j,k,T).
\end{aligned}$$

By Lemma 4.4.3, we can bound $|\det(T[i,j,k])|$ by

$$|\det(T[i,j,k])| \leq \frac{\sigma_n^6(T)}{\sigma_1(T)\ldots\sigma_6(T)}|\det(T)|.$$

Therefore, we can bound the forward error of parallel prefix as follows,

$$\frac{|p_n - \hat{p}_n|}{|p_n|} \leq 2\varepsilon \frac{\sigma_n^6(T)}{\sigma_1(T)\ldots\sigma_6(T)} \cdot \sum_{j=1}^{n-1} F(n,i,j,k,T).$$

In general,

$$\frac{|p_r - \hat{p}_r|}{|p_r|} \leq 2\varepsilon \frac{\sigma_r^6(T(1:r))}{\sigma_1(T(1:r))\ldots\sigma_6(T(1:r))} \cdot \sum_{j=1}^{n-1} F(r,i,j,k,T(1:r)).$$

Theorem 4.5.1 *Let T be a symmetric tridiagonal matrix, let $\hat{p}_r$ be the computed Sturm sequence by parallel prefix. Then*

$$\frac{|p_r - \hat{p}_r|}{|p_r|} \leq 2\varepsilon \frac{\sigma_r^6(T(1:r))}{\sigma_1(T(1:r))\ldots\sigma_6(T(1:r))} \cdot \sum_{j=1}^{n-1} F(r,i,j,k,T(1:r)).$$

In particular,

$$\frac{|p_n - \hat{p}_n|}{|p_n|} \leq 2\varepsilon \frac{\sigma_n^6(T)}{\sigma_1(T)\ldots\sigma_6(T)} \cdot \sum_{j=1}^{n-1} F(n,i,j,k,T).$$

Corollary 4.5.1 *Let T be a symmetric tridiagonal matrix, let $\hat{p}_n$ be the computed Sturm sequence by parallel prefix. Then*

$$\frac{|p_n - \hat{p}_n|}{|p_n|} \leq 2\varepsilon\kappa^6 \cdot \sum_{j=1}^{n-1} F(n,i,j,k,T).$$

where κ is the condition number of the matrix T.

Proof.

$$\begin{aligned}\frac{|p_n - \hat{p}_n|}{|p_n|} &\leq 2\varepsilon \frac{\sigma_n^6(T)}{\sigma_1(T)\ldots\sigma_6(T)} \cdot \sum_{j=1}^{n-1} F(n,i,j,k,T) \\ &\leq 2\varepsilon \frac{\sigma_n^6(T)}{\sigma_1^6(T)} \cdot \sum_{j=1}^{n-1} F(n,i,j,k,T) \\ &= 2\varepsilon\kappa^6 \cdot \sum_{j=1}^{n-1} F(n,i,j,k,T).\end{aligned}$$

∎

The above corollary leads to the following *conjecture*:

Conjecture: For general symmetric tridiagonal matrices, the forward relative error for the computed Sturm sequence by parallel prefix algorithm can be as large as $O(\varepsilon\kappa^6)$, where κ is the condition number of matrix T.

We have done many numerical experiments to search for an example such that the error bound $O(\varepsilon\kappa^6)$ in above conjecture can be attained, but we haven't found one yet. How large the forward error can be for general symmetric tridiagonal matrix still remains as an open problem.

4.6 Computing the Sturm Sequence is Equivalent to Solving A Linear System of Equations

In this section, we prove that computing a Sturm sequence is equivalent to solving a unit lower triangular banded linear system of equations. Let $\hat{p}_i$ and p_i be the computed and exact Sturm sequence, respectively. Let δ_i be the difference of p_i and $\hat{p}_i$, i.e. $\delta_i = p_i - \hat{p}_i$. Since

$$p_i = a_i p_{i-1} - b_{i-1}^2 p_{i-2},$$

we can write

$$\begin{aligned} \hat{p}_i + \delta_i &= a_i(\hat{p}_{i-1} + \delta_{i-1}) - b_{i-1}^2(\hat{p}_{i-2} + \delta_{i-2}) \\ &= a_i\hat{p}_{i-1} - b_{i-1}^2\hat{p}_{i-2} + a_i\delta_{i-1} - b_{i-1}^2\delta_{i-2}. \end{aligned}$$

Let $P_i = a_i\hat{p}_{i-1} - b_{i-1}^2\hat{p}_{i-2}$, so

$$\hat{p}_i + \delta_i = P_i + a_i\delta_{i-1} - b_{i-1}^2\delta_{i-2}.$$

Equivalently,

$$b_{i-1}^2\delta_{i-2} - a_i\delta_{i-1} + \delta_i = -(\hat{p}_i - P_i).$$

Notice the fact that $p_0 = 1 = \hat{p}_0$ and $p_1 = a_1 = \hat{p}_1$, hence, $\delta_0 = \delta_1 = 0$.

To solve for δ_i for $i = 2, 3, \ldots, n$, all we need to do is to solve the following lower triangular linear system of equations:

$$\begin{bmatrix} 1 & & & & & & \\ -a_3 & 1 & & & & & \\ b_3^2 & -a_4 & 1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & b_{i-1}^2 & -a_i & 1 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & b_{n-1}^2 & -a_n & 1 \end{bmatrix} \begin{bmatrix} \delta_2 \\ \delta_3 \\ \delta_4 \\ \vdots \\ \delta_i \\ \vdots \\ \delta_n \end{bmatrix} = - \begin{bmatrix} \hat{p}_2 - P_2 \\ \hat{p}_3 - P_3 \\ \hat{p}_4 - P_4 \\ \vdots \\ \hat{p}_i - P_i \\ \vdots \\ \hat{p}_n - P_n \end{bmatrix}. \tag{4.6.6}$$

Denote the coefficient matrix in (4.6.6) by L, and let

$$\hat{p}_i = 0 \quad i = 2, 3, \ldots, n,$$

so that $\delta_i = p_i$ for $i \geq 2$. Therefore, computing the Sturm sequence is equivalent to solving the following linear system of equations:

$$L \cdot d \equiv L \cdot \begin{bmatrix} p_2 \\ p_3 \\ p_4 \\ \vdots \\ p_i \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} a_2 a_1 - b_1^2 \\ -b_2^2 a_1 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix} \equiv e. \tag{4.6.7}$$

There is no need to solve for p_1 since $p_1 = a_1$. We summarize our derivation in the following theorem.

Theorem 4.6.1 *In exact arithmetic, the solution components p_i for the lower triangular linear system of equations (4.6.7) form the Sturm sequence of T for $i = 2, \ldots, n$. Therefore solving the linear system (4.6.7) is equivalent to computing the Sturm sequence.*

Because of the above theorem, we can compute the Sturm sequence in parallel in an alternative way: parallel linear equation solver plus iterative refinement.

Algorithm 4.6.1 *Compute Sturm sequence by parallel linear system solver and iterative refinement.*

Solve lower triangular equation $L \cdot p = e$ *using some parallel method.*
Compute the residual $r = e - L \cdot \hat{p}$ *where* $\hat{p}$ *is the computed solution.*
while *the residual* r *is above some tolerance* τ, do
    Solve $L \cdot f = r$ *by the same parallel method.*
    Update the solution $\hat{p}$ *by* $\hat{p} = \hat{p} + f$.
    Compute the residual $r = e - L \cdot \hat{p}$ *where* $\hat{p}$ *is the updated solution.*
end

Several alternatives to the standard substitution algorithm for solving triangular linear systems have been proposed for parallel computation and several parallel implementations have been developed [54, 71, 82]. In [58], four parallel triangular linear system solvers and their stabilities have been discussed: Fan-In algorithm, Block Elimination, Power Series and Matrix Inversion by Divide-and-Conquer. Based on some results from [58], we will analyze the errors of Algorithm 4.6.1 by using these four parallel triangular solvers. Our analysis will be in both conventional norm accuracy style and in Skeel's componentwise accuracy style [86, 87].

4.7 Four Parallel Triangular Equation Solvers

In this section, we introduce the four parallel triangular linear equation solvers whose stablity are discussed in [58]: Fan-In algorithm, Block Elimination, Power Series and Matrix Inversion by Divide-and-Conquer.

4.7.1 Fan-In Algorithm

Any unit $n \times n$ lower triangular matrix L can be factorized $L = L_1 L_2 \cdots L_n$, where L_i equals the identity matrix except for colum i where it matches L:

$$L_i = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1 & & & \\ & & & l_{i+1,i} & 1 & & \\ & & & \vdots & & \ddots & \\ & & & l_{n,i} & & & 1 \end{bmatrix}.$$

Therefore, the solution to a linear system $Lx = b$ can be solved as follows [60]:

$$x = L^{-1}b = W_n W_{n-1} \cdots W_1 b, \tag{4.7.8}$$

where

$$W_i = L_i^{-1} = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & 1/l_{ii} & & & \\ & & & -l_{i+1,i}/l_{ii} & 1 & & \\ & & & \vdots & & \ddots & \\ & & & -l_{n,i}/l_{ii} & & & 1 \end{bmatrix},$$

i.e. L_i^{-1} equals the identity matrix except for the column i, where the diagonal element is $1/l_{ii}$ and the subdiagonals are the negative of L_i divided by l_{ii}.

The Fan-In algorithm solves $Lx = b$ by computing the product (4.7.8) in $O(\log_2 n)$ steps by the fan-in operation. For example, when $n = 7$, x can be computed as follows:

$$x = ((W_7 W_6)(W_5 W_4))((W_3 W_2)(W_1 b)).$$

The computation takes $\log_2 n$ parallel steps using a tree, where each parallel step involves multiplying $n \times n$ matrices, and so takes about $\log_2 n$ parallel substeps, for a total of $\log_2^2 n$ steps. More precisely, it can be implemented in $\frac{1}{2}\log_2^2 n + \frac{3}{2}\log_2 n + 3$ steps on $\frac{1}{68}n^3 + O(n^2)$ processors [85]. The error analysis [85] yields an error bound proportional to $\varepsilon\kappa(L)^3$ where

$\kappa(L)$ is the condition number; in contrast to the error bound $\varepsilon\kappa(T)$ for the conventional substitution algorithm. The error bound may be pessimistic, but an example can be found which has an error growing like $\varepsilon\kappa(L)^{1.5}$ [32]. Also, the requirement of $O(n^3)$ processors to achieve the maximum speedup is unrealistic for large n.

4.7.2 Block Elimination Algorithm

In addition to the fan-in algorithm, Sameh and Brent [85] introduce a parallel block elimination algorithm. It requires the same time as the fan-in algorithm but roughly twice the number of processors. The advantage of this algorithm is that it can be adapted to band structure [20].

Let $L^{(0)} = DL$ and $b^{(0)} = Db$ where $D = \text{diag}\ (l_{11}^{-1}, l_{22}^{-1}, \ldots, l_{nn}^{-1})$. We form matrices $D^{(j)}$, $j = 0, 1, \ldots, m-1$, such that if

$$L^{(j+1)} = D^{(j)}L^{(j)} \quad \text{and} \quad b^{(j+1)} = D^{(j)}b^{(j)},$$

then $L^{(m)} = I$ and $x = b^{(m)} = L^{-1}b$. For example, when $n = 8$,

$$\begin{aligned}
L^{(1)} &= D^{(0)}L^{(0)} \\
&= \text{diag}\ (\begin{bmatrix} 1 & \\ -l_{21} & 1 \end{bmatrix}, \begin{bmatrix} 1 & \\ -l_{43} & 1 \end{bmatrix}, \begin{bmatrix} 1 & \\ -l_{65} & 1 \end{bmatrix}, \begin{bmatrix} 1 & \\ -l_{87} & 1 \end{bmatrix})L^{(0)} \\
&= \begin{bmatrix} I_2 & & & \\ L_{21}^{(1)} & I_2 & & \\ L_{31}^{(1)} & L_{32}^{(1)} & I_2 & \\ L_{41}^{(1)} & L_{42}^{(1)} & L_{43}^{(1)} & I_2 \end{bmatrix}, \\
b^{(1)} &= D^{(0)}b^{(0)}, \\
L^{(2)} &= D^{(1)}L^{(1)} = \text{diag}\ (\begin{bmatrix} I_2 & \\ -L_{21}^{(1)} & I_2 \end{bmatrix}, \begin{bmatrix} I_2 & \\ -L_{43}^{(1)} & I_2 \end{bmatrix})L^{(1)} = \begin{bmatrix} I_4 & \\ L_{21}^{(2)} & I_4 \end{bmatrix}, \\
b^{(2)} &= D^{(1)}b^{(1)},
\end{aligned}$$

and finally ,

$$\begin{aligned}
L^{(3)} &= D^{(2)}L^{(2)} = \begin{bmatrix} I_4 & \\ -L_{21}^{(2)} & I_4 \end{bmatrix} L^{(2)} = I, \\
x &= b^{(3)} = D^{(2)}b^{(2)} = \begin{bmatrix} I_4 & \\ -L_{21}^{(2)} & I_4 \end{bmatrix} b^{(2)}.
\end{aligned}$$

Thus in $\log_2 n$ steps L is reduced to the identity matrix and b is transformed to $x = L^{-1}b$. If the $n \times n$ lower triangular matrix L has bandwidth $m+1$, then $Lx = b$ can be solved in $(2 + \log_2 m)\log_2 n - \frac{1}{2}(\log_2^2 m + \log_2 m) + 3$ steps using no more than $\frac{1}{2}m^2 n + O(mn)$ processors.

4.7.3 Power Series Method

The following method has been discussed by Heller [55] and Orcutt [76]. Let $L = D(I - N)$ be the $n \times n$ lower triangular matrix, where $n = 2^k$, $D = \text{diag } (L)$ and N is the strictly lower triangular part. Then

$$\begin{aligned} x &= L^{-1}b = (I - N)^{-1}D^{-1}b \\ &= (I + N + \cdots + N^{n-1})D^{-1}b \\ &= (I + N^{2^{k-1}})(I + N^{2^{k-2}}) \cdots (I + N)D^{-1}b. \end{aligned}$$

We use the fact that $N^n = 0$ in the above equation. The powers $M^2, M^4, \ldots, M^{2^{k-1}}$, are formed by repeated squaring. This method can be implemented in $\log_2^2 n + \log_2 n$ steps on $n^3 + n^2$ processors [56].

4.7.4 Matrix Inversion by Divide and Conquer

Borodin and Munro [15] and Heller [56] discuss the following method for inverting a triangular matrix based on the divide and conquer technique:

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix}, \quad W = L^{-1} = \begin{bmatrix} L_{11}^{-1} & 0 \\ -L_{22}^{-1}L_{21}L_{11}^{-1} & L_{22}^{-1} \end{bmatrix}.$$

The size of the diagonal blocks L_{11} and L_{22} are about same, the inversion L_{11}^{-1} and L_{22}^{-1} are computed by the same method recursively. This method can be implemented in $O(\log_2 n)$ steps in $O(n^3)$ processors.

4.8 Conventional Error Analysis

For the four parallel triangular solvers we introduced in last section section, Higham proved that they all satisfy a universal forward error bound [58]:

$$|L\hat{x} - b| \leq c_n \varepsilon |L| M(L)^{-1} |b| + O(\varepsilon^2). \tag{4.8.9}$$

where

$$M(A) = (m_{ij}) = \begin{cases} |a_{ii}| & \\ -|a_{ij}| & i \neq j \end{cases},$$

c_n is a lower order function of n and $\hat{x}$ is the computed solution. Let

$$(L + \triangle L)\hat{x} = b.$$

where $\triangle L$ is the backward error for L, and define

$$\eta = \inf\{\varepsilon : (L + \triangle L)\hat{x} = b, \ \ \|\triangle L\| \leq \varepsilon \|A\|\}.$$

where $\|\cdot\|$ is $\|\cdot\|_\infty$. It can be shown that

$$\eta = \frac{\|L\hat{x} - b\|}{\|L\|\|\hat{x}\|}.$$

Since

$$\begin{aligned} \frac{\|\triangle L\|_\infty}{\|L\|_\infty} &\leq \eta = \frac{\|L\hat{x} - b\|_\infty}{\|L\|_\infty \|\hat{x}\|_\infty} \leq \frac{\varepsilon c_n \|L\|_\infty \|M(L)^{-1}\|_\infty \|b\|_\infty}{\|L\|_\infty \|\hat{x}\|_\infty} \\ &\leq \frac{\varepsilon c_n \|L\|_\infty \|M(L)^{-1}\|_\infty \|L\|_\infty \|\hat{x}\|_\infty}{\|L\|_\infty \|\hat{x}\|_\infty}, \end{aligned}$$

therefore,

$$\|\triangle L\|_\infty \leq \varepsilon \gamma \|L\|_\infty.$$

where $\gamma \leq c_n \|L\|_\infty \|M(L)^{-1}\|_\infty$. For different algorithms, we will have different γ's. Now we show that if γ satisfies some constraints, the iterative refinement in Algorithm 4.6.1 will converge.

Theorem 4.8.1 *If we compute the residual r of Algorithm 4.6.1 by double precision, and $\varepsilon \kappa_\infty(L)(\gamma + 1) \leq c < 1$, then the iterative refinement will converge.*

Proof. Let

$$r = fl(L\hat{x}_i - b) = L\hat{x}_i - b + f$$

where $|f| \leq n\varepsilon^2(|L||\hat{x}_i| + |b|) + \varepsilon|L\hat{x}_i - b| \approx \varepsilon|L\hat{x}_i - b|$, because r is computed in double precision. From previous discussion, we know that the backward error $\triangle L$ such that

$$(L + \triangle L)d = r$$

must satisfy $\|\triangle L\|_\infty \leq \varepsilon \gamma \|L\|_\infty$.

Assume that $\hat{x}_{i+1} = \hat{x}_i - d$ be computed exactly, then

$$\begin{aligned} d &= (L + \triangle L)^{-1} r = (I + L^{-1}\triangle L)^{-1} L^{-1} r \\ &= (I + L^{-1}\triangle L)^{-1} L^{-1}(L\hat{x}_i - b + f) = (I + L^{-1}\triangle L)^{-1}(\hat{x}_i - x + L^{-1} f) \\ &\approx (I - L^{-1}\triangle L)(\hat{x}_i - x + L^{-1} f) \\ &\approx \hat{x}_i - x - L^{-1}\triangle L(\hat{x}_i - x) + L^{-1} f \end{aligned}$$

Therefore,

$$\hat{x}_{i+1} - x = \hat{x}_i - d - x = L^{-1}\triangle L(\hat{x}_i - x) - L^{-1} f.$$

and (the following $\|\cdot\|$ is $\|\cdot\|_\infty$)

$$\begin{aligned} \|\hat{x}_{i+1} - x\|_\infty &\leq \|L^{-1}\|\|\triangle L\|\|\hat{x}_i - x\| + \|L^{-1}\|\varepsilon\|L\hat{x}_i - b\| \\ &\leq \|L^{-1}\|\|\triangle L\|\|\hat{x}_i - x\| + \varepsilon\|L^{-1}\|\|L(\hat{x}_i - x)\| \\ &\leq \gamma\varepsilon\|L^{-1}\|\|L\|\|\hat{x}_i - x\| + \varepsilon\|L^{-1}\|\|L\|\|\hat{x}_i - x\| \\ &\leq \varepsilon(\gamma + 1)\kappa_\infty(L)\|\hat{x}_i - x\|_\infty \\ &\leq c\|\hat{x}_i - x\|_\infty \end{aligned}$$

Since $c < 1$, we know that the iterative refinement converges. ∎

By applying the above theorem, we will show under what circumstances Algorithm 4.6.1 will converge when we use the four parallel triangular solvers to solve the linear equations.

i. **Fan-In Algorithm:** In [85] it was shown when we use fan-in algorithm to solve the triangular system, the backward error $\triangle L$ satisfies:

$$\|\triangle L\|_\infty \leq \alpha_n \varepsilon \kappa_\infty(L)^2 \|L\|_\infty.$$

where $\alpha_n = n^2 \log n/4 + O(nlogn)$. From a forward bound in [58]:

$$|L\hat{x} - b| \leq d_n \varepsilon |L||L^{-1}||L||L^{-1}||L||x|,$$

an improved backward error bound can be obtained:

$$\|\triangle L\|_\infty \leq \frac{\|L\hat{x} - b\|_\infty}{\|\hat{x}\|_\infty} \leq d_n \varepsilon \|L\|_\infty^2 \|L^{-1}\|_\infty^2 \|L\|_\infty = d_n \varepsilon \kappa_\infty^2(L) \|L\|_\infty.$$

where $d_n = an\log n$, $a = O(1)$. Therefore, $\gamma_{Fan-In} = d_n \kappa_\infty^2(L)$. By Theorem 4.8.1, to guarantee the convergence of iterative refinement, we have to make the following assumption:

$$d_n \varepsilon \kappa_\infty^3(L) < 1.$$

ii. **Block Elimination:** The error bound is about the same as universal bound (4.8.9), so it is not necessary to discuss further details.

iii. **Power Series:** As with Block Elimination, the analysis of this algorithm adds nothing new.

iv. **Matrix Inversion by Divide-and-Conquer:** This algorithm has the best backward error bound of the four triangular solvers, let $\hat{X}$ be the computed inversion of L, then [58]:

$$|L\hat{X} - I| \le c_n\varepsilon|L||\hat{X}|.$$

and

$$\hat{x} = \hat{X}b + f.$$

where f is the roundoff error, $f = O(\varepsilon)$. Therefore,

$$|L\hat{x} - b| \le 2c_n\varepsilon|L||\hat{X}||b|,$$

and

$$\begin{aligned}\|\triangle L\|_\infty &\le \frac{\|L\hat{x} - b\|_\infty}{\|\hat{x}\|_\infty} \\ &\le 2c_n\varepsilon\|L\|_\infty\|\hat{X}\|_\infty\|L\|_\infty \\ &\approx 2c_n\varepsilon\kappa_\infty(L)\|L\|_\infty\end{aligned}$$

Therefore,

$$\gamma_{Divide-\&-Conquer} = 2c_n\kappa_\infty(L),$$

which is the best that we can expect. To achieve the convergence of the iterative refinement, we need

$$2c_n\varepsilon\kappa_\infty^2(L) < 1 \text{ i.e. } \varepsilon\kappa_\infty^2(L) = O(1).$$

The conventional error analysis can only give us the normwise error bound of the backward error $\triangle L$. However, when we solve the linear system (4.6.7), we want the componentwise error bound of the solution vector p, so that we can estimate the relative error of the Sturm sequence p_i. The conventional normwise error clearly won't achieve this. Indeed, We can scale T to be a bit less than 1 in norm, which guarantees $\kappa(L) = O(1)$. But then the vector of p_i is strongly graded, and norm error bounds are irrelevant. Therefore, in next section, we will discuss the componentwise error analysis of Algorithm 4.6.1.

4.9 Componentwise Error Bound

In this section, we first introduce a universal result which is true for any linear system solver [57], and then apply this universal result to the four triangular solvers we have discussed. For an approximate solution $\hat{x}$ to linear system $Ax = b$, the componentwise backward error is defined by:

$$\omega = \min\{\varepsilon : (A + \triangle A)y = b + \triangle b, \quad |\triangle A| \leq \varepsilon |A|, |\triangle b| \leq \varepsilon |b|\}.$$

where $|A|_{ij} = |A_{ij}|$ and $|x|_i = |x_i|$. In [75], Oettli and Prager proved that

$$\omega = \max_i \frac{|b - Ay|_i}{(|A||y| + |b|)_i}.$$

In last section, we showed the error analysis for the iterative refinement when the residual is computed in double precision. If we are willing to compute the residual in single precision, Theorem 4.8.1 does not hold anymore. A lot of error analysis has been done for the iterative refinement when the residual is computed in single precision [7, 87, 57], it can be proved that under certain conditions, iterative refinement converges after one update [87]. Here we give a short review of the analysis in [57]. To do this, we need to introduce the measure of ill-scaling of the vector $|A||x|$ [86]:

$$\sigma(A, x) = \frac{\max_i (|A||x|)_i}{\min_i (|A||x|)_i}.$$

We assume the computed solution $\hat{x}$ of linear system $Ax = b$ satisfies

$$|b - A\hat{x}| \leq \varepsilon [g(A, b)|\hat{x}| + h(A, b)], \tag{4.9.10}$$

and when compute the residual $r = b - A\hat{x}$, the computed residual $\hat{r}$ satisfies

$$|\hat{r} - r| \leq \varepsilon \cdot t(A, b, \hat{x}). \tag{4.9.11}$$

If we use SAXPY [70], to compute r, we can take

$$t(A, b, \hat{x}) = \frac{\nu_{n+1}}{\varepsilon}(|A||\hat{x}| + |b|). \tag{4.9.12}$$

where $\nu_{n+1} = (n + 1)\varepsilon/(1 - (n + 1)\varepsilon)$.

Theorem 4.9.1 (Higham [57]) *Suppose we solve the linear system $Ax = b$ with one step of iterative refinement. Assume the computed solution $\hat{x}$ satisfies (4.9.10) and the computed* *residual $\hat{r}$ satisfies (4.9.11), and $t(A,b,\hat{x})$ satisfies (4.9.12). If there exist two matrices G and H whose entries are nonnegative, such that $g(A,b) = G|A|$ and $h(A,b) = H|b|$, and*

$$\varepsilon^2(h(\|G\|_\infty + n + 1) + 2(\|G\|_\infty + n + 2)^2(1 + \varepsilon\|H\|_\infty)^2 cond(A^{-1})) \le \nu_{n+1},$$

where $cond(A^{-1}) = \||A||A^{-1}|\|_\infty$, then the refined solution $\hat{y}$ will satisfy:

$$|b - A\hat{y}| \le 2\nu_{n+1}|A||\hat{y}|.$$

Corollary 4.9.1 (Higham [57]) *Under the assumptions of Theorem 4.9.1, if $H = 0$, then if*

$$2\varepsilon^2\sigma(A,\hat{y})(\|G\|_\infty + n + 2)^2 cond(A^{-1}) \le \nu_{n+1}, \tag{4.9.13}$$

the refined solution $\hat{y}$ will satisfy:

$$|b - A\hat{y}| \le 2\nu_{n+1}|A||\hat{y}| = O(\varepsilon)|A||\hat{y}|.$$

Since $\nu_{n+1} = (n+1)\varepsilon/(1-(n+1)\varepsilon) \sim (n+1)\varepsilon$, (4.9.13) can be simplied to:

$$\varepsilon \cdot \sigma(A,\hat{y}) \cdot 2\frac{(\|G\|_\infty + n + 2)^2}{n+1} \cdot cond(A^{-1}) \le 1. \tag{4.9.14}$$

Based on the Corollary 4.9.1, we can apply the general error analysis to the four parallel triangular solvers.

i. **Fan-In Algorithm:** The computed solution $\hat{x}$ of $Lx = b$ solved by fan-in algorithm satisfies [57]

$$|L\hat{x} - b| \le d_n\varepsilon|L||L^{-1}||L||L^{-1}||L||\hat{x}|.$$

So $g(L,b) = (|L||L^{-1}|)^2|L|$, therefore,

$$G = (|L||L^{-1}|)^2 \text{ and } g = \|G\|_\infty = \||L||L^{-1}|\|_\infty^2.$$

Because of (4.9.14), to achieve the componentwise accuracy of the corrected solution $\hat{y}$, approximately, the corrected solution $\hat{y}$ has to satisfy

$$\varepsilon \cdot c_n \cdot \sigma(L,\hat{x})\||L||L^{-1}|\|_\infty^5 = O(1).$$

ii. **Block Elimination and Power Series:** As with the conventional error analysis presented earlier, we will not discuss the error analysis for these algorithms.

iii. **Matrix Inversion by Divide-&-Conquer:** Since the computed solution $\hat{x}$ satisfies [58]

$$|L\hat{x} - b| \leq 2c_n \varepsilon |L||\hat{X}||L|\hat{x}|,$$

where $\hat{X}$ is the computed L^{-1}, and

$$|L\hat{X} - I| \leq c_n \varepsilon |L||\hat{X}|.$$

It is easy to show that $\hat{x}$ satisfies

$$|L\hat{x} - b| \leq 2c_n \varepsilon |L||L^{-1}||L|\hat{x}|.$$

Therefore,

$$G = |L||L^{-1}| \text{ and } g = \||L||L^{-1}|\|_\infty.$$

To achieve the componentwise accuracy of the corrected solution $\hat{y}$, we need the following condition to be satisfied:

$$\varepsilon \cdot c_n \cdot \sigma(L, \hat{y}) \||L||L^{-1}|\|_\infty^3 = O(1).$$

Since the matrix L of the triangular system we want to solve is very special, in fact L has only two subdiagonals(see(4.6.7)), the measure of ill-scaling of the vector $|L||\hat{p}|$ is:

$$\begin{aligned} \sigma(L, \hat{p}) &= \frac{\max_i (|L||\hat{p}|)_i}{\min_i (|L||\hat{p}|)_i} \\ &= \frac{\max_i (|\hat{p}_i| + |a_i \hat{p}_{i-1}| + |b_{i-1}^2 \hat{p}_{i-2}|)}{\min_i (|\hat{p}_i| + |a_i \hat{p}_{i-1}| + |b_{i-1}^2 \hat{p}_{i-2}|)}. \end{aligned}$$

In particular, when the tridiagonal matrix T is a positive definite matrix,

$$\sigma(L, \hat{p}) \approx \frac{\max_i (a_i \det(T(1:i-1)))}{\min_i (a_i \det(T(1:i-1)))}.$$

Therefore, $\sigma(L, \hat{p})$ can be arbitrarily large.

From both the conventional and componentwise analysis, we conclude that when we use fast parallel triangular solvers like the fan-in algorithm, we can not compute the Sturm sequence with as much guaranteed accuracy as the serial algorithm, when the matrix is not very well-conditioned. However, inspired by the idea of iterative refinement to solve linear system of equations, we can apply this idea for parallel prefix algorithm which appears in next section.

4.10 Iterative Refinement for Parallel Prefix Algorithm

In section 4.6, we showed that if $\delta_i = p_i - \hat{p}_i$, where p_i and $\hat{p}_i$ are the exact and computed Sturm sequence components, then δ_i satisfies the following relation:

$$b_{i-1}^2 \delta_{i-2} - a_i \delta_{i-1} + \delta_i = -(\hat{p}_i - P_i).$$

Or

$$\delta_i = a_i \delta_{i-1} - b_{i-1}^2 \delta_{i-2} + r_i. \tag{4.10.15}$$

where $P_i = a_i \hat{p}_{i-1} - b_{i-1}^2 \hat{p}_{i-2}$ and $r_i = -(\hat{p}_i - P_i)$. Inspired by the idea of iterative refinement for solving linear system of equations, if we can solve equation (4.10.15) for the residual δ_i in parallel, and iteratively refine the computed Sturm sequence $\hat{p}_i$, then after converge, the computed Sturm sequence should be accurate enough for computing the count. In fact, equation (4.10.15) can be considered as nonhomogeneous three term linear recurrence, it can be solved by parallel prefix operation as follows.

$$[\delta_i,\ \delta_{i-1},\ 1] = [\delta_{i-1},\ \delta_{i-2},\ 1] \cdot \begin{bmatrix} a_i & 1 & 0 \\ -b_{i-1}^2 & 0 & 0 \\ r_i & 0 & 1 \end{bmatrix} = [\delta_{i-1},\ \delta_{i-2},\ 1] \cdot R_i \tag{4.10.16}$$

Therefore,

$$[\delta_i,\ \delta_{i-1},\ 1] = [\delta_0,\ \delta_{-1},\ 1] \cdot R_1 R_2 \cdots R_i. \tag{4.10.17}$$

Hence, we can solve for δ_i by parallel prefix as the way we solve for Sturm sequence, except here we use 3×3 matrix multiplication instead of 2×2. So we have the following parallel prefix + iterative refinement algorithm.

Algorithm 4.10.1 *Compute Sturm sequence by parallel prefix operation and iterative refinement.*

```
Compute Sturm sequence by 2 × 2 parallel prefix matrix multiplication.
/* denote the computed Sturm sequence by p̂ */
P_i = a_i p̂_{i-1} − b_{i-1}^2 p̂_{i-2}.
Compute residual r_i = P_i − p̂.
while the residual r_i is above some tolerance τ, do
        Solve δ_i by 3 × 3 parallel prefix matrix multiplication.
        Update the solution p̂ by p̂ = p̂ + δ_i.
        P_i = a_i p̂_{i-1} − b_{i-1}^2 p̂_{i-2}.
        Compute the residual r_i = P_i − p̂.
end
```

Now the question remains is whether Algorithm 4.10.1 will converge in constant steps or $o(n)$ steps, where n is the order of the symmetric tridiagonal matrix T.

We implemented Algorithm 4.10.1 in MATLAB, our numerical experiments show that for certain matrices, it takes $O(n)$ steps for the algorithm to converge, making the complexity of the parallel algorithm to $O(n \log_2 n)$, which is even larger than the serial algorithm.

Table 4.1 shows the sequence of the updated Sturm sequence for a glued 16×16 positive definite matrix, when we use parallel prefix to compute the count at $x = -2 \cdot 10^{-8}$, the computed count is 1 and the true count is 0. We denote the relative error of the Sturm sequence by

$$\varepsilon_i = \frac{|p_i - \hat{p}_i|}{|p_i|}.$$

Figure 4.5 plots the sequence of the updated Sturm sequence for a glued 64×64 positive definite matrix and the maximum relative error $\max_i \varepsilon_i$ at each iterate, it takes 63 steps to converge. We compute at $x = -2 \cdot 10^{-10}$, the computed count by parallel prefix is 7 and the true count is 0.

Iteration #	Computed Count	$\max_i \varepsilon_i$
0	1	5.2333e+01
1	1	5.2333e+01
2	1	4.8049e+06
3	3	8.7467e+11
4	5	4.2877e+17
5	5	1.7959e+18
6	5	2.6166e+18
7	4	6.2610e+22
8	4	2.5043e+23
9	3	3.7564e+23
10	3	3.7564e+23
11	2	2.5043e+23
12	1	6.2607e+22
13	1	1.0000e+00
14	0	0

Table 4.1: Iterative refinement of parallel prefix algorithm for 16×16 glued positive definite matrix

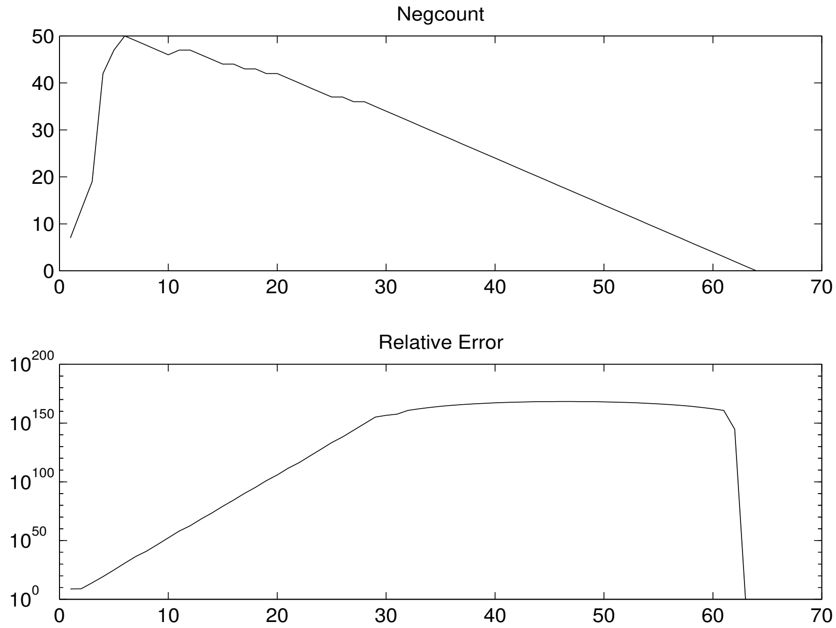


Figure 4.5: Iterative refinement of parallel prefix algorithm for 64×64 glued positive definite matrix

4.11 Criterion to Determine the Accuracy of Parallel Prefix

Parallel prefix algorithm can be very unstable when being used to compute clusters of eigenvalues, as we have shown earlier. However, it works well for a random matrix with entries independently and uniformly distributed in $[-1, 1]$ (see figure 3.9 and figure 4.4). It also works well for several other types of matrices [94]. If we can find a criterion which can estimate the error made by parallel prefix quite accurately, then we can use a general technique to take advantage of the superior speed of parallel prefix algorithm. The technique is used to deal with the tradeoff between parallelism and stability in order to make the whole algorithm go as fast as possible, but not sacrificing much accuracy [28]. It takes the following three steps:

- Compute the solution for the problem by fast but not very stable parallel algorithm.
- Quickly and reliably confirm or deny the accuracy of the computed solution.
- If computed solution is denied, recompute the counts by serial algorithm.

The key issue to apply this technique is to find a satisfactory criterion. The backward error

$$\eta = \max_i \eta_i = \max_i \frac{|\hat{p}_i - \hat{P}_i|}{|a_i||\hat{p}_{i-1}| + 2b_{i-1}^2|\hat{p}_{i-2}|}.$$

is a reasonable one, but from our numerical experiments, (figure 3.6—figure 3.9), it's difficult to decide what magnitude of η is large enough for us to deny the computed solution.

Another reasonable proposal is to use

$$\tau = \max_i \frac{P_i - \hat{p}_i}{\hat{p}_i},$$

but for the same reason as for η, τ can not provide enough information for us to decide when to reject the computed solution. Therefore, to find a satisfactory criterion remains as an open problem.

Chapter 5

Applying the Divide-and-Conquer Method to the Singular Value Decomposition and Least Squares Problem

5.1 Introduction

Computing the singular value decomposition (SVD) is one of the most important problems in numerical linear algebra. The SVD reveals a great deal about the structure of a matrix, e.g. its rank. The most reliable methods to solve a linear least squares problem are based on the SVD, in particular when the matrix is nearly rank-deficient [43, 2]. In this chapter, we discuss the divide-and-conquer method for computing the SVD. We compare the performance of the divide-and-conquer method with the method based on QR-iteration [41, 33]. We also compare the performance of various linear least squares solvers:

- `xGELS`, the fastest and least reliable method based on plain QR decomposition.
- `xGELSX`, based on QR with column pivoting, runs twice as slowly as `xGELS`[1].
- `xGELSY`, also based on QR with column pivoting but using BLAS 3, runs 1.3 times as slowly as `xGELS`.

[1]all the performance mentioned here is on an IBM RS6000/590 for 1600×1600 random matrices

- `xGELSA` and `xGELSB`, based on rank-revealing QR (RRQR), run 1.1 times as slowly as `xGELS`.
- `xGELSS`, based on SVD using QR-iteration, runs 96 times as slowly as `xGELS`.
- `xGELSF`, also based on SVD using QR-iteration but using "factored form", runs only 3.3 times as slowly as `xGELS`.
- `xGELSD`, based on divide-and-conquer SVD, runs 3.5 times as slowly as `xGELS`.

In section 5.6, we show that for 1600×1600 random matrices whose elements are independently and uniformly distributed, our implementation of divide-and-conquer method runs 50 times faster than the QR based SVD for bidiagonal matrices, and 13 times faster for dense matrices. The least squares solver based on divide-and-conquer SVD runs 28 times faster than `DGELSS`, the solver based on QR-iteration SVD.

5.2 Review of the SVD and Least Squares Problem

Given an $m \times n$ real matrix A, the **singular value decomposition (SVD)** of A is the factorization

$$A = U\Sigma V^T,$$

where

$$U = [u_1, u_2, \ldots, u_m] \in \mathcal{R}^{m\times m} \quad \text{and} \quad V = [v_1, v_2, \ldots, v_n] \in \mathcal{R}^{n\times n}$$

are orthogonal matrices, and

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r, 0, \ldots, 0) \in \mathcal{R}^{m\times n} \quad r = \min(m, n)$$

with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$.

The σ_i are called the *singular values* of A and for $i = 1, \ldots, r$, the vectors u_i and v_i are called the ith *left singular vector* and *right singular vector* respectively. It is easy to see that

$$Av_i = \sigma_i u_i \quad \text{and} \quad A^T u_i = \sigma v_i.$$

If A is complex, then its SVD is $A = U\Sigma V^H$ where U and V are unitary matrices, and Σ is a diagonal matrix with real nonnegative diagonal elements. From now on, we assume A is real.

One of the many important applications of the SVD is to find the solution of a (possibly) rank-deficient *linear least squares problem*:

$$\min_{x \in \mathcal{R}^n} \|Ax - b\|_2, \tag{5.2.1}$$

where $b \in \mathcal{R}^m$ is a given vector.

In the case $m > n$ and $\text{rank}(A) = n$, i.e., A has full column rank, the problem is referred to as finding a **least squares solution** to an **overdetermined** system of linear equations. It is well known that the solution to the problem is unique,

$$x_{LS} = V(\Sigma_1^{-1} \quad 0_{n\times(m-n)})U^T b,$$

where $\Sigma_1 \in \mathcal{R}^{n\times n}$ is the leading $n \times n$ submatrix of Σ.

In the case $m < n$ and $\text{rank}(A) = m$, there are an infinite number of solutions x which satisfy $Ax = b$. It is often useful to find the unique solution x which also minimizes $\|x\|_2$, and the problem is referred to as finding a **minimum norm solution** to an **underdetermined** system of linear equations. The solution can be expressed as:

$$x_{LS} = V \begin{bmatrix} \Sigma_2^{-1} \\ 0_{(n-m)\times m} \end{bmatrix} U^T b,$$

where $\Sigma_2 \in \mathcal{R}^m$ is the leading $m \times m$ submatrix of Σ.

In the general case when we may have $\text{rank}(A) < \min(m, n)$, which is called **rank-deficient** linear least squares problem, the solution x should minimize both $\|x\|_2$ and $\|b - Ax\|_2$.

The SVD of a dense matrix (we call it dense SVD) is usually computed in two stages [41]:

Stage 1 The matrix A is reduced to bidiagonal form:

$$A = UBV^T,$$

where U and V are orthogonal matrices and B is a bidiagonal matrix. B is upper bidiagonal when $m \geq n$ and lower bidiagonal when $m < n$, so that B is nonzero only on the main diagonal and either on the first superdiagonal (if $m \geq n$) or the first subdiagonal (if $m < n$):

$$B = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & \beta_{n-1} \\ & & & \alpha_n \end{bmatrix} \text{ when } m \geq n$$

or

$$B = \begin{bmatrix} \alpha_1 & & & \\ \beta_1 & \alpha_2 & & \\ & \ddots & \ddots & \\ & & \beta_{n-1} & \alpha_n \end{bmatrix} \text{ when } m < n$$

Stage 2 Compute the SVD of bidiagonal matrix B (we call this the bidiagonal SVD to differentiate it from the dense SVD):

$$B = Q\Sigma W^T,$$

where Q and W are orthogonal matrices and Σ is the diagonal matrix we described before.

The SVD of A is then computed as

$$A = (UQ)\Sigma(VW)^T.$$

Stage 2 has previously been implemented using QR-iteration [33, 41, 42] or QD-iteration (singular values only) [38, 83]. This has been the bottleneck of the whole algorithm, it can take as much as 95% of the total time when n is large (see table 5.10).

Without loss of generality, from now on, we will assume A is an $m \times n$ matrix with $m \geq n$, therefore, the bidiagonal matrix B will be upper bidiagonal.

We have implemented a divide-and-conquer algorithm to compute the bidiagonal SVD to overcome this bottleneck. We call it `xBDSDC` which follows the LAPACK naming convention [2]. Based on this, we implemented a dense SVD algorithm, called `xGESDD`, which uses `xBDSDC` SVD algorithm for Stage 2. All the implementations will appear in LAPACK Release 2.1. The bidiagonal divide-and-conquer algorithm which `xBDSDC` uses is a variation of the Gu and Eisenstat algorithm [51], which is based on previous work by Arbenz and Golub [6], Cuppen [22], Golub [45], Gu and Eisenstat [49], and Jessup and Sorensen [63], for computing the SVD of B. Our numerical experiments on an IBM RS6000/590 show that `DBDSDC` (double precision implementation) runs at least 47 times faster for $n = 1600$ than the LAPACK implementation `DBDSQR` [33, 3] of the traditional QR based algorithm (see section 5.6 for details on the RS6000/590 configuration). `DGESDD`, the implementation of dense SVD, is from 7.6 to 13.5 times faster than `DGESVD` (the corresponding LAPACK implementation [2] for computing the dense SVD) of A when $n = 1600$. We should mention that in general, for a bidiagonal matrix, `xBDSQR` computes the singular values to high relative accuracy, whereas `xBDSDC` can only guarantees the absolute accuracy.

`xBDSDC` uses a "factored form", which allows us to compute the SVD of B in $O(n^2)$ flops by representing Q and W as products of $O(\log_2 n)$ structured orthogonal matrices [48, 47]. Once A has been reduced to upper bidiagonal form (Stage 2), this new version of the bidiagonal SVD allows us to finish the rest of the computation for solving the dense linear least squares problem in $O(mn)$ flops. The implementation of this algorithm is called `xGELSD`. Since the cost of Stage 1 is about $4mn^2 - 4n^3/3$ flops [43], about twice the cost of computing a QR factorization on A, our result means that the flop count of the SVD based least squares solver is only about twice that of the QR based solver. The "factored form" version is also useful for the case where the least squares solution is subject to some simple constraints [43].

Demmel [29, 48] implemented a technique for representing the SVD using QR-iteration in factored form, originally suggested in [21], and also known to Rutishauser in the context of Jacobi's method [83]. The idea is to store all the Givens rotations produced during bidiagonal QR iteration and apply them directly to the solution vector, rather than accumulating them. This simple change to the current LAPACK routine `xGELSS` for solving the least squares problem with the SVD, also reduces the flop count to just twice that of QR decomposition, but at the cost of $O(n^2)$ storage. This new implementation is called `xGELSF`.

Based purely on operation counts, we expect either of our two least squares algorithms, `xGELSD` and `xGELSF` to take only about twice as long as the fastest method (QR decomposition). However, when using optimized ESSL BLAS on the RS6000, the new SVD based least squares solvers are about 2.9 to 3.6 times slower than QR decomposition for $n = 1600$, not twice as slow. This is because the QR decomposition can be reorganized to do almost all its floating point operations by calls to Level 3 BLAS [36], whereas Stage 1 of the SVD does half its flops in the Level 3 BLAS and half in Level 2 BLAS [37].

It may be possible to break the "BLAS 2" barrier in reduction to bidiagonal form by exploiting successive band reduction techniques proposed for the symmetric eigenproblem [14], but we have not yet pursued this.

The new routines we mentioned above are implemented by a group of people. In particular, `xBDSDC` is implemented by Ming Gu, the root finder is implemented by Rencang Li, `xGELSF` is implemented by James Demmel, `xGESDD` is implemented by author, and `xGELSD` is implemented by Ming Gu and author.

5.3 `xBDSDC` and "Factored Form"

`xBDSDC` recursively divides B into two subproblems as follows [48, 47]:

$$B = \begin{pmatrix} B_1 & 0 \\ \alpha_k e_k^T & \beta_k e_1^T \\ 0 & B_2 \end{pmatrix} , \tag{5.3.2}$$

where $B_1 \in \mathcal{R}^{(k-1)\times k}$ and $B_2 \in \mathcal{R}^{(n-k)\times(n-k)}$ are upper bidiagonal matrices, and e_j is the j-th column of an identity matrix with appropriate dimension. We take $k = \lfloor n/2 \rfloor$.

Remark 5.3.1 `xBDSDC` actually uses the dividing strategy used in [6]; the algorithm in [51] takes out a column (instead of a row) of B at a time.

Assume that we are given the SVDs of B_1 and B_2:

$$B_1 = Q_1 (D_1 \quad 0) W_1^T \quad \text{and} \quad B_2 = Q_2 D_2 W_2^T ,$$

where Q_i and W_i are orthogonal matrices of appropriate dimensions, and the D_i's are non-negative diagonal matrices. Let $(l_1^T \quad \lambda_1)$ be the last row of W_1, and let f_2^T be the first row of W_2. Plugging these into (5.3.2), we get

$$B = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} \begin{pmatrix} D_1 & 0 & 0 \\ \alpha_k l_1^T & \alpha_k \lambda_1 & \beta_k f_2^T \\ 0 & 0 & D_2 \end{pmatrix} \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix}^T . \tag{5.3.3}$$

Note that the middle matrix is quite simple in that its entries can be non-zero only on the diagonal and in the k-th row. We will discuss the computation of its SVD later in this section. Let $S\Sigma G^T$ be the SVD of the middle matrix. Plugging it into (5.3.3), we get the SVD of B as

$$B = Q\Sigma W^T,$$

with

$$Q = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} S \quad \text{and} \quad W = \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix} G\,.$$

To compute the SVDs of B_1 and B_2, this process can be recursively applied until the sizes of the subproblems are sufficiently small[2]. These small subproblems are then solved using a QR type algorithm (`xBDSQR` in LAPACK). There can be at most $O(\log_2 n)$ levels of recursion.

`xBDSDC` also has a recursion for computing just the singular values. Let f_1^T be the first row of W_1; let l_2^T be the last row of W_2; and let f^T and l^T be the first and last rows of W, respectively. Suppose that D_i, f_i, l_i, and λ_1 are given for $i = 1, 2$. Then we can compute Σ, f, and l by computing the SVD of the middle matrix in (5.3.3) as $S\Sigma G^T$, and computing

$$f^T = (f_1^T \quad 0)\,G \quad \text{and} \quad l^T = (0 \quad l_2^T)\,G\ .$$

The "factored form" version of bidiagonal divide-and-conquer is based on the singular value recursion. We store S and G for each subproblem in the recursion, and never explicitly form any Q and W at any level, except the bottom level where we use a QR type algorithm.

In order to compute the SVD of the middle matrix in (5.3.3), we note that, by permuting the k-th row and column to the first row and column, this matrix can be written as

$$M = \begin{pmatrix} z_1 & z_2 & \cdots & z_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}, \tag{5.3.4}$$

where d_i's are the diagonal elements of D_1 and D_2; and z_i's are entries of the k-th row of the middle matrix, with z_1 being the (k,k) entry. We permute the matrix M so we can

[2]Strictly speaking, this process is not quite recursive since, unlike B, B_1 is not a square matrix. This is true for the following singular value recursion also. See [51] for the complete recursions.

write $D = \text{diag}\ (d_1, d_2, \ldots, d_n)$ with[3] $0 \equiv d_1 \le d_2 \le \ldots \le d_n$, and $z = (z_1, z_2, \ldots, z_n)^T$. We further assume that

$$d_{j+1} - d_j \ge \tau \|M\|_2 \quad \text{and} \quad |z_j| \ge \tau \|M\|_2\ , \tag{5.3.5}$$

where τ is a small multiple of ε specified in [51]. Any matrix of the form (5.3.4) can be reduced to one that satisfies these conditions by the *deflation* procedure described in [51].

The following lemma characterizes the singular values and singular vectors of M.

Lemma 5.3.1 (Jessup and Sorensen [62]) *Let $S\Sigma G^T$ be the* SVD *of M with*

$$S = (s_1, \ldots, s_n) \quad , \quad \Sigma = \text{diag}\ (\sigma_1, \ldots, \sigma_n) \quad \textit{and} \quad G = (g_1, \ldots, g_n)\ ,$$

where $0 < \sigma_1 < \ldots < \sigma_n$. Then the singular values $\{\sigma_i\}_{i=1}^n$ satisfy the interlacing property

$$0 = d_1 < \sigma_1 < d_2 < \ldots < d_n < \sigma_n < d_n + ||z||_2 \quad ,$$

and the secular equation

$$f(\sigma) = 1 + \sum_{k=1}^{n} \frac{z_k^2}{d_k^2 - \sigma^2} = 0\ .$$

The singular vectors satisfy

$$s_i = \left(-1, \frac{d_2 z_2}{d_2^2 - \sigma_i^2}, \ldots, \frac{d_n z_n}{d_n^2 - \sigma_i^2}\right)^T \Bigg/ \sqrt{1 + \sum_{k=2}^{n} \frac{(d_k z_k)^2}{\left(d_k^2 - \sigma_i^2\right)^2}} \quad , \tag{5.3.6}$$

$$g_i = \left(\frac{z_1}{d_1^2 - \sigma_i^2}, \ldots, \frac{z_n}{d_n^2 - \sigma_i^2}\right)^T \Bigg/ \sqrt{\sum_{k=1}^{n} \frac{z_k^2}{\left(d_k^2 - \sigma_i^2\right)^2}} \quad . \tag{5.3.7}$$

On the other hand, given D and all the singular values, we can construct a matrix with the same structure as (5.3.4).

Lemma 5.3.2 (Gu and Eisenstat [51]) *Given a diagonal matrix $D = \text{diag}\ (d_1, d_2, \ldots, d_n)$ and a set of numbers $\{\hat{\sigma}_i\}_{i=1}^n$ satisfying the interlacing property*

$$0 \equiv d_1 < \hat{\sigma}_1 < d_2 < \ldots < d_n < \hat{\sigma}_n\ , \tag{5.3.8}$$

there exists a matrix

$$\hat{M} = \begin{pmatrix} \hat{z}_1 & \hat{z}_2 & \cdots & \hat{z}_n \\ & d_2 & & \\ & & \ddots & \\ & & & d_n \end{pmatrix}$$

[3] d_1 is introduced to simplify the presentation.

whose singular values are $\{\hat{\sigma}_i\}_{i=1}^n$. *The vector* $\hat{z} = (\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_n)^T$ *is determined by*

$$|\hat{z}_i| = \sqrt{(\hat{\sigma}_n^2 - d_i^2) \prod_{k=1}^{i-1} \frac{(\hat{\sigma}_k^2 - d_i^2)}{(d_k^2 - d_i^2)} \prod_{k=i}^{n-1} \frac{(\hat{\sigma}_k^2 - d_i^2)}{\left(d_{k+1}^2 - d_i^2\right)}}\,, \tag{5.3.9}$$

where the sign of $\hat{z}_i$ *can be chosen arbitrarily.*

We use the root-finder provided by R.-C. Li [73] to find approximate singular values $\{\hat{\sigma}_k\}_{k=1}^n$. Following [51], we then compute $\{\hat{z}_k\}_{k=1}^n$ by using (5.3.9) and compute the left and right singular vectors of M using (5.3.6) and (5.3.7), except we replace z_k by $\hat{z}_k$ using the sign of z_k. It has been shown [51] that this procedure is numerically stable, provided that one computes the differences $d_i - d_j$ to high relative accuracy, for $1 \le i \le j \le n$. This assumption is automatically satisfied on most modern computers except some earlier Cray machines (Cray XMP, YMP, C90 and 2) which do not have a guard digit. We overcome this difficulty by using the following technique provided by Kahan [67]. Before the singular values are computed, we first compute

$$d_i := (d_i + d_i) - d_i \quad \text{for} \quad i = 1, \ldots, n\,.$$

On machines with a guard digit, this does not change d_i at all (barring overflow), but it chops off the last bit of d_i on the above mentioned Cray machines. After doing so, the differences $d_i - d_j$ can be computed to high relative accuracy even on these machines. To the best of our knowledge, our code should work on any commercially significant modern North America computers.

Since S and G are generally dense matrices, storing them explicitly will take $O(n^2)$ storage for the whole recursion. However, we note that they can be reconstructed from $\{\hat{z}_k\}_{k=1}^n$, $\{\hat{\sigma}_k\}_{k=1}^n$, and $\{d_k\}_{k=1}^n$ whenever they are needed[4]. Hence in our implementation, we store these data rather than S and G themselves. This increases the cost of `xBDSDC` by $O(n^2)$ overall, but reduces the memory requirement from $O(n^2)$ to $O(n \log_2 n)$, since there are $O(\log_2 n)$ levels of recursion.

5.4 Computing the Dense SVD

In the current version of LAPACK [2], `xGESVD`, in Stage 1 of the SVD computation (see Section 1.1), the matrices U and V are generated as products of Householder

[4]The actual implementation is slightly more complicated for efficiency and stability reasons.

transformations, and in Stage 2, the matrices Q and W are generated as products of Givens rotations. When the full SVD of A is desired, U and V are explicitly computed and the Givens rotations in Q and W are applied to U and V as soon as they are generated.

In contrast, in `xGESDD`, we first compute the matrices Q and W explicitly, by organizing the computation to use level 3 BLAS as much as possible, and then compute UQ and VW by applying the sequence of Householder transformations to Q and W, respectively. This approach is similar to that used for computing the full eigendecomposition of a dense symmetric matrix by using Cuppen's divide-and-conquer algorithm [84].

If $m \gg n$, it may be more efficient to first perform a QR factorization of A, and then to compute the SVD of the $n \times n$ upper triangular matrix R. Since

$$A = QR \quad \text{and} \quad R = U\Sigma V^T,$$

therefore the SVD of A is given by

$$A = (QU)\Sigma V^T.$$

For the implementation of `xGESDD`, we select a threshold Γ. When $m \geq \Gamma \cdot n$, we first perform a QR factorization, and then compute the SVD of the triangular matrix; otherwise, we compute the SVD of A directly. The magnitude of the threshold Γ is based on the flop count of the QR factorization, bidiagonal reduction (Stage 1) and generation of left and right singular vector matrices. To generate left and right singular vectors, there are two choices. One is to store the orthogonal matrices U and V by sequence of Householder vectors, and apply the Householder vectors to the singular vectors of bidiagonal matrix: Q and W, this will be computed by calling LAPACK routine `xORMBR`. The other way is to generate U and V explicitly by calling LAPACK routine `xORGBR`, then perform two matrix multiplications $U \cdot Q$ and $V \cdot W$ by calling BLAS routine `xGEMM`.

The QR factorization is computed by LAPACK routine `xGEQRF`, the bidiagonal reduction is computed by `xGEBRD`. Since the flop count is different for complex arithmetic from real arithmetic, we will consider both real and complex cases. Table 5.1 shows the flop count of several routines for real and complex arithmetic. Table 5.2 shows the flop count for the real dense SVD of dimension $m \times n$ using different combinations of routines and table 5.3 shows the flop count for the complex SVD. In both tables, we exclude the flop count for `SBDSDC` since it is same in all cases.

Real Arithmetic		
Routine	Description	Flops
SGEBRD	bidiagonal reduction	$2n^2(2m - \frac{2}{3}n)$
SGEQRF	QR factorization	$2n^2(m - \frac{n}{3})$
SORMBR	applying Householder transformations	$2n^2(2m - n)$
SORGBR	generate orthogonal matrix from Householder transformations	$2n^2(m - \frac{n}{3})$
SGEMM	matrix-matrix multiply	$2mn^2$
Complex Arithmetic		
CGEBRD	bidiagonal reduction	$8n^2(2m - \frac{2}{3}n)$
CGEQRF	QR factorization	$8n^2(m - \frac{n}{3})$
CUNMBR	applying Householder transformations	$8n^2(2m - n)$
CUNGBR	generate unitary matrix from Householder transformations	$8n^2(m - \frac{n}{3})$
CGEMM	matrix-matrix multiply	$8mn^2$

Table 5.1: Flop Count of LAPACK Routines

Paths	Combinations of Routines	Flops
Direct SVD 1	`SGEBRD + 2 SORMBR`	$8mn^2 - \frac{4}{3}n^3$
Direct SVD 2	`SGEBRD + 2 SORGBR + 2 SGEMM`	$8mn^2 + \frac{4}{3}n^3$
(QR + SVD) 1	`SGEQRF + SGEBRD + 2 SORMBR + SGEMM`	$4mn^2 + 6n^3$
(QR + SVD) 2	`SGEQRF + SGEBRD + 2 SORGBR + 3 SGEMM`	$4mn^2 + \frac{26}{3}n^3$

Table 5.2: Flop Count of Real SVD Excluding SBDSDC for Different Paths

Paths	Combinations of Routines	Flops
Direct SVD 1	`CGEBRD + 2 CUNMBR`	$32mn^2 - \frac{16}{3}n^3$
Direct SVD 2	`CGEBRD + 2 CUNGBR + 4 SGEMM`	$28mn^2 + \frac{4}{3}n^3$
(QR + SVD) 1	`CGEQRF + CGEBRD + 2 CUNMBR + CGEMM`	$16mn^2 + 24n^3$
(QR + SVD) 2	`CGEQRF + CGEBRD + 2 CUNGBR + CGEMM + 4 SGEMM`	$16mn^2 + \frac{80}{3}n^3$

Table 5.3: Flop Count of Complex SVD Excluding SBDSDC for Different Paths

If A is real, since the path Direct SVD 2 is always more expensive than Direct SVD 1, and (QR + SVD) 2 is more expensive than (QR + SVD) 1, we only need two paths for `SGESDD`. By comparing the flop count, when $m \geq \frac{11}{6}n$, Direct SVD 1 is more expensive than (QR + SVD) 1. Therefore, we pick threshold $\Gamma = 11/6$ and use (QR + SVD) 1 if $m \geq \Gamma \cdot n$; otherwise, we use Direct SVD 1.

In the case when A is complex, we need three paths. Again, by comparing the flop count, we derive two thresholds, $\Gamma_1 = \frac{5}{3}n$ and $\Gamma_2 = \frac{17}{9}$, such that:

- When $m \geq \Gamma_2 \cdot n$, we use (QR + SVD) 1.
- When $\Gamma_1 \cdot n \leq m < \Gamma_2 \cdot n$, we use Direct SVD 2.
- Otherwise, when $m < \Gamma_1 \cdot n$, we use Direct SVD 1.

5.5 Solving Linear Least Squares Problem

In this section, we discuss various linear least squares solvers which are based on SVD using divide-and-conquer, SVD using QR-iteration, and QR factorization.

5.5.1 SVD Least Squares Solver Based on Divide-and-Conquer

When we solve the linear least squares problem (5.2.1) using the SVD, LAPACK's `xGELSS` [2] computes the solution x_{LS} using the SVD $A = (UQ)\Sigma(VW)^T$ as follows [43, 48]:

$$x_1 \equiv U_1^T b, \quad x_2 \equiv Q^T x_1, \quad x_3 \equiv \Sigma_1^{-1} x_2, \quad x_{LS} \equiv (VW)x_3, \tag{5.5.10}$$

where U_1 is the first n columns of U. x_1 is computed by applying the Householder transformations directly to b, x_2 is computed by applying the Givens rotations directly to x_1, and x_{LS} is computed by explicitly forming the matrix VW, as is done in the dense SVD case, and then applying VW to x_3. We note that computing VW takes $O(n^3)$ flops in general.

To compute the least squares solution x_{LS} more quickly using the SVD, we use the "factored form" version of bidiagonal divide-and-conquer. After Stage 1, A is reduced to the upper bidiagonal matrix B, with the orthogonal transformations U and V returned as products of Householder transformations. We then compute x_1 as in (5.5.10). This can be done in $O(mn)$ flops [43]. To compute x_2, we note that Q is represented as a product of $O(\log_2 n)$ orthogonal matrices, the i-th of which is block diagonal with the diagonal blocks being 1's and left singular vector matrices on the i-th level in the recursion. Since there are 2^{i-1} submatrices on the i-th level with each submatrix having size $O(n/2^{i-1})$, the cost of applying the transposes of these matrices to a vector is

$$O\left(\left(n/2^{i-1}\right)^2\right) \times 2^{i-1} = O\left(n^2/2^{i-1}\right) ,$$

summing all these costs up, the cost for computing $x_2 = Q^T x_1$ is

$$\sum_{i=1}^{O(\log_2 n)} O\left(n^2/2^{i-1}\right) = O(n^2)$$

flops. Computing x_3 takes $O(n)$ flops. To compute x_{LS} from x_3, we do not explicitly form VW. Instead, we compute

$$x_4 \equiv W x_3 \quad \text{and} \quad x_{LS} = V x_4 . \tag{5.5.11}$$

By the same argument as above, x_4 can be computed in $O(n^2)$ flops. Finally, it is again well known that computing x_{LS} as Vx_4 takes $O(n^2)$ flops [43]. Overall, computing x_{LS} after Phase I takes $O(mn)$ flops.

The routine for solving the least squares problem using divide-and-conquer is called `xGELSD`; this name will be used in section 5.6.

5.5.2 SVD Least Squares Solver Based on QR Iteration

It turns out that explicit computation of VW can be avoided even with the QR based SVD algorithms [48, 29], as originally noted in [21]. Instead of computing x_{LS} as $(VW)x_3$, we can again compute x_{LS} as in (5.5.11). x_4 can be computed by saving all $O(n^2)$ Givens rotations performed in computing the SVD of B, and applying them to x_3 in reverse order; x_{LS} can then be computed as Vx_4 as above. Let t be the total number of such Givens rotations. Then the cost of computing x_{LS} after Stage 1 is $O(mn + t)$ flops. Since we usually expect $t = O(n^2)$, this cost is again $O(mn)$ flops. One drawback with this approach, however, is that it requires $O(t)$ storage, and we cannot bound t exactly beforehand.

The routine implementing this idea is called `xGELSF` (for "factored form") this name will be used in section 5.6.

Recall that `xBDSQR` computes the singular values to high relative accuracy, whereas `xBDSDC` can only guarantee the absolute accuracy. Therefore it seems that there is still some room left to improve the performance of `xGELSF` since in general we only need absolute accuracy to solve a least squares problem.

5.5.3 Least Squares Solvers Based on QR Factorization

The least squares problem can be also solved using QR factorization. The solvers based on QR factorization usually run much faster than SVD based methods. However, they are not as reliable as those based on SVD, since they are not as accurate when the problem is rank-deficient. Of all the methods for solving least squares problem, the fastest as well as the least reliable one is the plain QR which we now describe.

Assume the QR factorization of A is given by

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix} = (Q_1 \quad Q_2) \begin{bmatrix} R \\ 0 \end{bmatrix}, \quad m \geq n,$$

where R is an $n \times n$ upper triangular matrix, Q is an $m \times m$ orthogonal matrix, Q_1 consists of the first n columns of Q and Q_2 the remaining $m - n$ columns.

If A has full column rank, since

$$\|b - Ax\|_2 = \|Q^T b - Q^T Ax\|_2 = \| \begin{bmatrix} c_1 - Rx \\ c_2 \end{bmatrix} \|_2,$$

where $c_1 = Q_1^T b$ and $c_2 = Q_2^T b$, x_{LS} is then computed by solving the upper triangular system

$$Rx = c_1.$$

This algorithm is implemented in LAPACK. The name of the routine is `xGELS`.

When A is not of full rank, or the rank of A is in doubt, we can perform a QR factorization with column pivoting, which was introduced by Businger and Golub [17, 44]. It is more reliable than plain QR, but it is slower.

The QR Factorization with column pivoting is given by

$$A = Q \begin{bmatrix} R \\ 0 \end{bmatrix} P^T, \quad m \geq n,$$

where Q and R are as before and P is a permutation matrix, such that

$$|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{nn}|$$

and for each k,

$$|r_{kk}| \geq \|R_{k:j,j}\|_2 \quad \text{for } j = k+1, \ldots, n.$$

In exact arithmetic, if $\text{rank}(A) = k$, then the submatrix R_{22} in rows and columns $k+1$ to n would be 0. However, in floating point arithmetic, we can only expect that we can determine an index k, such that the leading principal matrix in the first k rows and columns is well conditioned, and R_{22} is negligible:

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix} \simeq \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}.$$

Then k is the effective rank of A. The solution to the least squares problem is then given by

$$x_{LS} = P \begin{bmatrix} R_{11}^{-1} c_1 \\ 0 \end{bmatrix},$$

where c_1 contains the first k elements of $c = Q^T b$. The QR factorization with column pivoting does not enable us to compute a minimum norm solution to a rank-deficient linear least squares problem, unless $R_{12} = 0$. However, by applying further orthogonal transformations from the right to the upper trapezoidal matrix $(R_{11} \quad R_{12})$, R_{12} can be eliminated:

$$(R_{11} \quad R_{12}) Z = (T_{11} \quad 0).$$

This gives the *complete orthogonal factorization*

$$AP = Q \begin{bmatrix} T_{11} & 0 \\ 0 & 0 \end{bmatrix} Z^T.$$

Thus, the minimum norm solution can be computed as

$$x_{LS} = PZ \begin{bmatrix} T_{11}^{-1} c_1 \\ 0 \end{bmatrix}.$$

This algorithm is implemented in LAPACK's routine `xGELSX` [3].

QR with column pivoting works well in practice for general matrices, but it can fail in pathological cases [65]. To overcome this, several more sophisticated "Rank-Revealing QR" algorithms have been developed [18, 93, 50, 46, 19, 79, 59].

LAPACK's routine `xGELSX` is based on `xGEQPF`, which implements QR with column pivoting using Level 2 BLAS [37]. Recently, Bischof et al [39] developed a variant algorithm of the QR factorization with pivoting which allows the use of Level 3 BLAS [36], thus increasing cache data locality while enabling the use of the most efficient BLAS kernels but it takes much longer in the worst case although this rarely happens. The routine for computing the least squares problem based on this new algorithm is called `xGELSY`.

Bischof and Quintana-Orti also implemented two rank-revealing QR least squares solvers [13]. One is related to Chandrasekaran and Ispen's algorithm [19], which is called `xGELSA`. The other is related to Pan and Tang's algorithm [79], which is called `xGELSB`.

The implementations `xGELSY`, `xGELSA`, `xGELSB` are available through Bischof[5].

5.6 Numerical Experiments on the RS6000/590

We ran our numerical experiments on an IBM RS6000/590 with a 66.5 Mhz clock and 256KB cache. We compiled using xlf with the -O3 optimization option. The optimized BLAS were those in IBM's Engineering and Scientific Subroutine Library (ESSL)[1]. All experiments were run in double precision, i.e. 64-bit, IEEE floating point arithmetic. We let $\varepsilon = 2^{-53}$ denote the machine precision.

Table 5.4 lists the names of the subroutines we test and what they do. The reader may want to refer to this table to interpret the following performance tables.

5.6.1 Performance of the BLAS and basic LAPACK decompositions on the RS6000

Table 5.5 reports on the speed in Megaflops of the BLAS, `DGEMV` (matrix–vector multiplication) and `DGEMM` (matrix-matrix multiplication). It also reports the speeds of LU decomposition (`DGETRF`), QR decomposition (`DGEQRF`) and bidiagonal reduction (`DGEBRD`). It does this both for Fortran BLAS and ESSL BLAS. All matrices are dimensioned (LDA,N), where LDA = 1601. The block size NB in the blocked algorithms for `DGETRF`, `DGEQRF` and `DGEBRD` was 32. It is interesting to see that the performance of `DGEMV` is a strongly *nonmonotonic* function of matrix dimension. This is because the cache size for RS6000/590 is 256KB, when the matrix size is small, the matrix fits in cache; otherwise, cache misses will occur.

---

[5]email: bischof@mcs.anl.gov

Table 5.4: Names and descriptions of routines tested

Name	Description	Status
Fundamental Routines		
xGEMV	Matrix-vector multiply	Level 2 BLAS
xGEMM	Matrix-matrix multiply	Level 3 BLAS
xGETRF	LU decomposition	in LAPACK
xGEQRF	QR decomposition	in LAPACK
xGEBRD	Reduction to bidiagonal form	in LAPACK
Bidiagonal SVD		
xBDSQR	Compute complete SVD of a bidiagonal matrix using QR iteration	in LAPACK
xBDSDC	Compute complete SVD of a bidiagonal matrix using divide-and-conquer	new routine
Dense SVD		
xGESVD	Compute complete SVD of a dense matrix using QR iteration	in LAPACK
xGESVF	Compute complete SVD of a dense matrix using QR iteration, version of DGESVD optimized for RS6000	in ESSL
xGESDD	Compute complete SVD of a dense matrix using divide-and-conquer	new routine
Linear Least Squares Solvers		
xGELS	Solve the LS problem using QR decomposition	in LAPACK
xGELSX	Solve the LS problem using QR decomposition with column pivoting	in LAPACK
xGELSY	Solve the LS problem using QR decomposition with column pivoting implemented by BLAS 3 (Bischof et al [39])	new routine
xGELSA	Solve the LS problem using Rank-Revealing QR based on the method related to Chandrasekaran and Ipsen's algorithm (Bischof et al [13, 19])	new routine
xGELSB	Solve the LS problem using Rank-Revealing QR based on the method related to Pan and Tang's algorithm (Bischof et al [13, 79])	new routine
xGELLS	Solve the LS problem using QR decomposition with column pivoting, version of DGELSX optimized for RS6000	in ESSL
xGELSS	Solve the LS problem using the SVD based on QR-iteration	in LAPACK
xGESVS	Solve the LS problem using the SVD based on QR-iteration, version of DGELSS optimized for RS6000	in ESSL
xGELSF	Solve the LS problem using the SVD based on QR-iteration but where the left singular vectors are left factored	new routine
xGELSD	Solve the LS problem using the SVD based on divide-and-conquer	new routine

Table 5.5: Speed of BLAS and LAPACK Routines on RS6000 (NB = 32, LDA = 1601)

Speed in megaflops using ESSL BLAS							
		Dimension					
Routine	Description	50	100	200	400	800	1600
DGEMV	matrix-vector multiply	128.4	205.1	114.6	116.5	117.4	120.3
DGEMM	matrix-matrix multiply	210.3	229.2	224.0	228.6	229.1	233.5
DGETRF	LU decomposition	42.4	119.1	151.8	163.8	152.2	171.8
DGEQRF	QR decomposition	60.2	96.7	139.3	171.3	189.5	197.7
DGEBRD	Bidiagonal reduction	33.8	135.1	102.0	113.2	125.4	138.0
Speed in megaflops using Fortran BLAS							
		Dimension					
Routine	Description	50	100	200	400	800	1600
DGEMV	matrix-vector multiply	68.7	81.2	62.9	65.1	67.0	68.3
DGEMM	matrix-matrix multiply	70.9	80.0	64.9	65.3	67.4	68.6
DGETRF	LU decomposition	41.7	56.7	68.9	71.0	70.2	65.5
DGEQRF	QR decomposition	50.5	69.0	78.2	77.9	80.0	76.0
DGEBRD	Bidiagonal reduction	48.8	65.9	58.7	63.6	64.4	61.4

5.6.2 Performance of the Bidiagonal SVD on the RS6000

We report on the speed of the bidiagonal SVD (computing all singular values and left and right singular vectors). We used four types of test matrices, all generated by LAPACK test matrix generator `DLATMS`:

Type 1. These bidiagonal matrices were randomly generated with singular values distributed arithmetically from ε up to 1.

Type 2. These bidiagonal matrices were randomly generated with singular values distributed geometrically from ε up to 1.

Type 3. These bidiagonal matrices have 1 singular value at 1 and the other $n-1$ clustered at ε.

Type 4. These bidiagonal matrices were generated by taking a dense matrix with independent random entries uniformly distributed in $(-1, 1)$, and reducing it to bidiagonal form.

Table 5.6 shows the speedup of `DBDSDC`, the bidiagonal SVD based on divide-and-conquer, with respect to `DBDSQR`, the bidiagonal SVD based on QR-iteration (all singular

Table 5.6: Speedup of DBDSDC over DBDSQR on RS6000

Speedup using ESSL BLAS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	0.93	1.29	2.11	3.16	5.16	30.91	47.78
type 2	0.91	0.83	1.78	3.60	5.92	37.38	67.14
type 3	1.06	6.67	22.00	48.00	58.57	365.56	938.83
type 4	1.11	1.29	1.89	3.43	5.59	33.00	50.51
Speedup using Fortran BLAS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	1.00	1.31	1.31	2.03	2.82	15.45	20.33
type 2	0.76	0.90	1.17	2.70	4.30	26.61	42.06
type 3	1.00	24.69	38.00	83.83	67.69	361.54	927.03
type 4	1.00	1.00	1.25	2.26	2.97	16.50	21.86

values and left and right singular vectors are computed). As can be seen, the speedup is large and a growing function of matrix dimension. For $n = 1600$, the speedup is 50 for matrices of type 4 and over 900 for matrices of type 3, when using ESSL BLAS. Also, the speedup is better when using the optimized BLAS rather than Fortran BLAS, because DBDSDC spends much of its time in DGEMM, whereas DBDSQR cannot even use Level 2 BLAS. In general, DBDSDC can only compute the singular values to high absolute accuracy; in contrast, DBDSQR can compute the singular values to high *relative accuracy* [33].

Figure 5.1 shows the performance of DBDSQR and DBDSDC in MFLOPS for type 2 and 4 matrices using ESSL BLAS, where DBDSQR is plotted by blue line and DBDSDC by red line. We can see that the MFLOPS of DBDSDC is a growing function of matrix size whereas the MFLOPS of DBDSQR peaks around $n = 400$ then drops very quickly. When $n = 1600$, for matrices of type 4, DBDSDC achieves 117 MFLOPS whereas DBDSQR runs only at 14 MFLOPS.

5.6.3 Performance of the Dense SVD on the RS6000

We report on the speed of the dense SVD (computing all singular values and left and right singular vectors). We used the same four test matrix types as before, but now all are dense. We compared DGESDD and DGESVD with ESSL's SVD routine DGESVF; see tables 5.7 and 5.8. We also compared the performance of DGESDD and DGESVD using Fortran BLAS,

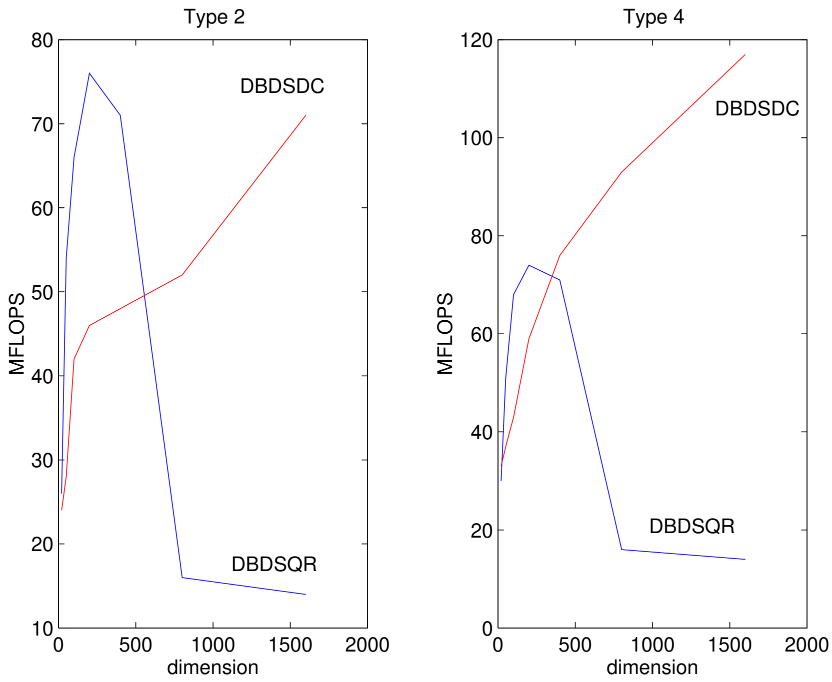


Figure 5.1: Performance of DBDSQR and DBDSDC in MFLOPS

see table 5.9. For all four types of matrices, DGESDD achieved a good speedup over DGESVF. For $n = 1600$, the speedup of DGESDD over DGESVF ranges from 6.3 to 14.5. For matrices of types 1,2,4, the speed of DGESVD is comparable to DGESVF. However, DGESVF achieved a good speedup over DGESVD for matrices of type 3. This is because DGESVD computes the singular values of the bidiagonal matrix to high relative accuracy whereas DGESVF does not.

Table 5.10 shows what fraction of time the dense SVD spends on doing the bidiagonal SVD. The most significant result is that the bidiagonal fraction takes 91% to 96%

Table 5.7: Speedup of DGESDD over DGESVF on RS6000

Speedup using ESSL BLAS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	1.11	1.25	1.63	2.31	3.06	11.62	14.47
type 2	0.82	0.67	0.85	1.54	2.05	5.34	6.32
type 3	0.30	0.64	0.89	1.64	2.13	5.78	7.03
type 4	0.82	1.00	1.41	2.28	3.13	11.36	14.20

Table 5.8: Speedup of `DGESVF` over `DGESVD` on RS6000

Speedup using ESSL BLAS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	0.90	0.88	1.00	0.84	0.80	0.90	0.93
type 2	1.17	1.46	1.36	0.99	0.85	1.13	1.20
type 3	3.23	3.70	3.63	2.22	1.67	2.62	3.11
type 4	1.15	1.05	1.00	0.83	0.80	0.94	0.95

Table 5.9: Speedup of `DGESDD` over `DGESVD` on RS6000

Speedup using Fortran BLAS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	1.08	0.93	1.08	1.36	1.55	5.26	5.99
type 2	0.88	0.86	0.90	1.20	1.29	3.58	4.00
type 3	1.03	2.56	2.31	2.38	2.26	7.67	9.54
type 4	0.88	0.92	1.04	1.28	1.47	5.29	6.03

of the total time for `DGESVD` to at most 27% for `DGESDD`, for large matrices. For matrices of type 3, `DBDSDC` only costs 2% of the total time in contrast to 96% by `DBDSQR`, which means bidiagonal SVD is never the bottleneck in the dense SVD.

Thus, any significant further improvements in the speed of the dense SVD for large matrices must come from speeding up the non-bidiagonal part of the computation. One way to do this is to abandon computing the singular vectors explicitly, leaving them in the factored form provided by the algorithm. We exploit this possibility in the next section.

Table 5.11 shows how well the performance of `DGESVD`, `DGESVF` and `DGESDD` as well as `DGEBRD`, `DORGBR` and `DORMBR` compares to the speed of `DGEMM` (ESSL), the measure we used is run-time(dense SVD)/run-time(DGEMM of same matrix size). We can see that when $n = 1600$, the run-time of computing a dense SVD is reduced from over 90 matrix multiplications to under 7 matrix multiplications. This is a big improvement.

Figure 5.2 shows the performance of `DGESVD` and `DGESDD` in MFLOPS for type 2 and 4 matrices, where `DGESVD` is plotted by blue line and `DGESDD` by red line. Again, we can see that for matrices of type 4, when $n = 1600$, `DGESDD` achieves nearly 150 MFLOPS

Table 5.10: Fraction of time Dense SVD spends in Bidiagonal SVD (ESSL BLAS on RS6000)

Fraction of DGESDD spent in DBDSDC							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	0.72	0.58	0.56	0.46	0.39	0.32	0.27
type 2	0.50	0.50	0.35	0.25	0.19	0.14	0.10
type 3	0.64	0.26	0.11	0.06	0.05	0.04	0.02
type 4	0.58	0.52	0.53	0.44	0.35	0.30	0.25

Fraction of DGESVD spent in DBDSQR							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	0.67	0.68	0.73	0.75	0.82	0.95	0.96
type 2	0.48	0.43	0.53	0.58	0.64	0.89	0.91
type 3	0.70	0.72	0.76	0.75	0.75	0.95	0.96
type 4	0.68	0.63	0.71	0.80	0.79	0.94	0.94

whereas `DGESVD` runs at only 20 MFLOPS.

5.6.4 Performance of Solvers for the Linear Least Squares Problem on the RS6000

We consider solving least squares problems with single right hand sides. We use the same four test matrices as before. The algorithms we consider are

- `DGELS` – QR decomposition (currently in LAPACK)
- `DGELSX` – QR decomposition with column pivoting (currently in LAPACK)
- `DGELSY` – QR decomposition with column pivoting but implemented using BLAS 3 [39]
- `DGELSA` – Rank-Revealing QR based on method related to Chandrasekaran and Ipsen's algorithm [13, 19]
- `DGELSB` – Rank-Revealing QR based on method related to Pan and Tang's algorithm number 3 [13, 79]
- `DGELSS` – SVD based on QR iteration (currently in LAPACK)

Table 5.11: Ratios of run-time(dense SVD) to run-time(DGEMM(ESSL)) on RS6000

Time(DGESVD) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	60.65	37.18	29.80	22.40	21.43	80.09	89.77
type 2	37.34	24.57	17.19	13.02	12.50	39.15	44.17
type 3	65.19	46.69	33.23	22.40	19.64	77.85	102.88
type 4	63.28	39.96	27.50	21.00	21.42	78.74	88.63

Time(DGESVF) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	67.31	42.07	29.79	26.60	26.79	88.37	96.89
type 2	31.89	16.83	12.60	13.16	14.64	34.68	36.76
type 3	20.18	12.62	9.17	10.08	11.79	29.75	33.06
type 4	55.00	37.86	27.50	25.20	26.79	83.89	93.47

Time(DGESDD) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	60.55	33.65	18.33	11.48	8.75	7.61	6.70
type 2	38.85	22.23	14.90	8.54	7.14	6.49	5.81
type 3	67.31	19.60	10.31	6.16	5.54	5.15	4.70
type 4	67.31	37.86	19.48	11.06	8.57	7.38	6.58

Time(DGEBRD) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
all types	10.40	4.78	3.25	2.94	2.64	2.45	2.25

Time(DORGBR) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
all types	3.73	2.01	1.49	1.06	0.90	0.86	0.84

Time(DORMBR) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
all types	4.67	2.69	1.83	1.37	1.22	1.270	1.18

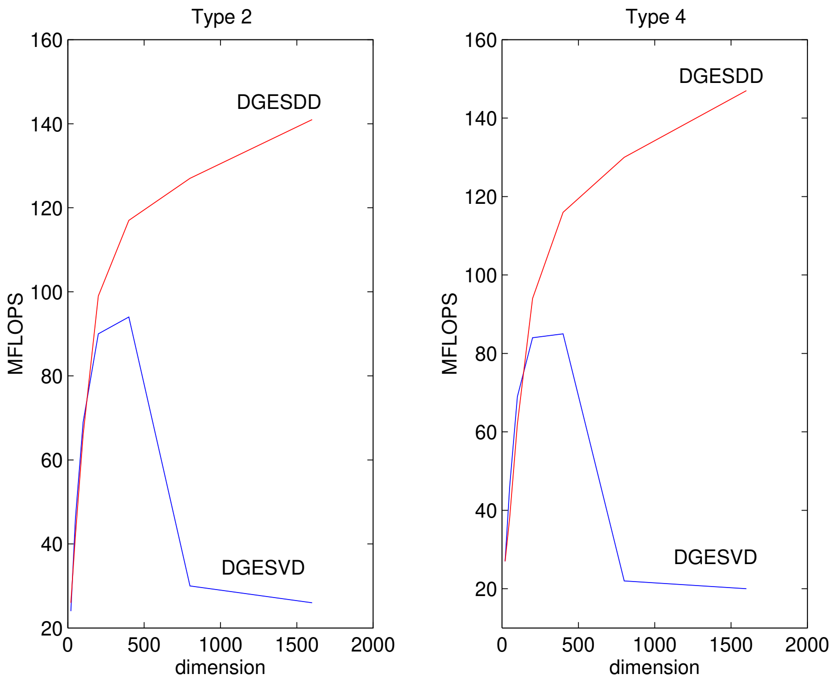


Figure 5.2: Performance of DGESVD and DGESDD in MFLOPS

- DGELSF – SVD based on QR iteration but maintaining the left singular vectors of the bidiagonal matrix as a list of $O(N^2)$ Givens rotations
- DGELSD – SVD based on divide-and-conquer, factored form
- DGELLS – QR decomposition with pivoting (currently in ESSL)
- DGESVS – SVD based on QR iteration (currently in ESSL)

We present square problems only, since M-by-N problems with $M \gg N$ are generally reduced to an N-by-N problem by an initial QR decomposition, and this dominates all later computations. All the computations are done in single precision.

In addition to measuring the speedup of DGELSD and DGELSF over DGELSS, we measure times relative to DGELS, the fastest, and least reliable, of all the methods. This quantifies the tradeoff between speed and reliability inherent in this problem. Results shown in tables are for ESSL BLAS only.

Table 5.12 shows that both new least squares solvers, DGELSF and DGELSD, are significantly faster than the older DGELSS. For matrices of size $n = 1600$, DGELSF and DGELSD achieves the speedup ranges from 15 — 40.

Table 5.12: Speedups of New SVD-based Least Squares Solvers (using ESSL BLAS on RS6000)

Speedup of DGELSD over DGELSS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	1.15	1.00	1.21	1.69	2.58	19.41	29.00
type 2	1.06	0.87	1.00	1.67	2.29	12.31	15.91
type 3	1.02	3.50	4.50	3.85	4.56	29.09	40.74
type 4	0.89	0.75	1.00	1.61	2.63	18.82	28.00
Speedup of DGELSF over DGELSS							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	1.26	1.30	1.70	2.20	3.20	22.00	30.21
type 2	1.14	1.33	1.29	1.86	2.40	12.31	15.91
type 3	1.22	1.30	1.64	2.08	3.04	21.33	34.02
type 4	1.09	1.00	1.60	2.13	3.16	21.33	29.17

Table 5.13 shows that a fully reliable SVD-based solution to the linear least square problem now costs no more than 3.46 times as much as the fastest solver (`DGELS`), whereas it used to cost at least 48 times more for $n = 1600$. This is a big improvement. In particular, for matrices of type 1 and 3, it takes more than 100 times as much as `DGELS`, we believe this is due to `DBDSQR` runs at a very low MFLOPS rate when matrices are large.

Table 5.15 shows how well the performance of the various least squares solvers compares to the speed of DGEMM(ESSL). The measures we used is run-time(least squares solver)/run-time(DGEMM of the same matrix size). We can see that for the matrices with dimension as large as $n = 1600$, `DGELSF` and `DGELSD` takes less than 3 matrix multiplications to solve a least squares problem fully reliably, whereas the older solver `DGELSS` has to take at least 40 matrix multiplications.

5.6.5 Accuracy Assessment on the RS6000

We use two measures of accuracy of the computed SVD $A = X\Sigma Y^T$: the residual $\max_i \|Ay_i - \sigma_i x_i\|/(\varepsilon\sigma_1)$ and the orthogonality of the singular vectors $\max(\|YY^T - I\|/\varepsilon, \|XX^T - I\|/\varepsilon)$, where ε is machine precision. Ideally these two measure should be $O(1)$ for any dimension, but we would not be unhappy to get numbers growing with N, perhaps as $O(N)$, although we cannot prove so tight a bound. In fact, the QR based SVD

Table 5.13: Timings of Least Squares Solvers relative to DGELS (using ESSL BLAS on RS6000)

Time(DGELSX) / Time(DGELS)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	2.29	2.00	1.75	1.50	2.14	2.16	2.11
type 2	2.56	1.72	2.00	1.55	2.10	2.31	2.28
type 3	1.59	1.26	1.50	1.78	2.11	2.11	2.08
type 4	2.54	1.74	1.20	1.60	1.96	2.08	2.00

Time(DGELSY) / Time(DGELS)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	2.29	2.33	1.50	1.50	1.63	1.53	1.40
type 2	2.56	2.00	2.00	1.36	1.75	1.59	1.48
type 3	1.71	1.42	1.50	1.56	1.56	1.45	1.39
type 4	2.54	1.93	1.20	1.50	1.59	1.47	1.31

Time(DGELSA) / Time(DGELS)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	2.99	2.79	2.00	1.50	1.21	1.21	1.14
type 2	3.41	2.40	2.00	1.27	1.49	1.44	1.45
type 3	1.95	1.34	1.25	1.11	1.12	1.05	1.07
type 4	3.56	2.33	1.40	1.30	1.16	1.16	1.10

Time(DGELSB) / Time(DGELS)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	2.99	2.33	1.75	1.30	1.20	1.16	1.14
type 2	2.68	2.00	1.50	1.27	1.33	1.41	1.41
type 3	2.32	2.00	1.75	1.78	1.47	1.24	1.18
type 4	3.22	2.33	1.00	1.30	1.20	1.16	1.10

Time(DGELLS) / Time(DGELS)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	1.84	1.93	1.25	1.50	1.96	2.08	2.07
type 2	1.06	1.12	1.25	1.09	1.93	2.00	2.00
type 3	1.22	1.26	1.50	1.89	2.28	2.53	2.07
type 4	1.44	1.30	1.00	1.40	1.96	2.08	2.00

Table 5.14: Continued: Timings of Least Squares Solvers relative to `DGELS` on RS6000

Time(`DGELSF`) / Time(`DGELS`)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	5.75	6.28	5.00	5.00	4.46	3.95	3.43
type 2	3.54	3.00	3.50	3.17	3.51	3.33	3.03
type 3	5.61	5.40	5.50	5.33	4.21	3.95	3.46
type 4	7.80	6.98	4.00	4.70	4.46	3.95	3.31

Time(`DGELSD`) / Time(`DGELS`)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	6.32	8.14	7.00	6.50	5.54	4.47	3.57
type 2	3.78	4.60	4.50	3.55	3.68	3.33	3.03
type 3	6.71	2.00	2.00	2.89	2.81	2.89	2.89
type 4	9.49	9.30	6.40	6.20	5.35	4.47	3.45

Time(`DGELSS`) / Time(`DGELS`)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	7.24	8.14	8.50	11.00	14.29	86.84	103.57
type 2	4.02	4.00	4.50	5.91	8.42	41.03	48.28
type 3	6.83	7.00	9.00	11.11	12.81	84.21	117.86
type 4	8.47	6.98	6.40	10.00	14.11	84.21	96.55

Time(`DGESVS`) / Time(`DGELS`)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	6.32	6.98	8.00	11.00	14.29	15.00	15.00
type 2	3.17	3.40	4.00	5.00	7.89	8.72	8.62
type 3	2.20	2.00	2.50	4.56	6.49	7.37	7.50
type 4	7.80	8.14	6.80	10.00	13.57	14.47	14.14

Table 5.15: Ratios of run-time(least squares solver) to run-time(DGEMM(ESSL)) on RS6000

Time(DGELS) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	9.78	3.62	2.29	1.40	1.00	0.85	0.80
type 2	8.28	4.21	2.29	1.54	1.02	0.87	0.83
type 3	8.28	4.21	2.29	1.26	1.02	0.85	0.80
type 4	5.95	3.62	2.86	1.40	1.00	0.85	0.83
Time(DGELSX) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	20.02	7.24	4.01	2.10	2.14	1.83	1.68
type 2	21.19	7.24	4.58	2.38	2.14	2.01	1.88
type 3	13.12	5.30	3.44	2.24	2.14	1.79	1.65
type 4	15.14	6.31	3.44	2.24	1.96	1.77	1.65
Time(DGELSY) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	20.18	8.41	3.44	2.10	1.63	1.30	1.11
type 2	21.19	8.41	4.58	2.10	1.79	1.39	1.23
type 3	14.13	5.97	3.44	1.96	1.59	1.23	1.11
type 4	15.14	6.98	3.44	2.10	1.59	1.25	1.08
Time(DGELSA) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	26.24	10.10	4.58	2.10	1.21	1.03	0.91
type 2	28.26	10.10	4.58	1.96	1.52	1.25	1.20
type 3	16.15	5.64	2.86	1.40	1.14	0.89	0.85
type 4	21.19	8.41	4.01	1.82	1.16	0.98	0.91
Time(DGELSB) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	26.24	8.41	4.01	1.82	1.20	0.98	0.91
type 2	22.20	8.41	3.44	1.96	1.36	1.23	1.17
type 3	19.17	8.41	4.01	2.24	1.50	1.05	0.94
type 4	19.17	8.41	2.86	1.82	1.20	0.98	0.91

Table 5.16: Continued: Ratios of run-time(least squares solver) to run-time(DGEMM(ESSL)) on RS6000

Time(DGELLS) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	16.15	6.98	2.86	2.10	1.96	1.77	1.65
type 2	8.78	4.71	2.86	1.68	1.96	1.75	1.65
type 3	10.09	5.30	3.44	2.38	2.32	2.15	1.65
type 4	8.58	4.71	2.86	1.96	1.96	1.77	1.65

Time(DGELSS) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	63.58	29.45	19.48	15.40	14.29	73.83	82.64
type 2	33.30	16.83	10.31	9.10	8.57	35.79	39.90
type 3	56.51	29.45	20.63	14.00	13.04	71.59	94.04
type 4	50.46	25.24	18.33	14.00	14.11	71.59	79.80

Time(DGELSF) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	50.46	22.72	11.46	7.00	4.46	3.36	2.73
type 2	29.27	12.62	8.02	4.90	3.57	2.90	2.51
type 3	46.42	22.72	12.60	6.72	4.29	3.36	2.76
type 4	46.42	25.24	11.46	6.58	4.46	3.36	2.73

Time(DGELSD) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	55.50	29.45	16.04	9.10	5.54	3.80	2.85
type 2	31.28	19.35	10.31	5.46	3.75	2.91	2.51
type 3	55.50	8.41	4.58	3.64	2.86	2.46	2.31
type 4	56.51	33.65	18.33	8.68	5.36	3.80	2.85

Time(DGESVS) / Time(DGEMM)							
	Dimension						
Test Matrix	20	50	100	200	400	800	1600
type 1	55.50	25.24	18.33	15.40	14.29	12.75	11.97
type 2	26.24	14.30	9.17	7.70	8.04	7.61	7.12
type 3	18.17	8.41	5.73	5.74	6.61	6.26	5.98
type 4	46.42	29.45	19.48	14.00	13.57	12.30	11.68

routines exhibit measures as large as $N/5$ for $N = 400$, whereas the measures for divide-and-conquer routines never exceeds 10. Therefore, the divide-and-conquer based SVD is not only faster but more accurate than the QR based approach.

The above results are for dense matrices. It turns out one can prove tighter relative error bounds for singular values and singular vectors for the QR-based bidiagonal SVD [33, 24]. We currently cannot guarantee this high relative accuracy with divide-and-conquer, just the absolute accuracy described in the last paragraph.

By comparing the residuals and orthogonality, in most cases, we observed that the computational routines in ESSL like `DGESVS` are not as accurate as those in LAPACK or the new ones. For example, for dense SVD, `DGESDD` and `DGESVD` are almost one decimal digit more accurate than ESSL's `DGESVF`.

Chapter 6

Generalized Singular Value Decomposition

6.1 Introduction

Since it was introduced by Van Loan [95], generalized singular value decomposition(GSVD) has been found to be a very useful tool in numerical linear algebra. Its applications in many generalized problems are in the same spirit as the SVD in corresponding standard problems [9], such as in finding the intersection of the null spaces of two matrices [43], in the generalized eigenvalue problem arising from signal processing [90], in computing the Kronecker form of matrix pencil $A - \lambda B$ [64], in the constrained least squares problem [43], in the least squares problem with Tikhonov regularization [53], and so on.

In this chapter, we first discuss two improvements we made on the LAPACK's `xGGSVD` which implemented a variation of Paige's algorithm [77] by Bai and Demmel [10]. One is in the stopping criteria, the other is in "postprocessing". We then discuss an implementation of Van Loan's algorithm [96] which is based on the divide-and-conquer SVD and the QR decomposition. We show that our implementation achieves good speedups over the `SGGSVD`.

6.2 Review of GSVD

The **generalized(or quotient) singular value decomposition** of an $m \times n$ matrix A and a $p \times n$ matrix B is given by the pair of factorizations [78, 2]

$$A = U\Sigma_1[0 \quad R]Q^T \quad \text{and} \quad B = V\Sigma_2[0 \quad R]Q^T, \tag{6.2.1}$$

where $U \in \mathcal{R}^{m\times m}$, $V \in \mathcal{R}^{p\times p}$ and $Q \in \mathcal{R}^{n\times n}$ are orthogonal matrices. R is an $r \times r$ nonsingular upper triangular matrix, where $r \leq n$ is the rank of $\begin{bmatrix} A \\ B \end{bmatrix}$. Σ_1 is an $m \times r$ diagonal matrix, Σ_2 is a $p \times r$ diagonal matrix, the diagonal elements of both matrices are nonnegative, and satisfy

$$\Sigma_1^T\Sigma_1 + \Sigma_2^T\Sigma_2 = I.$$

Let

$$\Sigma_1^T\Sigma_1 = \begin{bmatrix} \alpha_1^2 & & & \\ & \alpha_2^2 & & \\ & & \ddots & \\ & & & \alpha_r^2 \end{bmatrix} \quad \text{and} \quad \Sigma_2^T\Sigma_2 = \begin{bmatrix} \beta_1^2 & & & \\ & \beta_2^2 & & \\ & & \ddots & \\ & & & \beta_r^2 \end{bmatrix},$$

where $0 \leq \alpha_i, \beta_i \leq 1$. The ratios

$$\frac{\alpha_1}{\beta_1}, \frac{\alpha_2}{\beta_2}, \ldots, \frac{\alpha_r}{\beta_r}$$

are called the **generalized singular values** of the matrix pair A, B. If $\beta_i = 0$, then the generalized singular value α_i/β_i is **infinite**.

More precisely, if $m - r \geq 0$, then

$$\Sigma_1 = \begin{array}{c c} & \begin{array}{cc} k & l \end{array} \\ \begin{array}{c} k \\ l \\ m-k-l \end{array} & \begin{pmatrix} I & 0 \\ 0 & C \\ 0 & 0 \end{pmatrix} \end{array} \quad \text{and} \quad \Sigma_2 = \begin{array}{c c} & \begin{array}{cc} k & l \end{array} \\ \begin{array}{c} l \\ p-l \end{array} & \begin{pmatrix} 0 & S \\ 0 & 0 \end{pmatrix} \end{array}. \tag{6.2.2}$$

Here l is the rank of B, $k = r - l$, C and S are both diagonal matrices satisfying $C^2 + S^2 = I$, and S is nonsingular. We may also identify

$$\alpha_1 = \cdots = \alpha_k = 1, \quad \alpha_{k+i} = c_{ii}$$

and

$$\beta_1 = \cdots = \beta_k = 0, \quad \beta_{k+i} = s_{ii}$$

for $i = 1, \ldots, l$. Thus the first k generalized singular values

$$\frac{\alpha_1}{\beta_1}, \frac{\alpha_2}{\beta_2}, \ldots, \frac{\alpha_k}{\beta_k}$$

are infinite, and the remaining l generalized singular values are finite.

When $m - r < 0$,

$$\Sigma_1 = \begin{array}{c c} & \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{array}{c} k \\ m-k \end{array} & \begin{pmatrix} I & 0 & 0 \\ 0 & C & 0 \end{pmatrix} \end{array}, \quad \Sigma_2 = \begin{array}{c c} & \begin{array}{ccc} k & m-k & k+l-m \end{array} \\ \begin{array}{c} m-k \\ k+l-m \\ p-l \end{array} & \begin{pmatrix} 0 & S & 0 \\ 0 & 0 & I \\ 0 & 0 & 0 \end{pmatrix} \end{array} \tag{6.2.3}$$

Again, l is the rank of B, $k = r - l$, C and S are diagonal matrices satisfying $C^2 + S^2 = I$, S is nonsingular, and

$$\begin{aligned} \alpha_1 &= \cdots = \alpha_k = 1, \quad \alpha_{k+i} = c_{ii}, \alpha_{m+1} = \cdots = \alpha_r = 0, \\ \beta_1 &= \cdots = \beta_k = 0, \quad \beta_{k+i} = s_{ii}, \beta_{m+1} = \cdots = \beta_r = 1. \end{aligned}$$

for $i = 1, \ldots, m - k$. Thus, the first k generalized singular values

$$\frac{\alpha_1}{\beta_1}, \frac{\alpha_2}{\beta_2}, \ldots, \frac{\alpha_k}{\beta_k}$$

are infinite, and remaining l generalized singular values are finite.

In particular, if B is the identity matrix, the GSVD gives the SVD of A. There are several other important special cases of GSVD [2].

- If B is square and nonsingular, then $r = n$ and the GSVD of A and B is equivalent to the SVD of AB^{-1}, where the singular values of AB^{-1} are equal to the generalized singular values of the pair A, B:

$$AB^{-1} = (U\Sigma_1 RQ^T)(V\Sigma_2 RQ^T)^{-1} = U(\Sigma_1\Sigma_2^{-1})V^T.$$

- If the columns of $\begin{bmatrix} A \\ B \end{bmatrix}$ are orthonormal, then $r = n$, $R = I$ and the GSVD of A and B is equivalent to the **CS decomposition** [23, 92, 43] of $\begin{bmatrix} A \\ B \end{bmatrix}$:

$$\begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} U & 0 \\ 0 & V \end{bmatrix} \begin{bmatrix} \Sigma_1 \\ \Sigma_2 \end{bmatrix} Q^T.$$

- The generalized eigenvalues and eigenvectors of $A^T A - \lambda B^T B$ can be expressed in terms of GSVD. Let
$$X = Q \begin{bmatrix} I & 0 \\ 0 & R^{-1} \end{bmatrix},$$
Then
$$X^T A^T A X = \begin{bmatrix} 0 & 0 \\ 0 & \Sigma_1^T \Sigma_1 \end{bmatrix} \quad \text{and } X^T B^T B X = \begin{bmatrix} 0 & 0 \\ 0 & \Sigma_2^T \Sigma_2 \end{bmatrix}.$$
Therefore, the columns of X are the generalized eigenvectors of $A^T A - \lambda B^T B$, and the eigenvalues are the squares of the generalized singular values.

6.3 `SGGSVD` and the Stopping Criterion

LAPACK's `SGGSVD` [2, 10] computes the generalized singular value decomposition of a pair of matrices A and B using a variation of Paige's algorithm [77]. It uses a 2 by 2 triangular GSVD algorithm [10] to provide high accuracy.

Assume A is m-by-n and B is p-by-n. The computation proceeds in the following two steps:

i. **Preprocessing**

A subroutine `SGGSVP` is used to reduce the matrices A and B to triangular form:

$$U_1^T A Q_1 = \begin{array}{c} \\ k \\ l \\ m-k-l \end{array} \!\!\begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & A_{12} & A_{13} \\ 0 & 0 & A_{23} \\ 0 & 0 & 0 \end{pmatrix} \end{array}, \quad V_1^T B Q_1 = \begin{array}{c} \\ l \\ p-l \end{array} \!\!\begin{array}{c} \begin{array}{ccc} n-k-l & k & l \end{array} \\ \begin{pmatrix} 0 & 0 & B_{13} \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

where A_{12} and B_{13} are nonsingular and upper triangular, and A_{23} is upper triangular. If $m - k - l < 0$, then the bottom zero block of $U_1^T A Q_1$ does not appear, and A_{23} is upper trapezoidal. U_1, V_1 and Q_1 are orthogonal matrices. l is the rank of B, and $k + l$ is the rank of $\begin{bmatrix} A \\ B \end{bmatrix}$.

ii. **Compute GSVD of triangular matrices**

The generalized GSVD of two $l \times l$ upper triangular matrices A_{23} and B_{13} is computed using `STGSJA`, which uses a Jacobi-like method [10, 77]:

$$A_{23} = U_2 C R Q_2^T \quad \text{and} \quad B_{13} = V_2 S R Q_2^T.$$

Here, U_2, V_2 and Q_2 are orthogonal matrices, C and S are both real nonnegative diagonal matrices satisfying $C^2 + S^2 = I$, S is nonsingular, and R is upper triangular and nonsingular.

The reduction to triangular form, performed by `SGGSVP`, uses QR decomposition with column pivoting for numerical rank determination [11].

In fact, what `STGSJA` does is to use Jacobi rotations such that the rows of $U_2^T A_{23} Q_2$ are parallel to the corresponding rows of $V_2^T B_{13} Q_2$. To compute how parallel two vectors x and y are, we define a measure $par(x, y)$ to be the smallest singular value of the $n \times 2$ matrix (x, y). In `STGSJA`, $par(x, y)$ is computed by a small subroutine `SLAPLL`, which does a QR decomposition of (x, y) using `SLARFG`, and then computes the smaller singular value of the resulting 2×2 upper triangular matrix using `SLAS2`.

The stopping criterion in `STGSJA` is:

$$par(A_i, B_i) \leq n \cdot \min(tolA, tolB). \tag{6.3.4}$$

where A_i and B_i are i-th row of A and B, i.e. if (6.3.4) is satisfied, then we consider A_i to parallel to B_i. $tolA$ and $tolB$ are defined as

$$\begin{aligned} tolA &= \max(m, n) \cdot \max(\|A\|, \eta) \cdot \varepsilon \\ tolB &= \max(p, n) \cdot \max(\|B\|, \eta) \cdot \varepsilon \end{aligned}$$

where η is the underflow threshold and ε is the machine precision.

From now on, we will consider square matrices A and B only, i.e. $m = n = p$. Thus the stopping criterion becomes:

$$par(A_i, B_i) \leq \varepsilon \cdot n^2 \cdot \min(\|A\|, \|B\|). \tag{6.3.5}$$

It is not hard to see that n^2 on the right hand of (6.3.5) might be too large for a stopping criterion when n is large. For example, if the entries of A and B are uniformly randomly distributed in $[-1, 1]$, and if $n = 500$, then in IEEE single precision, i.e. $\varepsilon \approx 1.1921 \times 10^{-7}$, the right hand side of (6.3.5) is approximately 8, which makes the stopping criterion meaningless.

We tested `SGGSVD` with the following three types of matrix pairs A and B:

Type 1 The elements of A and B are uniformly randomly distributed on $[-1, 1]$.

Type 2 $A = U \cdot D \cdot X$ and $B = V \cdot E \cdot X$, where U and V are orthogonal matrices generated from random matrices[1], X is a random matrix whose elements are uniformly distributed on $[-1, 1]$, D and E are diagonal matrices, for some i, D_{ii}/E_{ii} is quite large, for some other i, D_{ii}/E_{ii} is quite small, and for the other i, D_{ii}/E_{ii} is $O(1)$.

Type 3 Almost as the same as **Type 2** except that X is produced randomly with geometrically distributed singular values.

Given the GSVD of A and B:

$$U^T \cdot A \cdot Q = \Sigma_1 \cdot R \quad \text{and} \quad V^T \cdot B \cdot Q = \Sigma_2 \cdot R,$$

we define the residuals

$$resA = \frac{\|U^T \cdot A \cdot Q - \Sigma_1 \cdot R\|}{\varepsilon \cdot \|A\| \cdot \sqrt{n}}, \quad resB = \frac{\|U^T \cdot B \cdot Q - \Sigma_2 \cdot R\|}{\varepsilon \cdot \|B\| \cdot \sqrt{n}}, \tag{6.3.6}$$

and

$$orthU = \frac{\|U^T \cdot U - I\|}{\varepsilon \cdot \sqrt{n}}, \quad orthV = \frac{\|V^T \cdot V - I\|}{\varepsilon \cdot \sqrt{n}}, \quad orthQ = \frac{\|Q^T \cdot Q - I\|}{\varepsilon \cdot \sqrt{n}}. \tag{6.3.7}$$

Table 6.1 shows the residuals of the GSVD computed by `sggsvd`. We can see that as n increases, the residuals of A and B become very large, which shows that the stopping criterion may not be appropriate when matrix size is fairly large.

After observing the original stopping criterion is not appropriate when n is fairly large, we change the stopping criterion to:

$$par(A_i, B_i) \leq \min(tolA, tolB) = \varepsilon \cdot n \cdot \min(\|A\|, \|B\|). \tag{6.3.8}$$

We did the computation again, table 6.2 shows the results.

From Table 6.2, we can notice that residuals are much smaller with the modified stopping criterion, however, it takes more CPU time. For example, for matrix pairs of Type 1, it runs almost twice as slowly as the original code. It is not surprising since to satisfy the more rigorous stopping criterion, it takes more Jacobi sweeps to converge. For matrices of Type 1, it takes twice as many Jacobi sweeps to converge.

For three different types of matrix pairs and for both original and the modified stopping criteria, figure 6.1 plots max($resA$, $resB$) versus n, figure 6.2 plots max($orthU$,$orthV$, $orthQ$) versus n and figure 6.3 compares the timing.

Types	dimension	50	100	200	300	400	500
Type 1	Time(secs)	3.7E-1	2.3E0	1.8E1	9.4E1	3.0E2	6.2E2
	$resA$	5.6E1	3.8E2	1.7E2	1.3E4	1.0E4	9.3E3
	$resB$	7.6E1	8.1E1	2.7E2	1.1E4	1.2E4	9.6E3
	$orthU$	4.7E1	7.4E1	1.3E2	1.0E2	1.1E2	1.4E2
	$orthV$	4.8E1	7.3E1	1.2E2	1.0E2	1.2E2	1.5E2
	$orthQ$	4.7E1	6.8E1	1.1E2	1.2E2	1.5E2	1.7E2
Type 2	Time(secs)	2.8E-1	1.6E0	1.2E1	6.1E1	1.6E2	1.9E2
	$resA$	4.4E1	2.2E3	3.9E3	5.6E3	4.2E4	8.2E4
	$resB$	4.9E1	6.9E1	9.7E1	1.2E2	2.4E2	6.5E1
	$orthU$	4.3E1	6.4E1	1.1E2	1.3E2	9.7E1	7.4E1
	$orthV$	3.6E1	5.9E1	1.0E2	1.3E2	9.4E1	6.7E1
	$orthQ$	4.2E1	6.2E1	1.0E2	1.3E2	1.3E2	1.3E2
Type 3	Time(secs)	1.7E-1	8.6E-1	4.2E0	9.2E0	2.1E1	4.8E1
	$resA$	4.0E1	6.9E1	4.4E2	2.0E4	1.7E4	1.4E4
	$resB$	3.7E1	8.0E1	2.5E2	1.0E4	4.7E3	3.3E3
	$orthU$	3.6E1	5.2E1	7.1E1	6.3E1	7.1E1	8.0E1
	$orthV$	3.4E1	4.8E1	7.0E1	5.5E1	6.1E1	6.8E1
	$orthQ$	3.5E1	4.9E1	7.1E1	7.5E1	9.0E1	1.0E2

Table 6.1: Speed and Accuracy of `SGGSVD` on RS6000 with ESSL BLAS (LDA = 501) with Original Stopping Criterion

Types	dimension	50	100	200	300	400	500
Type 1	Time(secs)	5.0E-1	3.0E0	2.4E1	1.9E2	6.0E2	1.2E3
	$resA$	7.5E1	9.7E1	1.5E2	1.8E2	2.0E2	2.2E2
	$resB$	7.8E1	1.0E2	1.5E2	1.8E2	2.0E2	2.3E2
	$orthU$	6.5E1	1.0E2	1.7E2	2.3E2	2.7E2	3.3E2
	$orthV$	6.5E1	1.0E2	1.7E2	2.3E2	2.7E2	3.5E2
	$orthQ$	5.9E1	8.8E1	1.4E2	1.8E2	2.2E2	2.5E2
Type 2	Time(secs)	2.8E-1	2.1E0	1.5E1	1.0E2	3.2E2	8.8E2
	$resA$	4.4E1	7.5E1	1.1E2	1.5E2	1.5E2	2.0E2
	$resB$	4.9E1	9.0E1	1.3E2	1.9E2	2.7E2	2.4E2
	$orthU$	4.3E1	9.0E1	1.5E2	2.4E2	2.3E2	3.8E2
	$orthV$	3.6E1	8.4E1	1.4E2	2.3E2	2.0E2	3.4E2
	$orthQ$	4.2E1	7.6E1	1.2E2	1.7E2	1.8E2	2.4E2
Type 3	Time(secs)	1.6E-1	1.1E0	5.4E0	1.6E1	3.8E1	8.2E1
	$resA$	4.0E1	7.5E1	1.2E2	2.2E2	2.6E2	3.3E2
	$resB$	3.7E1	8.3E1	2.5E2	3.5E2	3.8E2	4.5E2
	$orthU$	3.6E1	7.0E1	9.7E1	1.2E2	1.4E2	1.5E2
	$orthV$	3.4E1	6.6E1	9.6E1	1.2E2	1.3E2	1.5E2
	$orthQ$	3.5E1	5.9E1	8.5E1	1.0E2	1.2E2	1.4E2

Table 6.2: Speed and Accuracy of `SGGSVD` on RS6000 with ESSL BLAS (LDA = 501) with Modified Stopping Criterion

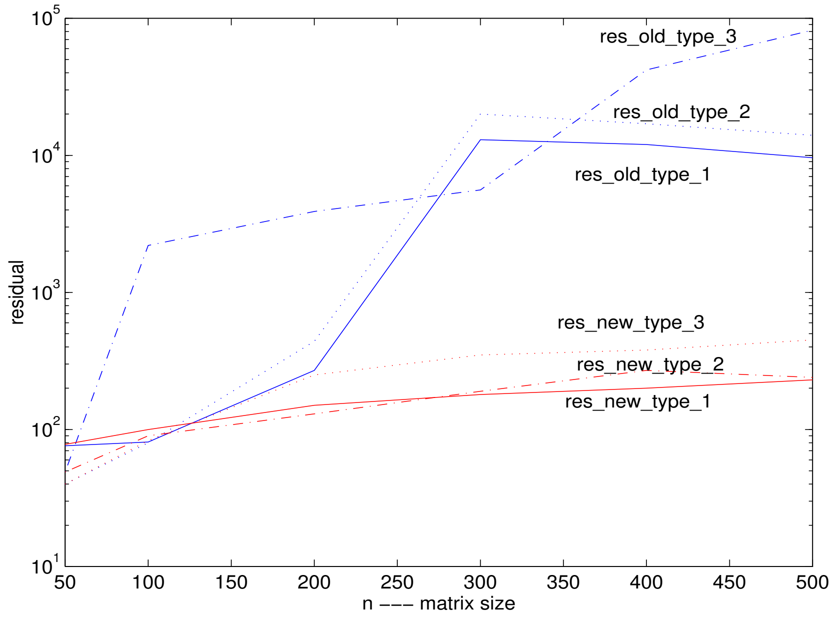


Figure 6.1: $\max(resA, resB)$ versus n

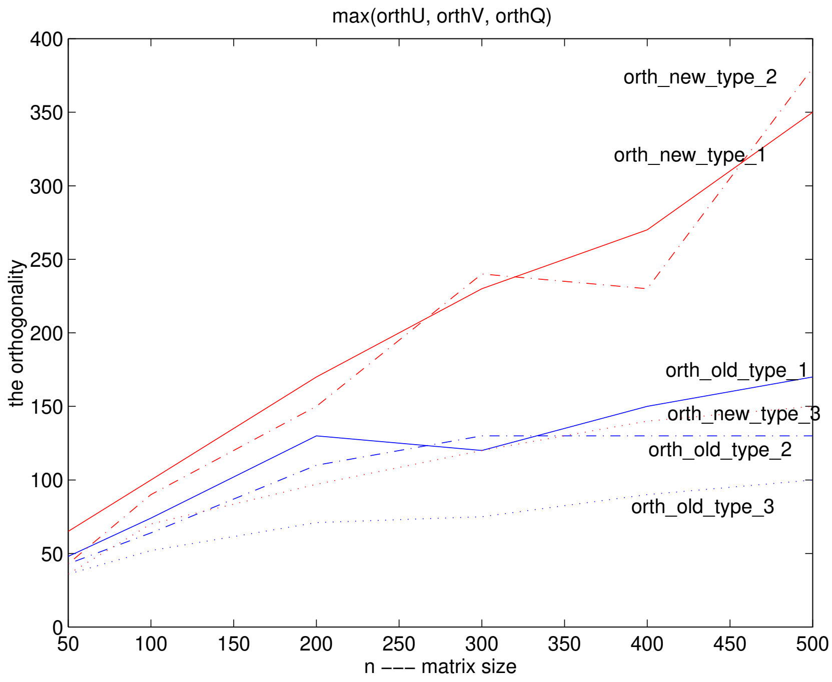


Figure 6.2: $\max(orthU, orthV, orthQ)$ versus n

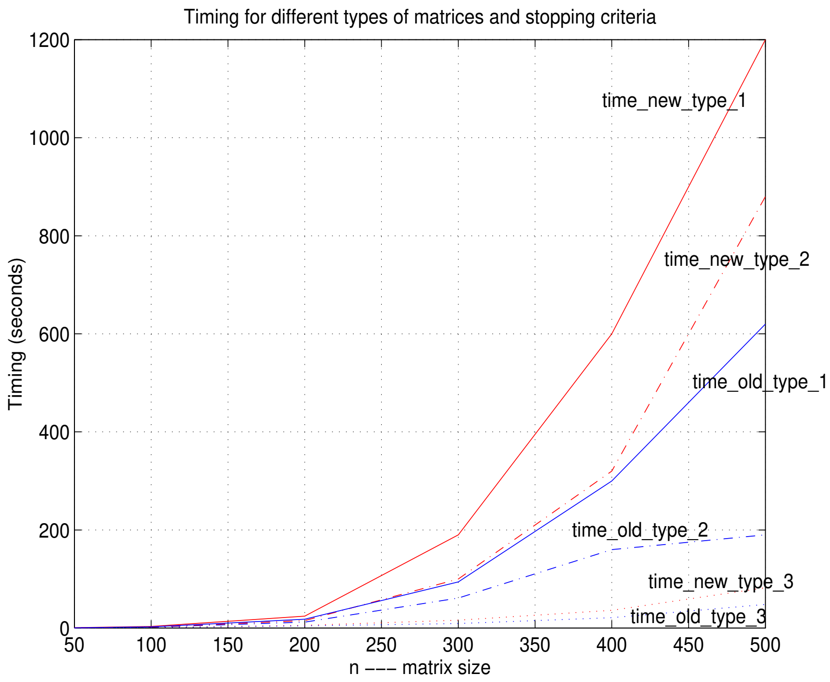


Figure 6.3: Timing of SGGSVD for different types of matrices and stopping criteria

From our numerical experiments, we found that with the new stopping criterion, $resA$ and $resB$ are mainly produced by orthogonal transformations (include Jacobi rotations), *not* by the stopping criterion. To illustrate this, we need to introduce some notations.

Since the preprocessing is done by QR decomposition, which is a very stable process, we only consider how large the error can be in STGSJA, the major computation subroutine. Let A and B be the input matrices to STGSJA, i.e. A and B are upper triangular matrices, and let $\hat{A}$ and $\hat{B}$ be the transformed matrices satisfying the stopping criterion after several Jacobi sweeps, and U, V, Q be the orthogonal matrices accumulated by Jacobi rotations. i.e.

$$U^T \cdot A \cdot Q = \hat{A} \quad \text{and} \quad V^T \cdot B \cdot Q = \hat{B}.$$

In STGSJA, $\hat{A}$ and $\hat{B}$ are considered to be "parallel", i.e. their corresponding rows are parallel, and after the postprocessing, $\hat{A} = C \cdot R$ and $\hat{B} = S \cdot R$ where C and S are diagonal matrices in (6.2.2) and (6.2.3). In principle, as long as the perturbations to make the corresponding rows exactly parallel are smaller than the error produced by Jacobi rotations,

[1]First, we generate a random matrix G whose elements are uniformly distributed on $[-1, 1]$, then we do a QR Decomposition of G, i.e. $G = Q \cdot R$. We then let $U = Q$.

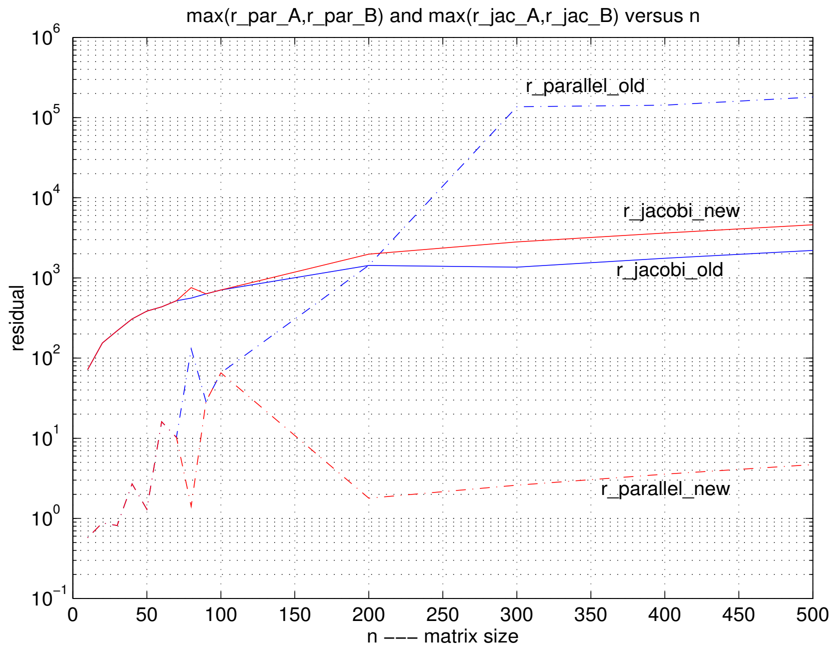


Figure 6.4: $\max(r^A_{\text{parallel}}, r^B_{\text{parallel}})$ and $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$ versus n, the residual is measured in ulps

the stopping criterion is a satisfactory one.

Let

$$r^A_{\text{jacobi}} = \frac{\|U^T \cdot A \cdot Q - \hat{A}\|}{\varepsilon \cdot \|A\|}, \quad \text{and} \quad r^B_{\text{jacobi}} = \frac{\|U^T \cdot B \cdot Q - \hat{B}\|}{\varepsilon \cdot \|B\|}.$$

and

$$r^A_{\text{parallel}} = \frac{\|\hat{A} - C \cdot R\|}{\varepsilon \cdot \|A\|}, \quad \text{and} \quad r^B_{\text{parallel}} = \frac{\|\hat{B} - S \cdot R\|}{\varepsilon \cdot \|B\|}$$

We only show the results for Type 1 here, since the other two types are similar. For both the original stopping criterion and the modified one, for $n = 10$, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, 300, 400, 500, figure 6.4 plots $\max(r^A_{\text{parallel}}, r^B_{\text{parallel}})$ and $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$ versus n with a log scale for the vertical axis, figure 6.6 plots the tolerance of the stopping criteria $\varepsilon n^2 \min(\|A\|, \|B\|)$ (old) and $\varepsilon n \min(\|A\|, \|B\|)$ (new), and $\max_i(par(A_i, B_i))$ after stopping criterion has been satisfied versus n, figure 6.5 plots $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$ versus n, and figure 6.7 shows the Jacobi sweeps need to be taken. The algorithm with original stopping criterion is represented by blue lines, and the algorithm with modified stopping criterion is represented by red lines.

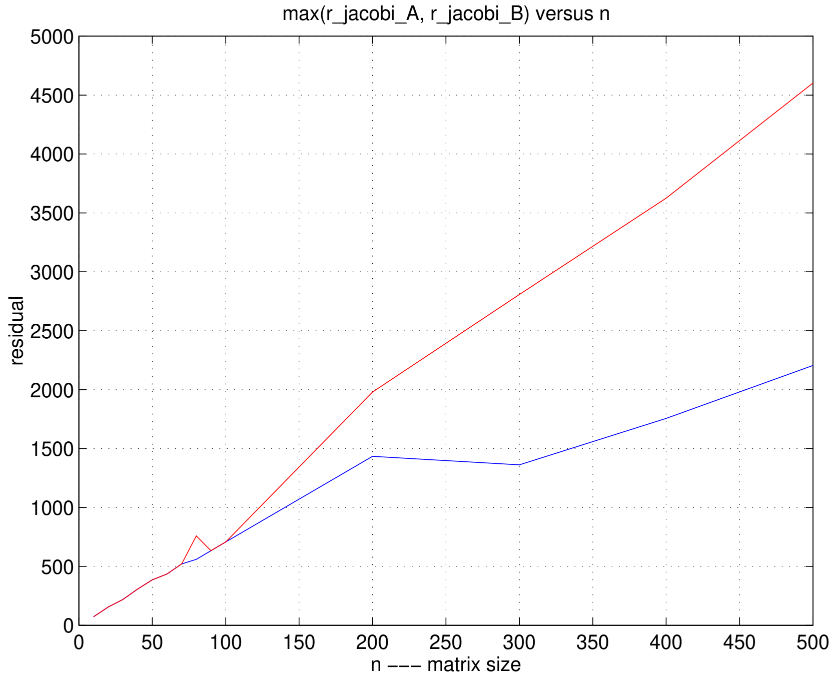


Figure 6.5: $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$ versus n, red—new, blue—old, the residual is measured in ulps

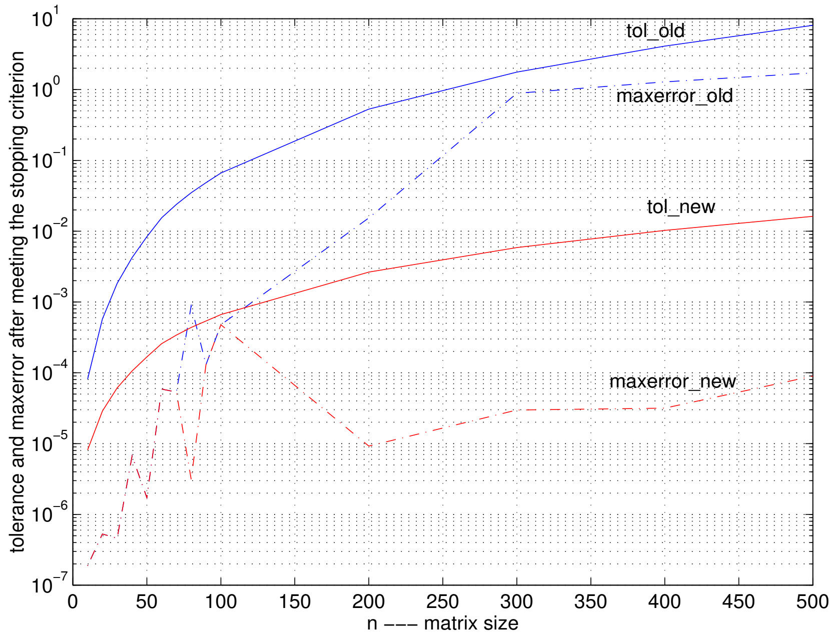


Figure 6.6: Tolerance and $\max_i((par(A_i, B_i))$ versus n

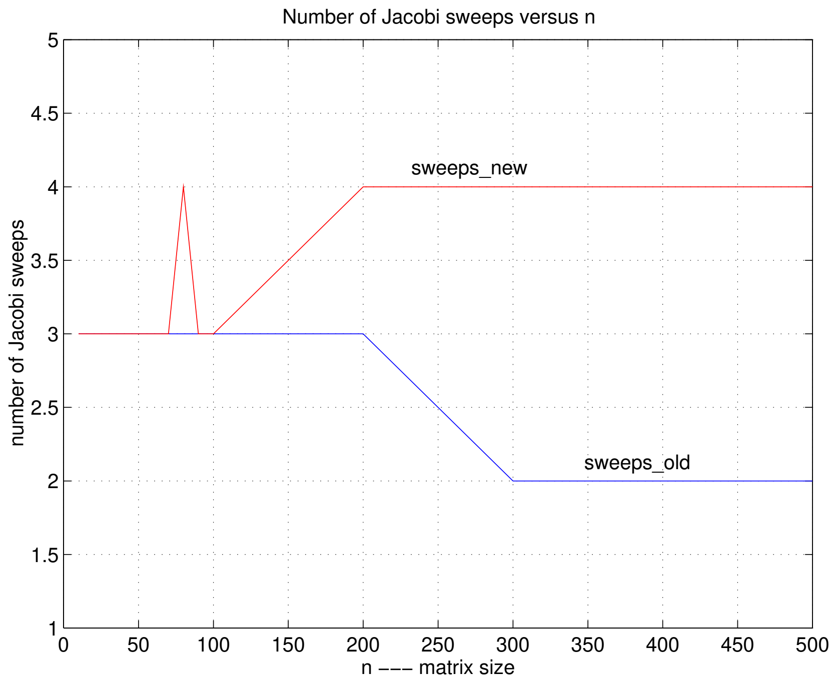


Figure 6.7: Number of the Jacobi Sweeps versus n

From figure 6.4, we can see that with the new stopping criterion, $\max(r^A_{\text{parallel}}, r^B_{\text{parallel}})$ is well below $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$, indicating that the orthogonal transformation is the major contributor to the residual. In contrast, with the old stopping criterion, $\max(r^A_{\text{parallel}}, r^B_{\text{parallel}})$ is much larger than $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$. Figure 6.5 plots $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$ in a normal scale, which indicates that $\max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}})$ is of $O(n)$, therefore, the error produced from Jacobi rotations is $O(\varepsilon \cdot n \cdot \max(\|A\|, \|B\|))$.

Another issue about the stopping criterion (for both the old one and the new one) is $\min(\|A\|, \|B\|)$ on the right hand side. If $\|B\|$ is significantly smaller than $\|A\|$, then the stopping criterion may never be met. The reason is that we cannot guarantee the relative accuracy of the smallest singular value of (A_i, B_i). An alternative approach is to use[9, 10]

$$par(\frac{A_i}{\|A_i\|}, \frac{B_i}{\|B_i\|}) \leq \tau$$

where A_i and B_i are the i-th rows of A and B and τ is some tolerance. However, this stopping criterion may be too strict in some cases, which results in taking unnecessarily more Jacobi sweeps to converge and making the code to run much longer (see Section 6.5). So we propose to scale the matrix pairs A and B such that $O(\|A\|) \approx O(\|B\|)$ at the first step of preprocessing, it will only take time of $O(n^2)$ whereas one Jacobi sweep takes time of $O(n^3)$.

6.4 Postprocessing

After modifying the stopping criterion to the new one, STGSJA still can get very inaccurate results even after we scale the matrix in some cases. The problem is with the postprocessing. In the original code, the postprocessing is done as follows:

Algorithm 6.4.1 Postprocessing of STGSJA

Input: *two $n \times n$ upper triangular matrices A and B whose corresponding rows are considered to be parallel.*

Output: *The diagonal matrices Σ_1 and Σ_2, and upper triangular matrix R such that $A = \Sigma_1 \cdot R$ and $B = \Sigma_2 \cdot R$ and $\Sigma_1 = \text{diag}\ (\alpha_i)$ and $\Sigma_2 = \text{diag}\ (\beta_i)$ where α_i/β_i are generalized singular values.*

do $i = 1, n$
    $a_1 = A_{ii}$
    $b_1 = B_{ii}$
    if $(a_1 \neq 0)$ then
        $\gamma = b_1/a_1$
        / change sign if necessary */*
        if $(\gamma < 0)$ then
            / change the sign of i-th row of B */*
            $B(i,:) = -B(i,:)$
            / change the sign of i-th column of V which is the orthogonal matrix in GSVD, see (6.2.1) */*
            $V(i,:) = -V(i,:)$
        end if
        Compute α_i and β_i such that $\gamma = \beta_i/\alpha_i$ and $\alpha_i^2 + \beta_i^2 = 1$.
        / produce upper triangular matrix R */*
        if $(\alpha_i \geq \beta_i)$
            $R(i,:) = A(i,:)/\alpha_i$
        else

$R(i,:) = B(i,:)/\beta_i$

end if

else /* $a_1 = A_{ii} = 0$ */

$\alpha_i = 0$

$\beta_i = 1$

$R(i,:) = B(i,:)$

end if

end do

What the postprocessing does is loop over the rows of A and B, and for the i-th rows A_i and B_i, mistakenly compares only the diagonal elements A_{ii} and B_{ii} instead of the corresponding rows. If $A_{ii} \neq 0$, it computes α_i and β_i such that $\beta_i/\alpha_i = B_{ii}/A_{ii}$, $\alpha_i^2 + \beta_i^2 = 1$, and $\alpha_i, \beta_i \geq 0$ (α_i/β_i is the generalized singular value). To make $\alpha_i, \beta_i \geq 0$ when $B_{ii}/A_{ii} < 0$, the code changes the sign of B_i, and at the same time, changes the sign of a column of V correspondingly to keep consistency, where V is the orthogonal matrix in GSVD (see (6.2.1)). The i-th row of R, R_i, is computed as A_i/α_i if $\alpha_i \geq \beta_i$ or computed as B_i/β_i otherwise.

Clearly, if A_{ii} and B_{ii} are significantly smaller than the other entries of the same rows, then α_i and β_i could be very inaccurate, thus produces the big errors in R_i. Also, the sign of B_{ii}/A_{ii} is not accurate enough to tell the sign difference of the corresponding rows. We constructed an example such that `STGSJA` fails due to the inappropriate postprocessing(see figure 6.8).

We propose two changes to fix the postprocessing problem:

- How to compute α_i and β_i.
  Instead of using the diagonal entries, we use the norms, i.e. we compute α_i and β_i such that $\alpha_i/\beta_i = \|A_i\|/\|B_i\|$, $\alpha_i^2 + \beta_i^2 = 1$, and $\alpha_i \geq 0$, $\beta_i \geq 0$.

- How to decide to change the sign of B_i.
  Instead of using the ratio of diagonal entries B_{ii}/A_{ii} to decide whether we should change the sign, we use the inner product of A_i and B_i, denoted by $< A_i, B_i >$: we don not change the sign unless $< A_i, B_i >$ is less than 0.

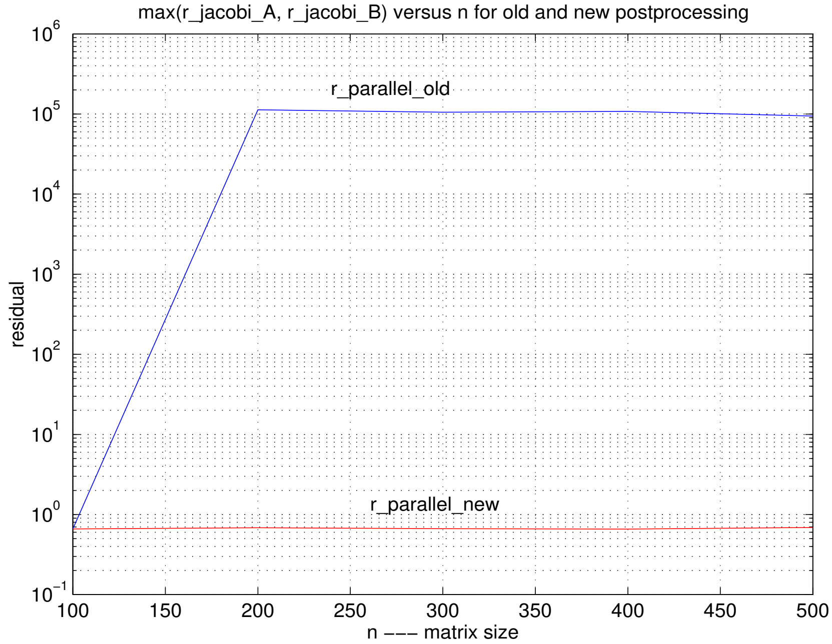


Figure 6.8: $\max(r^A_{\text{parallel}}, r^B_{\text{parallel}})$ versus n for old and new postprocessing

Figure 6.8 plots the $\max(r^A_{\text{parallel}}, r^B_{\text{parallel}})$ for the example we constructed, for both original postprocessing and modified one. The matrices we constructed are two random upper triangular matrices A and B, the offdiagonal elements of A are uniformly distributed on $[-1, 1]$, the diagonals are 10ε. B is the same as A except the diagonals of B are twice as large as those of A.

6.5 Stability of `STGSJA`

In this section, we prove the backward stability of `STGSJA`. We assume the scaling is done in preprocessing, and we ignore the errors resulting from the preprocessing since it uses a stable algorithm — QR decomposition. In other words, we assume the input matrix pair (A, B) are upper triangular matrices and therefore, we will only do error analysis for `STGSJA`. We also assume $O(\|A\|) \approx O(\|B\|)$.

As in section 6.3, we denote the transformed upper triangular matrices after the stopping criterion being satisfied by $\hat{A}$ and $\hat{B}$. U, V and Q are orthogonal matrices accumulated from Jacobi rotations and satisfying:

$$U^T \cdot A \cdot Q = \hat{A} \quad \text{and} \quad V^T \cdot B \cdot Q = \hat{B},$$

where the i-th rows of $\hat{A}$ and $\hat{B}$, $\hat{A}_i$ and $\hat{B}_i$, are considered to be parallel because

$$par(\hat{A}_i, \hat{B}_i) \le \varepsilon \cdot n \cdot \min(\|A\|, \|B\|)$$

for $i = 1, 2, \ldots, n$. So after postprocessing, we can write $\hat{A} = \Sigma_1 \cdot R$ and $\hat{B} = \Sigma_2 \cdot R$, where R is an upper triangular matrix, Σ_1 and Σ_2 are nonnegative diagonal matrices and $\Sigma_1^2 + \Sigma_2^2 = I$.

As we already observed in Section 6.3, r^A_{jacobi} and r^B_{jacobi} are $O(\varepsilon \cdot n \cdot \min(\|A\|, \|B\|))$, independent of the stopping criterion. So what we are really concerned about is whether the residuals r^A_{parallel} and r^B_{paralel} would be small if the new stopping criterion (6.3.8) is satisfied.

From the construction of the Algorithm 6.4.1 and by the properties of norms [43], we know that

$$r^A_{\text{parallel}} \le g(n) \max_i \frac{\|A_i - \alpha_i/\beta_i \cdot B_i\|}{\varepsilon \cdot \|A\|} = g(n) \max_{i \text{ with } R_i \text{ is constructed from } B_i} \frac{\|A_i - \alpha_i/\beta_i \cdot B_i\|}{\varepsilon \cdot \|A\|}, \tag{6.5.9}$$

where $\alpha_i/\beta_i \le 1$, and

$$r^B_{\text{parallel}} \le g(n) \max_i \frac{\|B_i - \beta_i/\alpha_i \cdot A_i\|}{\varepsilon \cdot \|B\|} = g(n) \max_{i \text{ with } R_i \text{ is constructed from } A_i} \frac{\|B_i - \beta_i/\alpha_i \cdot A_i\|}{\varepsilon \cdot \|B\|}, \tag{6.5.10}$$

where $\alpha_i/\beta_i > 1$. Here $g(n)$ is a low order polynomial in n.

We can only consider r^A_{parallel}, since r^B_{parallel} is similar. Let $\tilde{r}^A_{\text{parallel}} = \varepsilon \cdot r^A_{\text{parallel}}$, therefore

$$\tilde{r}^A_{\text{parallel}} = \max_{i \text{ with } R_i \text{ is constructed from } B_i} \frac{\|A_i - \alpha_i/\beta_i \cdot B_i\|}{\|A\|} \tag{6.5.11}$$

From the previous analysis, if $\tilde{r}^A_{\text{parallel}}$ is of $O(\varepsilon \cdot n)$, we know we have the backward stability, and the backward error is $O(\varepsilon \cdot n \cdot \|A\|)$.

For different i's, there are four cases to be considered:

- $\|A_i\|, \|B_i\|$ are both very tiny compared with $\|A\|$, i.e. $\|A_i\|, \|B_i\| = O(\varepsilon \cdot n \cdot \|A\|)$. This is an easy case since

$$\frac{\|A_i - \alpha_i/\beta_i \cdot B_i\|}{\|A\|} \le \frac{\|A_i\| + \alpha_i/\beta_i \cdot \|B_i\|}{\|A\|} \le \frac{\|A_i\| + \|B_i\|}{\|A\|} = O(\varepsilon \cdot n)$$

- $\|A_i\| = O(\varepsilon \cdot n \cdot \|A\|)$, $\|B_i\|$ is not tiny compared to $\|A\|$.

  By our new postprocessing, we know that $\alpha_i/\beta_i = \|A_i\|/\|B_i\|$, hence,

$$\frac{\|A_i - \alpha_i/\beta_i \cdot B_i\|}{\|A\|} \le \frac{\|A_i\| + \alpha_i/\beta_i\|B_i\|}{\|A\|} = \frac{\|A_i\| + \|A_i\|/\|B_i\| \cdot \|B_i\|}{\|A\|} = \frac{2\|A_i\|}{\|A\|} = O(\varepsilon \cdot n)$$

- $\|B_i\|$ is tiny whereas $\|A_i\|$ is not.

  This is impossible since we know that $\|A_i\|/\|B_i\| = \alpha_i/\beta_i \le 1$.

- Neither A_i nor B_i is tiny compared to $\|A\|$.

  Let the QR decomposition of (A_i, B_i) be

$$(A_i, B_i) = Q \cdot R = Q \cdot \begin{bmatrix} r_{11} & r_{12} \\ 0 & r_{22} \end{bmatrix}, \tag{6.5.12}$$

let $\lambda_{\min}$ be the smallest eigenvalue of $R^T R$, and $\sigma_{\min}$ be the smallest singular value of R. Thus,

$$\sigma_{\min}^2 = \lambda_{\min} = \frac{a + b - \sqrt{(a+b)^2 - 4(ab - c^2)}}{2}, \tag{6.5.13}$$

where $a = r_{11}^2$, $b = r_{12}^2 + r_{22}^2$ and $c = r_{11}r_{12}$. Let $\delta = \varepsilon \cdot n \cdot \|A\|$, so from the stopping criterion, we know that

$$\sigma_{\min} = par(A_i, B_i) \le \varepsilon \cdot n \cdot \min(\|A\|, \|B\|) \le \delta. \tag{6.5.14}$$

From (6.5.13) and (6.5.14), we have

$$\lambda_{\min} = \frac{a + b - \sqrt{(a+b)^2 - 4(ab - c^2)}}{2} \le \delta^2.$$

Therefore,

$$a + b - \sqrt{(a+b)^2 - 4(ab - c^2)} \le 2\delta^2$$

Equivalently,

$$a + b - 2\delta^2 \le \sqrt{(a+b)^2 - 4(ab - c^2)}$$

Take squares on both sides,

$$(a+b)^2 - 4\delta^2(a+b) + 4\delta^4 \le (a+b)^2 - 4(ab - c^2)$$

Hence

$$ab - c^2 \le (a+b)\delta^2 + \delta^4. \tag{6.5.15}$$

To simplify the computation, from now on, we will use 2-norm. By the invariance of $\|\cdot\|_2$ under orthogonal transformation, we have

$$a = r_{11}^2 = \|A_i\|_2^2, \quad b = r_{12}^2 + r_{22}^2 = \|B_i\|_2^2. \tag{6.5.16}$$

Let

$$\tilde{r}_{\text{parallel}}^{A_i} = \frac{\|A_i - \alpha_i/\beta_i \cdot B_i\|_2}{\|A\|_2},$$

since Q is an orthogonal matrix, therefore,

$$\begin{aligned}
\tilde{r}_{\text{parallel}}^{A_i} &= \frac{\|Q\{\begin{bmatrix} r_{11} \\ 0 \end{bmatrix} - \frac{\alpha_i}{\beta_i}\begin{bmatrix} r_{12} \\ r_{22} \end{bmatrix}\}\|_2}{\|A\|_2} \\
&= \frac{\|\begin{bmatrix} r_{11} \\ 0 \end{bmatrix} - \frac{\alpha_i}{\beta_i}\begin{bmatrix} r_{12} \\ r_{22} \end{bmatrix}\|_2}{\|A\|_2}.
\end{aligned}$$

To show $\tilde{r}_{\text{parallel}}^{A_i}$ is tiny, we need

Lemma 6.5.1 *In the QR Decomposition (6.5.12) of (A_i, B_i), if $r_{11} = \|A_i\|$, with the new postprocessing, we can conclude $r_{12} \geq 0$. Otherwise, if $r_{11} = -\|A_i\|$, then $r_{12} \leq 0$.*

Proof. We only prove the lemma for the case when $r_{11} = \|A_i\|$, the other case can be proved similarly. Let $x = A_i$ and $y = B_i$. Assume the Householder orthogonal matrix is

$$Q = I - 2u \cdot u^T.$$

where the Householder vector is

$$u = \frac{x - \|x\|_2 e_1}{\|x - \|x\|_2 e_1\|_2},$$

where $e_1 = (1, 0, \ldots, 0)^T$. Therefore we can write Q as

$$Q = I - \frac{2}{\|x - \|x\|_2 e_1\|_2^2} a a^T,$$

where $a = x - \|x\|_2 e_1$. We know that $(Qx)^T = (\|x\|_2, 0, \ldots, 0)^T$ by the construction. Now we need to compute $(Qy)_1$ which equals to r_{12}.

$$Qy = y - \frac{2a^T y}{\|x - \|x\|_2 e_1\|_2^2} a = y - \frac{2(x^T y - \|x\|_2 y_1)}{\|x - \|x\|_2 e_1\|_2^2} a.$$

Therefore,

$$\begin{aligned} r_{12} &= (Qy)_1 = y_1 - \frac{2(x^Ty - \|x\|_2 y_1)}{\|x - \|x\|_2 e_1\|_2^2}(x_1 - \|x\|_2) \\ &= y_1 - \frac{2(x^Ty - \|x\|_2 y_1)}{\|x\|_2^2 - x_1\|x\|_2} = y_1 - \frac{2(x^Ty - \|x\|_2 y_1)}{\|x\|_2(\|x\|_2 - x_1)} \\ &= y_1 - \frac{\|x\|_2 y_1 - x^Ty}{\|x\|_2} = \frac{x^Ty}{\|x\|_2}. \end{aligned}$$

Our postprocessing guarantees that $x^Ty \geq 0$, therefore, $r_{12} \geq 0$. ▮

From now on, without loss of generality, we will assume $r_{11} = \|A_i\|_2$, hence $r_{12} \geq 0$. In order for $\tilde{r}^{A_i}_{\text{parallel}}$ to be tiny, the following quantity

$$\| \begin{bmatrix} r_{11} \\ 0 \end{bmatrix} - \frac{\alpha_i}{\beta_i} \begin{bmatrix} r_{12} \\ r_{22} \end{bmatrix} \|_2^2 = (r_{11} - \frac{\alpha_i}{\beta_i} r_{12})^2 + (\frac{\alpha_i}{\beta_i} r_{22})^2,$$

has to be tiny. From (6.5.16) and $c = r_{11}r_{12}$, (6.5.15) can be simplified to

$$r_{11}^2 r_{22}^2 \leq \delta^2(r_{11}^2 + r_{12}^2 + r_{22}^2) + \delta^4.$$

Or,

$$\|A_i\|_2^2 r_{22}^2 \leq \delta^2(\|A_i\|_2^2 + \|B_i\|_2^2) + \delta^4. \tag{6.5.17}$$

Therefore,

$$(\frac{\alpha_i}{\beta_i} r_{22})^2 = \frac{\|A_i\|_2^2 r_{22}^2}{\|B_i\|_2^2} \leq \delta^2(\frac{\|A_i\|_2^2}{\|B_i\|_2^2} + 1) + \frac{\delta^4}{\|B_i\|_2^2} \leq 3\delta^2,$$

where we use the facts that $\|A_i\|_2/\|B_i\|_2 = \alpha_i/\beta_i \leq 1$ and $\|B_i\|_2 \geq \delta = \varepsilon \cdot n \cdot \|A\|_2$, since otherwise if $\|B_i\|_2 < \varepsilon \cdot n \cdot \|A\|_2$, it is the case we previously discussed. By (6.5.17)

$$r_{22}^2 \leq \delta^2(1 + \frac{\|B_i\|_2^2}{\|A_i\|_2^2}) + \frac{\delta^4}{\|A_i\|_2^2}.$$

Therefore,

$$r_{12}^2 = \|B_i\|_2^2 - r_{22}^2 \geq \|B_i\|_2^2 - \delta^2(1 + \frac{\|B_i\|_2^2}{\|A_i\|_2^2}) - \frac{\delta^4}{\|A_i\|_2^2}.$$

Thus,

$$r_{11}^2 - \frac{\alpha_i^2}{\beta_i^2} r_{12}^2 \quad = \quad \|A_i\|_2^2 - \frac{\|A_i\|_2^2}{\|B_i\|_2^2} r_{12}^2 \leq \delta^2(\frac{\|A_i\|_2^2}{\|B_i\|_2^2} + 1) + \frac{\delta^4}{\|B_i\|_2^2} \leq 3\delta^2.$$

Again, we use the facts that $\|A_i\|_2/\|B_i\|_2 = \alpha_i/\beta_i \leq 1$ and $\|B_i\|_2 \geq \delta = \varepsilon \cdot n \cdot \|A\|_2$.

Hence,

$$(r_{11}-\frac{\alpha_i}{\beta_i}r_{12})^2=\frac{(r_{11}^2-\frac{\alpha_i^2}{\beta_i^2}r_{12}^2)^2}{(r_{11}+\frac{\alpha_i}{\beta_i}r_{12})^2}\le\frac{(3\delta^2)^2}{(r_{11}+\frac{\alpha_i}{\beta_i}r_{12})^2}\le\frac{9\delta^4}{r_{11}^2}=9\delta^2\frac{\delta^2}{\|A_i\|_2^2}\le 9\delta^2.$$

Here, we use the fact that $\|A_i\|_2\ge\delta=\varepsilon\cdot n\cdot\|A\|_2$.

Therefore,

$$\|\begin{bmatrix} r_{11}\\ 0\end{bmatrix}-\frac{\alpha_i}{\beta_i}\begin{bmatrix} r_{12}\\ r_{22}\end{bmatrix}\|_2^2=(r_{11}-\frac{\alpha_i}{\beta_i}r_{12})^2+(\frac{\alpha_i}{\beta_i}r_{22})^2\le 3\delta^2+9\delta^2=12\delta^2,$$

and

$$\tilde{r}^{A_i}_{\text{parallel}}=\frac{\|\begin{bmatrix} r_{11}\\ 0\end{bmatrix}-\frac{\alpha_i}{\beta_i}\begin{bmatrix} r_{12}\\ r_{22}\end{bmatrix}\|_2}{\|A\|_2}\le\frac{\sqrt{12}\delta}{\|A\|_2}=2\sqrt{3}\varepsilon\cdot n=O(\varepsilon\cdot n).$$

From the analysis for the above cases and equations (6.5.9) and (6.5.10), we infer the following theorem:

Theorem 6.5.1 *In* STGSJA*, with the input matrices A and B are scaled such that $O(\|A\|)\approx O(\|B\|)$, with the new stopping criterion and the new postprocessing, we have*

$$\max(r^A_{\text{parallel}},r^B_{\text{parallel}})=f(n).$$

where $f(n)$ is a low order polynomial in n.

Remark 6.5.1 As we mentioned in Section 6.3, theoretically, a much more rigorous stopping criterion is proposed,

$$par(\frac{A_i}{\|A_i\|},\frac{B_i}{\|B_i\|})\le\tau,$$

where τ is some tolerance. However, from the first case we discussed above, when $\|A_i\|$ and $\|B_i\|$ are very tiny compared to $\|A\|$ and $\|B\|$, we don not require the above rigorous stopping criterion to be satisfied to get backward stability. So if we only want the backward stability (what we normally can expect in general), the above rigorous stopping criterion is *not* practical.

Using classical error bounds for the error in a product of Givens rotations, it can be shown that

$$\max(r^A_{\text{jacobi}},r^B_{\text{jacobi}})\le\tilde{f}(n),$$

where $f(n)$ is a low degree polynomial in n, so the residual satisfies

$$\begin{aligned} & \max\left(\frac{\|U^T \cdot A \cdot Q - C \cdot R\|}{\varepsilon \cdot \|A\|}, \frac{\|V^T \cdot B \cdot Q - S \cdot R\|}{\varepsilon \cdot \|B\|}\right) \\ \leq \; & \max(r^A_{\text{jacobi}}, r^B_{\text{jacobi}}) + \max(r^A_{\text{parallel}}, r^B_{\text{parallel}}) \leq h(n), \end{aligned}$$

where $h(n) = f(n) + g(n)$ is a low degree polynomial in n. Therefore, we end the section with the following theorem:

Theorem 6.5.2 *For* `STGSJA`*, with the input matrices A and B are scaled such that $O(\|A\|) \approx O(\|B\|)$, with the new stopping criterion and the new postprocessing,* `STGSJA` *is backward stable.*

6.6 Van Loan's Algorithm Implemented by Divide-and-Conquer SVD

Van Loan's algorithm [96] is based on the observation that if a well-conditioned matrix has nearly orthogonal columns, then it can be safely diagonalized by the QR factorization [9]. The observation can be described in the following theorem:

Theorem 6.6.1 *(***Van Loan [96]***) Assume that the $m \times k$ matrix $Y = (y_1, y_2, \ldots, y_k)$ satisfies*

$$Y^T Y = D^2 + E$$

where $D = \text{diag}\ (\|y_1\|, \|y_2\|, \ldots, \|y_k\|)$, and let

$$Y = QR$$

be the QR factorization of Y, where $Q \in \mathcal{R}^{m \times k}$ is an orthogonal matrix, and $R \in \mathcal{R}^{k \times k}$ is upper triangular. Let Y_i be the first i columns of Y. Then for all i and j ($j > i$), we have

$$|R_{ij}| \leq \min\{\|y_j\|, \frac{\|E\|}{\sigma_{\min}(Y_i)}\}.$$

Given the matrix pair A and B which both have n columns, the first step of the algorithm is a preprocessing step to compute the QR decomposition of $G = \begin{bmatrix} A \\ B \end{bmatrix}$:

$$G = \begin{bmatrix} A \\ B \end{bmatrix} = QR = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R.$$

Then we compute the CSD of Q_1 and Q_2 [23, 92, 96]:

$$\begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} = \begin{bmatrix} U_1 & 0 \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} \Sigma_1 \\ \Sigma_2 \end{bmatrix} V^T,$$

where Σ_1 and Σ_2 are the nonnegative diagonal matrices satisfying $\Sigma_1^T\Sigma_1 + \Sigma_2^T\Sigma_2 = I$, and U_1, U_2, V are orthogonal matrices. Therefore,

$$A = U_1\Sigma_1V^TR, \quad B = U_2\Sigma_2V^TR.$$

If we want this decomposition to be of the form in (6.2.1), we can do a postprocessing step. We compute $W = V^TR$, and then compute the RQ factorization of W:

$$W = \tilde{R}\tilde{Q}.$$

In the second step, we search for a well-conditioned submatrix among Q_1 and Q_2 to do the diagonalization by the QR decomposition as in Theorem 6.6.1, and use the SVD to diagonalize the remaining ill-conditioned submatrix[9]. In more detail, we first compute the SVD of Q_2:

$$Q_2 = U_2SV_2^T$$

where U_2 and V_2 are orthogonal matrices, and S is diagonal whose elements are increasing

$$0 \le s_1 \le s_2 \le \cdots \le s_k \le \tau < s_{k+1} \le \cdots \le s_n,$$

and where τ is certain tolerance which can be specified by user.

Then we do a QR factorization of the product Q_1V_2:

$$Q_1V_2 = U_1R_1.$$

In exact arithmetic, since

$$(Q_1V_2)^T(Q_1V_2) = I - (Q_2V_2)^T(Q_2V_2) = I - S^TS,$$

by Theorem 6.6.1, R_1 would be a diagonal matrix. However, because of roundoff, we may only have

$$R_1 = \begin{bmatrix} \text{diag } (c_1, c_2, \ldots, c_k) & 0 \\ 0 & R_2 \end{bmatrix},$$

where R_2 is $n-k$ by $n-k$ and

$$R_1^TR_1 + S^TS = I.$$

By Theorem 6.6.1, the first k columns of R_1 correspond to "large" singular values of Q_1. Now compute the SVD of submatrix R_2,

$$R_2 = \tilde{U}_1 \text{diag } (c_{k+1}, \ldots, c_n) \tilde{V}_1^T,$$

and the QR factorization of $n-k$ by $n-k$ matrix $W_1 = D\tilde{V}_1$, where $D = \text{diag } (s_{k+1}, \ldots, s_n)$,

$$W_1 = \tilde{U}_2 R_3.$$

We then have

$$R_3 = \text{diag } (s_{k+1}, \ldots, s_n),$$

since $s_{k+1}, \ldots, s_n$ are "large". Combining all the previous steps, we have

$$\begin{aligned} Q_1 &= U_1 \begin{bmatrix} I & 0 \\ 0 & \tilde{U}_1 \end{bmatrix} \Sigma_1 (V_2 \begin{bmatrix} I & 0 \\ 0 & \tilde{V}_1 \end{bmatrix})^T, \\ Q_2 &= U_2 \begin{bmatrix} I & 0 \\ 0 & \tilde{U}_2 \end{bmatrix} \Sigma_2 (V_2 \begin{bmatrix} I & 0 \\ 0 & \tilde{V}_1 \end{bmatrix})^T, \end{aligned}$$

where $\Sigma_1 = \text{diag } (c_1, \ldots, c_n)$ and $\Sigma_2 = \text{diag } (s_1, \ldots, s_n)$. This is the desired CSD of Q_1 and Q_2.

Remark 6.6.1 The tolerance τ is chosen as the dividing threshold between the large and small singular values. When $\tau = 1/\sqrt{2}$, it minimizes a backward error bound [9]. One may wish to adjust this tolerance under certain circumstances since the overall amount of work depends on the size of the index k. Large k will result in smaller subproblems, and reduce the total amount of work, but may increase the backward error. In our implementation, we use $\tau = 1/\sqrt{2}$.

6.7 Performance of Van Loan's Algorithm

We implemented Van Loan's algorithm using our new divide-and-conquer SVD implementation `SBDSDC`(see chapter 5). We call our routine `SGGSDC`. We use the same three types of test matrices as in section 6.3. We ran all the tests in single precision using `SGGSVD` with the modified stopping criterion and the postprocessing as described in sections 6.3 and 6.4. As we mentioned there, with the new stopping criterion, `SGGSVD` is likely to run twice as slowly as the original `SGGSVD` because it takes almost twice as many Jacobi sweeps

Table 6.3: Speedup of SGGSDC over SGGSVD on RS6000

Speedup using ESSL BLAS						
	Dimension					
Test Matrix	50	100	200	300	400	500
type 1	5.10	8.38	13.16	34.55	50.00	54.55
type 2	4.83	8.80	10.67	23.81	34.41	52.35
type 3	2.43	3.93	3.63	3.78	3.98	4.89

to converge. The new postprocessing does not affect the performance since it takes $O(n^2)$ time. We ran the tests on an IBM RS6000/590 using ESSL BLAS for $n \times n$ matrix pairs with $n = 50, 100, 200, 300, 400, 500$.

Table 6.3 shows the speedup of SGGSDC over SGGSVD. Figure 6.9 shows the run time of SGGSDC and SGGSVD, figure 6.10 shows the residuals of GSVD $\max(resA, resB)$ and figure 6.11 shows the residual of orthogonality $\max(orthU, orthV, orthQ)$(see (6.3.6) and (6.3.7) for definitions of the residuals). In all the figures, we plot the data of SGGSVD by blue line, SGGSDC by red line; the data of matrices of Type 1 are plotted by solid line, data of Type 2 are plotted by dashdot line and data of Type 3 are plotted by dashed line.

We can see that SGGSDC achieves a solid speedup over SGGSVD. When $n = 500$, the speedup is over 50 for random matrix pairs (see table 6.3). Also, SGGSDC computes the GSVD more accurately, see figures 6.10 and figure 6.11.

Our implementation of Van Loan's algorithm needs $O((m+p)n)$ workspace whereas SGGSVD only needs $\max(3n, m, p) + n$ [2]. Therefore, we recommend to use Van Loan's algorithm with divide-and-conquer SVD to compute the GSVD of A and B if we have enough workspace; otherwise, we use SGGSVD. How to reduce the workspace needed by Van Loan's algorithm needs further investigation.

When we use Van Loan's algorithm to compute the GSVD of A and B, the prescaling of A and B such that $O(\|A\|) \approx O(\|B\|)$ before calling SGGSDC is necessary since otherwise if $\|B\| \ll \|A\|$, then when we do a QR factorization of $\begin{bmatrix} A \\ B \end{bmatrix}$, most information of B will be lost.

Let $\alpha > 0$ and $\beta > 0$ be two scalars such that $\|\alpha A\| = \|\beta B\|$ and the GSVD of αA

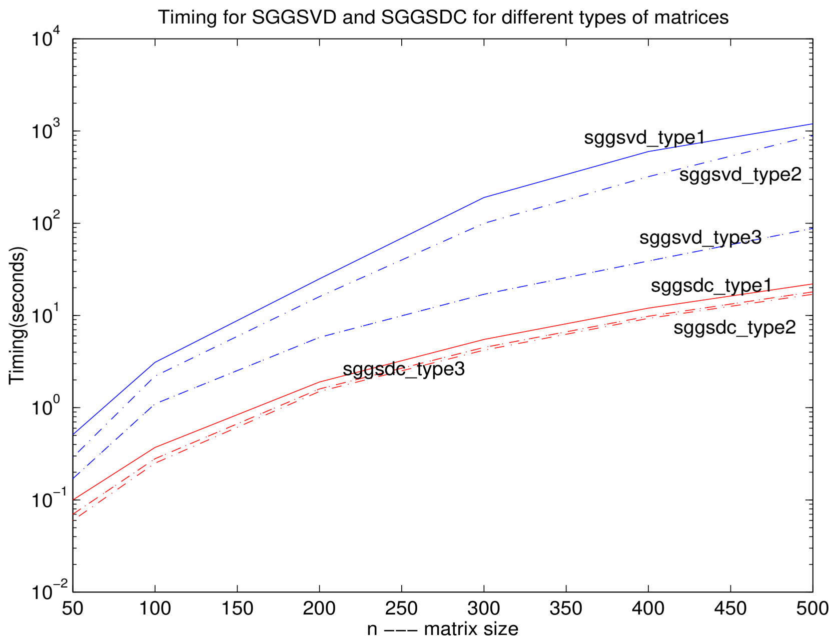


Figure 6.9: Timing of SGGSVD and SGGSDC for different types of matrices

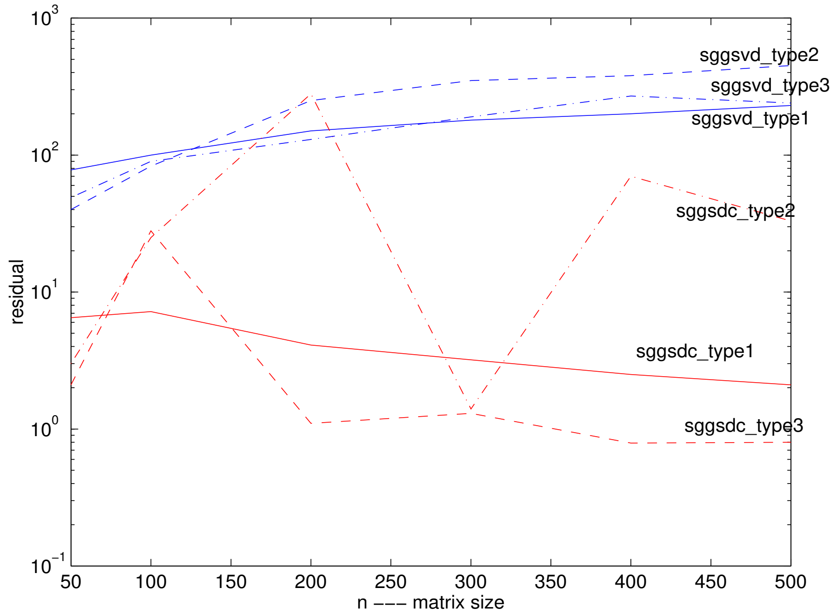


Figure 6.10: $\max(resA, resB)$ versus n

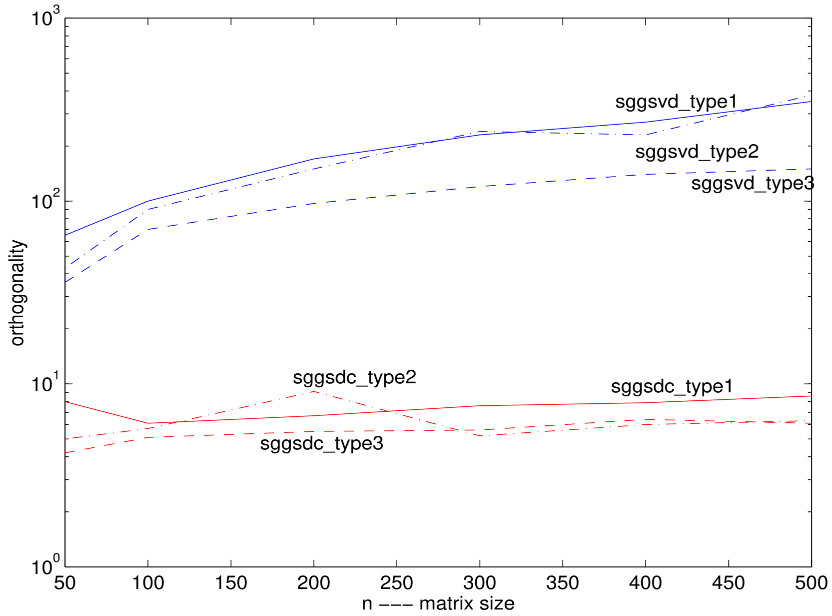


Figure 6.11: $\max(resA, resB)$ versus n

and βB is given by

$$\begin{aligned} \alpha A &= U\Sigma_1[0 \quad R]Q^T, \\ \beta B &= V\Sigma_2[0 \quad R]Q^T. \end{aligned}$$

Therefore,

$$\begin{aligned} A &= U(\frac{1}{\alpha}\Sigma_1)[0 \quad R]Q^T, \\ B &= V(\frac{1}{\beta}\Sigma_2)[0 \quad R]Q^T. \end{aligned}$$

Let

$$D = (\frac{1}{\alpha^2}\Sigma_1^T\Sigma_1 + \frac{1}{\beta^2}\Sigma_2^T\Sigma)^{\frac{1}{2}},$$

and

$$\hat{\Sigma}_1 = \frac{1}{\alpha}\Sigma_1 D^{-1}, \quad \hat{\Sigma}_2 = \frac{1}{\beta}\Sigma_2 D^{-1}.$$

Since

$$\hat{\Sigma}_1^T\hat{\Sigma}_1 + \hat{\Sigma}_2^T\hat{\Sigma}_2 = D^{-1}(\frac{1}{\alpha^2}\Sigma_1^T\Sigma_1 + \frac{1}{\beta^2}\Sigma_2^T\Sigma_2)D^{-1} = D^{-1}D^2D^{-1} = I,$$

the GSVD of A and B is then given by

$$\begin{aligned} A &= U\hat{\Sigma}_1[0 \quad DR]Q^T, \\ B &= V\hat{\Sigma}_2[0 \quad DR]Q^T. \end{aligned}$$

Chapter 7

Conclusions

In this thesis we have discussed a variety of algorithms for computing the eigendecomposition of a symmetric matrix, the singular value decomposition of a general matrix, and the generalized singular value decomposition for a pair of matrices. The main work concerns on the correctness, the stability and the efficiency of these algorithms.

In chapter 2, we discuss the correctness of the bisection algorithm for finding the eigenvalues of symmetric matrices. We focus on the function $\mathrm{Count}(x)$ which returns the number of eigenvalues less than x. We present examples to illustrate the incorrect implementations, and explain why they fail. We rigorously prove the correctness of several implementations, such as LAPACK's `DSTEBZ`.

In chapters 3 and 4, we discuss the parallel prefix algorithm which accelerates the bisection algorithm by reducing the complexity of $\mathrm{Count}(x)$ from $O(n)$ to $O(\log_2 n)$. We present numerical experiments to show the instability of the parallel prefix algorithm. We discuss its backward and forward error analysis, and discuss possible ways to improve its stability such as iterative refinement. Two problems remain open. The first is to find a tight bound on the forward error of the computed results by parallel prefix can be for a general symmetric tridiagonal matrix. The second is to find a cheap criterion to decide when the results computed by parallel prefix are too inaccurate to use.

In chapter 5, we discuss an implementation of a divide-and-conquer algorithm for computing the singular value decomposition. We have achieved good speedups over the previous LAPACK implementation using QR-iteration. We also compare the linear least squares solver based on our implementation of SVD with other solvers including plain QR, QR with pivoting, rank-revealing QR, etc. We show that the solver based on divide-and-conquer SVD (`xGELSD`) and the solver based on QR-iteration with "factored form" (`xGELSF`) make great improvements over the previous implementation in LAPACK. In fact, `xGELSF` runs a little faster than `xGELSD` in several cases, but it requires $O(n^2)$ storage in contrast to $O(n \log_2 n)$ for `xGELSD`. Therefore in the future LAPACK release, the SVD-based linear least squares solver should be based on `xGELSF` and `xGELSD`, with a switch such that when there is enough storage, use `xGELSF`; otherwise, we use `xGELSD`.

In chapter 6, we discuss two algorithms for computing the generalized singular value decomposition. We first discuss two improvements on LAPACK's implementation in order to maintain backward stability, and then we discuss a faster algorithm which we implemented using the divide-and-conquer SVD. Our implementation, `xGGSDC`, achieves good speedups over LAPACK's `xGGSVD`. However, it requires $O(n^2)$ storage whereas `xGGSVD` only needs $O(n)$. Therefore, the GSVD routine in the future LAPACK release should be based on `xGGSVD` and `xGGSDC`: when there is enough storage, use `xGGSDC`; otherwise, we use `xGGSVD`.

Bibliography

[1] R. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high performance numerical algorithms. *IBM J. of Research and Development*, 38(5):563–576, September 1994.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide (second edition)*. SIAM, Philadelphia, 1995. 324 pages.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Release 1.0*. SIAM, Philadelphia, 1992. 235 pages.

[4] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[5] ANSI/IEEE, New York. *IEEE Standard for Radix Independent Floating Point Arithmetic*, Std 854-1987 edition, 1987.

[6] P. Arbenz and G. Golub. On the spectral decomposition of Hermitian matrices modified by low rank perturbations with applications. *SIAM J. Matrix Anal. Appl.*, 9(1):40–58, January 1988.

[7] M. Arioli, J. Demmel, and I. S. Duff. Solving sparse linear systems with sparse backward error. *SIAM J. Matrix Anal. Appl.*, 10(2):165–190, April 1989.

[8] M. Assadullah, J. Demmel, S. Figueroa, A. Greenbaum, and A. McKenney. On finding eigenvalues and singular values by bisection. Unpublished Report.

[9] Z. Bai. The CSD, GSVD, their applications and computations. IMA Preprint 958, University of Minnesota, April 1992. Submitted to SIAM Rev.

[10] Z. Bai and J. Demmel. Computing the generalized singular value decomposition. *SIAM J. Sci. Comp.*, 14(6):1464–1486, November 1993. Available as all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-92-720.

[11] Z. Bai and H. Zha. A new preprocessing algorithm for the computation of the generalized singular value decomposition. *SIAM J. Sci. Comp.*, 14(4):1007–1012, 1993.

[12] J. Barlow and J. Demmel. Computing accurate eigensystems of scaled diagonally dominant matrices. *SIAM J. Num. Anal.*, 27(3):762–791, June 1990.

[13] C. Bischof and G. Quintana-Orti. Computing rank-revealing QR factorizations of dense matrices. Argonne Preprint ANL-MCS-P559-0196, Argonne National Laboratory, 1996.

[14] C. Bischof, X. Sun, and M. Marques. Parallel bandreduction and tridiagonalization. In R. Sincovec et al, editor, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume 1, pages 383–390. SIAM, 1993.

[15] A. Borodin and I. Munro. *The Computational Complexity of Algebraic and Numeric Problems.* American Elsevier, New York, 1975.

[16] J. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31(137):163–179, January 1977.

[17] P. A. Businger and G. Golub. Algorithm 358: Singular value decomposition of a complex matrix. *Comm. Assoc. Comput. Mach.*, 12:564–565, 1969.

[18] T. Chan. Rank revealing QR factorizations. *Lin. Alg. Appl.*, 88/89:67–82, 1987.

[19] S. Chandrasekaran and I. Ispen. On rank-revealing QR factorizations. *SIAM Journal on Matrix Analysis and Applications*, 15, 1994.

[20] S. C. Chen, D. J. Kuck, and A. H. Sameh. Practical parallel band triangular system solvers. *ACM Trans. Math. Software*, 4:270–277, 1978.

[21] J. J. M. Cuppen. The singular value decomposition in product form. *SIAM J. Sci. Stat. Comput.*, 4:216–221, 1983.

[22] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[23] C. Davis and W. Kahan. The rotation of eigenvectors by a perturbation iii. *SIAM J. Num. Anal.*, 7:248–263, 1970.

[24] P. Deift, J. Demmel, L.-C. Li, and C. Tomei. The bidiagonal singular values decomposition and Hamiltonian mechanics. *SIAM J. Num. Anal.*, 28(5):1463–1516, October 1991. (LAPACK Working Note #11).

[25] J. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.*, 5(4):887–919, Dec 1984.

[26] J. Demmel. Specifications for robust parallel prefix operations. Technical report, Thinking Machines Corp., 1992.

[27] J. Demmel. *Numerical Linear Algebra.* Lecture Notes, Mathematics Department, UC Berkeley, 1993.

[28] J. Demmel. Trading off parallelism and numerical stability. In G. Golub M. Moonen and B. de Moor, editors, *Linear Algebra for Large Scale and Real-Time Applications*, pages 49–68. Kluwer Academic Publishers, 1993. NATO-ASI Series E: Applied Sciences, Vol. 232; Available as all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/csd/csd-92-702.

[29] J. Demmel, 1995. private communication.

[30] J. Demmel, I. Dhillon, and H. Ren. On the correctness of some bisection-like parallel eigenvalue algorithms in floating point. *Electronic Transactions on Numerical Analysis*, pages 116–149, December 1995.

[31] J. Demmel and W. Gragg. On computing accurate singular values and eigenvalues of acyclic matrices. *Lin. Alg. Appl.*, 185:203–218, 1993.

[32] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica, volume 2.* Cambridge University Press, 1993.

[33] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990.

[34] J. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comp.*, 43(8):983–992, 1994. LAPACK Working Note 59.

[35] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. *SIAM J. Mat. Anal. Appl.*, 13(4):1204–1246, 1992. (also LAPACK Working Note #15).

[36] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[37] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[38] K. Fernando and B. Parlett. Accurate singular values and differential qd algorithms. *Numerische Mathematik*, 67:191–229, 1994.

[39] X. Sun G. Quintana-Orti and C. Bischof. A blas-3 version of the QR factorization with column pivoting. Argonne Preprint MCS-P551-1295, Argonne National Laboratory, 1995.

[40] F. Gantmacher. *The Theory of Matrices, vol. II (transl.).* Chelsea, New York, 1959.

[41] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Num. Anal. (Series B)*, 2(2):205–224, 1965.

[42] G. Golub and C. Reinsch. Singular value decomposition and least squares solutions. *Num. Math.*, 14:403–420, 1970.

[43] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[44] G. H. Golub. Numerical methods for solving linear least squares problems. *Numer. Math.*, 7:206–216, 1965.

[45] G. H. Golub. Some modified matrix eigenvalue problems. *SIAM Review*, 15:318–334, 1973.

[46] M. Gu. Studies in numerical linear algebra. Ph.D. thesis, 1993.

[47] M. Gu, 1996. private communication.

[48] M. Gu, J. Demmel, I. Dhillon, and H. Ren. Efficient computation of singular value decomposition with applications to least squares problem, 1995. Draft.

[49] M. Gu and S. Eisenstat. A stable algorithm for the rank-1 modification of the symmetric eigenproblem. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, September 1992.

[50] M. Gu and S. Eisenstat. An efficient algorithm for computing a rank-revealing QR decomposition. Computer Science Dept. Report YALEU/DCS/RR-967, Yale University, June 1993.

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

[52] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Mat. Anal. Appl.*, 16(1):172–191, January 1995.

[53] P. C. Hansen. Regularization, gsvd and truncated gsvd. *BIT*, pages 491–504, 1989.

[54] M. Heath and C. Romine. Parallel solution of triangular systems on distributed memory multiprocessors. *SIAM J. Sci. Stat. Comput.*, 9:558–588, 1988.

[55] D. Heller. On the efficient computation of recurrence relations, 1974. Institute for Computer Applications in Science and Engineering Rep. (ICASE), NASA Langly Res. Center, Hampton, VA.

[56] D. Heller. A survey of parallel algorithms in numerical linear algebra. *SIAM Review*, 20(4):740–777, 1978.

[57] N. J. Higham. Iterative refinement enhances the stability of QR factorization methods for solving linear systems. *BIT*, 1990.

[58] N. J. Higham. Stability of parallel triangular system solvers. *j-SISC*, 16(2):400–413, MAR 1995.

[59] P. Hong and C. T. Pan. The rank revealing QR and SVD. *Math. Comp.*, 58:575–232, 1992.

[60] A. S. Householder. The theory of matrices in numerical analysis. Blaisdell, New York, 1964.

[61] E. Jessup and I. Ipsen. Improving the accuracy of inverse iteration. *SIAM J. Sci. Stat. Comput.*, 13(2):550–572, 1992.

[62] E. Jessup and D. Sorensen. A parallel algorithm for computing the singular value decomposition of a matrix. Mathematics and Computer Science Division Report ANL/MCS-TM-102, Argonne National Laboratory, Argonne, IL, December 1987.

[63] E. Jessup and D. Sorensen. A divide and conquer algorithm for computing the singular value decomposition of a matrix. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 61–66, Philadelphia, PA, 1989. SIAM.

[64] B. Kågström. The generalized singular value decomposition and the general $A - \lambda B$ problem. *BIT*, 24:568–583, 1984.

[65] W. Kahan. Numerical linear algebra. *Canadian Math. Bull.*, 9:757–801, 1965.

[66] W. Kahan. Accurate eigenvalues of a symmetric tridiagonal matrix. Computer Science Dept. Technical Report CS41, Stanford University, Stanford, CA, July 1966 (revised June 1968).

[67] W. Kahan, 1993. private communication.

[68] W. Kahan, 1996. private communication.

[69] H. T. Kung. New algorithms and lower bounds for the parallel evaluation of certain rational expressions. Technical report, Carnegie Mellon University, February 1974.

[70] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[71] G. Li and T. Coleman. A parallel triangular solver on a distributed memory multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9:485–502, 1988.

[72] K. Li and T.-Y. Li. An algorithm for symmetric tridiagonal eigenproblems — divide and conquer with homotopy continuation. *SIAM J. Sci. Comp.*, 14(3), May 1993.

[73] R.-C. Li, 1994. private communication.

[74] R. Mathias. The instability of parallel prefix matrix multiplication. *SIAM J. Sci. Stat. Comput.*, 16(4):956–973, July 1995.

[75] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right hand sides. *Num. Math.*, 6:405–409, 1964.

[76] S. E. Orcutt. Parallel solution methods for triangular linear systems of equations. Technical Report Tech Report 77, Digital Systems Lab., Stanford Electronics Labs., Stanford, CA, 1974.

[77] C. Paige. Computing the generalized singular value decomposition. *SIAM J. Sci. Stat. Comput.*, 7:1126–1146, 1986.

[78] C. Paige and M. Saunders. Towards a generalized singular value decomposition. *SIAM J. Num. Anal.*, 15:241–256, 1981.

[79] V. Pan and P. Tang. Bounds on singular values revealed by QR factorization. Technical Report MCS-P332-1092, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[80] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[81] G. Peters and J. H. Wilkinson. Inverse iteration, ill-conditioned equations and newton's method. *SIAM Review*, 21:339–360, 1979.

[82] C. Romine and J. Ortega. Parallel solution of triangular systems of equations. *Parallel Computing*, 6:109–114, 1988.

[83] H. Rutishauser. On Jacobi rotation patterns. In N. Metropolis, A. Taub, J. Todd, and C. Tompkins, editors, *Proceedings of Symposia in Applied Mathematics, Vol. XV: Experimental Arithmetic, High Speed Computing and Mathematics.* American Mathematical Society, 1963.

[84] J. Rutter. A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem. Mathematics Dept. Master's Thesis available by anonymous ftp to tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-94-799, file all.ps, University of California, 1994.

[85] A. Sameh and R. Brent. Solving triangular systems on a parallel computer. *SIAM J. Num. Anal.*, 14:1101–1113, 1977.

[86] R. D. Skeel. Scaling for numerical stability in Gaussian elimination. *J. of the ACM*, 26:494–526, 1979.

[87] R. D. Skeel. Iterative refinement implies numerical stability for Gaussian elimination. *Math. Comput.*, 35:817–832, 1980.

[88] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – EISPACK Guide*, volume 6 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1976.

[89] D. Sorensen and P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Num. Anal.*, 28(6):1752–1775, 1991.

[90] J. Speiser and C. Van Loan. Signal processing computations using the generalized singular value decomposition. In *Real Time Signal Processing VII, v. 495*, pages 47–55. SPIE, 1984.

[91] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[92] G. W. Stewart. On the perturbations of pseudo-inverses, projections and linear least squares problems. *SIAM Rev.*, 19:634–662, 1977.

[93] G. W. Stewart. Rank degeneracy. *SIAM J. Sci. Stat. Comput.*, 5:403–413, 1984.

[94] P. Swarztrauber. A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix. *Math. Comp.*, 60(202):651–668, 1993.

[95] C. V. Van Loan. Generalizing the singular value decompositions. *SIAM J. Num. Anal.*, 13:76–83, 1976.

[96] C. V. Van Loan. Computing the CS and the generalized singular value decomposition. *Numer. Math.*, 46:479–491, 1985.

[97] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford, 1965.