On the Correctness of Parallel Bisection in Floating Point

James W. Demmel [*]
Computer Science Division and Department of Mathematics
University of California
Berkeley, California 94720

Inderjit Dhillon[†]
Computer Science Division
University of California
Berkeley, California 94720

Huan Ren[‡]
Department of Mathematics
University of California
Berkeley, California 94720

Computer Science Division Technical Report UCB//CSD-94-805. University of California, Berkeley, CA 94720. March 30, 1994.

Abstract

Bisection is an easily parallelizable method for finding the eigenvalues of real symmetric tridiagonal matrices, or more generally symmetric acyclic matrices. It requires a function Count(x) which counts the number of eigenvalues less than x. In exact arithmetic Count(x) is an increasing function of x, but this is not necessarily the case with roundoff. Our first result is that as long as the floating point arithmetic is monotonic, the computed function Count(x) implemented appropriately will also be monotonic; this extends an unpublished 1966 result of Kahan to the larger class of symmetric acyclic matrices. Second, we analyze the impact of nonmonotonicity of Count(x) on the serial and parallel implementations of bisection. We present simple and natural implementations which can fail because of nonmonotonicity; this includes the routine `bisect` in EISPACK. We also show how to implement bisection correctly despite nonmonotonicity; this is important because the fastest known parallel implementation of Count(x) is nonmonotonic even if the floating point is not.

[*] The author was supported by NSF grants ASC-9005933 and CCR-9196022, and DARPA grant DAAL03-91-C-0047 via a subcontract from the University of Tennessee.

[†] The author was supported by DARPA grant DAAL03-91-C-0047 via a subcontract from the University of Tennessee.

[‡] The author was supported by DARPA grant DAAL03-91-C-0047 via a subcontract from the University of Tennessee.

1 Introduction

Let T by an n-by-n real symmetric tridiagonal matrix with diagonals $a_1, ..., a_n$ and off diagonals $b_1, ..., b_{n-1}$; we let $b_0 \equiv 0$. Let $\lambda_1 \le \cdots \le \lambda_n$ be T's eigenvalues. It is well known [20] that the function Count(x) defined below 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) :

Algorithm 1: 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
endfor
```

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

It is easy to see that the number of eigenvalues in the half-open interval $[\sigma_1, \sigma_2)$ is Count(σ_2) − Count(σ_1). This observation may be used as the basis for a "bisection" algorithm to find all the eigenvalues of T, or just those in an interval $[\sigma_1, \sigma_2)$ or $[\lambda_j, \lambda_k)$. Here we interpret bisection very broadly, referring to any algorithm which involves dividing an interval containing at least one eigenvalue into smaller subintervals of any size, and recomputing the numbers of eigenvalues in the subintervals. The algorithm terminates when the intervals are narrow enough.

The logic of such a bisection algorithm would seem to depend on the simple fact the Count(x) is a monotonic increasing step function of x. 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 4 below, we will see that this can indeed occur using the the EISPACK routine `bisect` (using IEEE floating point standard arithmetic [2, 3], and without over/underflows or other exceptions).

The goal of this paper is to explore the impact of nonmonotonicity on the bisection algorithm. There are at least three reasons why FloatingCount(x) might not be monotonic:

1. the floating point arithmetic is too inaccurate,

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

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

Our first result is to give examples showing monotonicity failures for all three reasons; see sections 4 and 6.

Our second result is to show that as long as the floating point arithmetic is monotonic (we define this in section 2.1), and FloatingCount(x) is implemented in a reasonable (but serial) way, then FloatingCount(x) is 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 [16]; in this paper we extend this result to *symmetric acyclic matrices*, a larger class including tridiagonal matrices, arrow matrices, and exponentially many others [8]; see section 6.

Our third result is to formalize the notion of a correct implementation of bisection, and use this characterization to identify correct and incorrect serial and parallel implementations of bisection. We illustrate with several simple, natural but wrong implementations of parallel bisection, and show how to implement it correctly in the absence of monotonicity; See sections 4 and 7. Nonmonotonic implementations of FloatingCount(x) remain of interest, even though nonmonotonic arithmetics are a dying breed, because the fastest known parallel prefix implementations of FloatingCount(x) appear unavoidably nonmonotonic.

We feel this paper is also of interest because it is an example of a rigorous correctness proof of an algorithm using floating point arithmetic. We make clear exactly which properties of floating point are necessary to prove correctness.

The rest of this paper is organized as follows. Section 2 gives the definitions and assumptions. Section 3 gives tables to illustrate the results of this paper and the assumptions needed to prove the results. Section 4 gives some examples of incorrect bisection algorithms, and also gives some serial and parallel algorithms that are provably correct subject to some assumptions about the computer arithmetic and FloatingCount(x). Section 5 reviews the roundoff error analysis of FloatingCount(x), and how to account for over/underflow; this material may also be found in [16, 8]. Section 6 illustrates how monotonicity can fail, and proves that a natural serial implementation of FloatingCount(x) must be monotonic if the arithmetic is. Section 7 gives formal proofs for the correctness of the bisection algorithms given in section 4. Section 8 discusses some practical implementation issues and Section 9 concludes the paper.

2 Definitions and Assumptions

Section 2.1 defines the kinds of matrices whose eigenvalue problems we will be considering, what monotonic arithmetic is, and what "jump points" of the functions Count() and FloatingCount() are. Section 2.2 presents our (mild) assumptions about floating point arithmetic, the input matrices our algorithms will accept, the way the bisection point of an interval may be chosen. Section 2.3 list the criteria a bisection algorithm must satisfy to be correct.

2.1 Preliminary Definitions

Algorithm 1 was recently extended to the larger class of *symmetric acyclic matrices* [8], 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. 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 has not yet been visited by the algorithm(See Algorithm 6 in section 6 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 $G(T)$).

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. In Section 6, we will prove that the FloatingCount function (Floating TreeCount) for a symmetric acyclic matrix is monotonic if the floating point arithmetic is monotonic.

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

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

Note that if $\lambda_i = \lambda_j$, then λ_i is simultaneously the i^{th} and j^{th} jump point. 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

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

where *nextbefore*(λ_i'') is the largest floating point number smaller than λ_i''. For a nonmonotonic FloatingCount(x) function, there may be more than one such jump-point.

2.2 Assumptions

In order to prove correctness of our algorithms, we need to make some assumptions about the computer arithmetic, the inputs, the bisection algorithm and the function FloatingCount(). 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 3 is about the method used to chose the "bisection" point of an interval; it is also easily enforced. Assumption 4 consists of two statements about the implementation of the function FloatingCount(), which can be proven for most current implementations provided appropriate parts of Assumption 1, about the arithmetic, are true. We still call these two statements an assumption, rather than a theorem, because they are the most convenient building blocks for the ultimate correctness proofs.

Assumption 1

1A. Assumptions about Floating Point Arithmetic Models

Barring overflow, the usual expression for roundoff is extended to include underflow as follows [7]:

$$fl(a \otimes b) = (a \otimes b)(1 + \delta) + \eta \tag{2.1}$$

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:

i. **Model 1.** $fl(a \otimes b) = (a \otimes b)(1+\delta) + \eta$ as above, and overflows terminate.

ii. **Model 2.** IEEE arithmetic with ∞ and NaN arithmetic, and with gradual underflow.

iii. **Model 3.** IEEE arithmetic with ∞ and NaN arithmetic, but with underflow flushing to zero.

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 may not be true of Model 1.

1D. When we talk of parallel implementations, we will assume that all the processors have *identical* floating point arithmetic so that the result of the same floating point operation is bitwise identical on all processors.

Assumption 2

2A. Assumption on the problem size n. $n\varepsilon \le .1$.

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

i. $\bar{B} \ge \omega$.

ii. $\bar{M} \le \sqrt{\Omega}$.

These assumptions may be achieved by explicitly scaling the input matrix (multiplying it by an appropriate scalar), and by ignoring small off-diagonal elements $T_{ij}^2 < \omega$ and so splitting the matrix into *unreduced* blocks [4]; see section 5.8 for details. By Weyl's Theorem [20], 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. These are used to get refined error bounds in Section 5.

[1] These caveats about "safe participation in *any* floating point operation" take machines like the Cray into account, since they have "partial overflow".

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

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

Assumption 3

All the algorithms we will consider try to bound an eigenvalue within an interval. A fundamental operation in these algorithms is to compute a point which lies in an interval (α, β) — we denote this point by *inside*(α, β). This point may be computed by simply bisecting the interval (binary chop) or by applying Newton's or Laguerre's iteration. *We assume that* $fl(inside(\alpha, \beta)) \in (\alpha, \beta)$*, for all attainable values of* α *and* β*.* For example, if we use binary chop, *inside*$(\alpha, \beta) = \frac{\alpha+\beta}{2}$ and we will assume that $fl(\frac{\alpha+\beta}{2}) \in (\alpha, \beta)$, for all $\alpha < \beta$ such that $\beta - \alpha > 2\max(\alpha, \beta)\epsilon$ (i.e., there is at least one floating point number in (α, β)), where ϵ is the machine precision. This assumption always holds in IEEE arithmetic, and for any model of arithmetic which rounds correctly, i.e., rounds a result to the nearest floating point number. For a detailed treatment of how to compute $\frac{\alpha+\beta}{2}$ correctly on various machines, see [17].

An easy way to enforce this assumption given an arbitrary *inside*(α, β) is to replace it by

$$\min(\max(\mathit{inside}(\alpha, \beta), \mathit{nextafter}(\alpha)), \mathit{nextbefore}(\beta))),$$

where *nextafter*(α) is the next floating point number greater than α, and *nextbefore*(β) is the next floating point number less than β.

Assumption 4

4A. FloatingCount(x) does not abort.

4B. Let $\lambda_i^{(1)''}, \lambda_i^{(2)''}, \ldots, \lambda_i^{(k)''}$ be the i^{th} jump-points of FloatingCount(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 $\xi_i \geq 0$. We have assumed that FloatingCount(x) has a bounded region of possible nonmonotonicity, and ξ_i is the width of possible nonmonotonicity around eigenvalue λ_i. Different implementations of Count(x) result in different values of ξ_i (see Section 5).

For some of the practical FloatingCount functions in use, we will prove Assumption 4 in Section 5.

2.3 When is a Bisection Algorithm Correct?

We now describe the functional behaviour required of any correct implementation of bisection. Let τ_i be a user-supplied upper bound on the desired error in the i^{th} computed eigenvalue; this means the user wants the computed eigenvalue λ_i' to differ from the true eigenvalue λ_i by no more than τ_i. Note that not all values of τ_i are attainable, and the attainable values of τ_i depend on the FloatingCount function, the underlying computer arithmetic and the input matrix T. For example, when using Algorithm 1, τ_i can range from $\max_i |\lambda_i|$ down to $O(\varepsilon)\max_i |\lambda_i|$, or perhaps smaller for special matrices [5]. Let U be a non-empty set of indices that correspond to the eigenvalues that the user wants to compute, e.g., $U = \{1, \ldots, n\}$ if the user wants to compute all the eigenvalues of a matrix of dimension n. The output of the algorithm should be a sorted list of the computed eigenvalues, i.e. a list (i, λ'_i) where each $i \in U$ occurs exactly once, and $\lambda'_i \leq \lambda'_j$, when $i \leq j$. In a parallel implementation, this list may be distributed over the processors in any way at the end of the computation. Thus, the algorithm must compute the eigenvalues and also present the output in sorted order. Beyond neatness, the reason that we require the eigenvalues to be returned in sorted order is that it facilitates subsequent uses that require scanning through the eigenvalues in order, such as reorthogonalization of eigenvectors in inverse iteration.

In summary, when we say that an implementation of the bisection algorithm is *correct*, we assert that it terminates and all of the following hold:

- Every desired eigenvalue is computed exactly once.
- The computed eigenvalues are correct to within the user specified error tolerance, i.e. for all desired $i > 0$, $|\lambda_i - \lambda'_i| \leq \tau_i + \xi_i$ (in case $\tau_i > \xi_i$, the implementation can easily guarantee that $|\lambda_i - \lambda'_i| \leq \tau_i$). See section 2.2, Assumption 4B for a definition of ξ_i.
- The computed eigenvalues are in sorted order.

We say that an implementation of the bisection algorithm is *incorrect* when any of the above fails to hold.

3 Outline of the Paper

In this section, we outline our results in four tables. Table 1 lists all the implementations of FloatingCount() we consider, and says where they are discussed in the paper. Table 2 lists all the implementations of bisection we consider, and says where they are discussed in the paper. These bisection algorithms all use an implementation of FloatingCount() internally. Table 3 summarizes the error analyses of the FloatingCount() implementations in Table 1. It reports which parts of Assumption 1, about arithmetic, and Assumption 2, about the input matrix, are necessary to prove whether FloatingCount() satisfies Assumption 4, and whether or not FloatingCount() is monotonic. Detailed numerical error bounds for each implementation of FloatingCount() are reported in section 5, especially Tables 5 through 9. Table 4 says which assumptions are needed to guarantee the correctness of the overall bisection algorithms in Table 2. Basically, all algorithms require Assumption 3, about choosing the bisection point, Assumption 4, which depends on FloatingCount as summarized in Table 3, and all parallel bisection algorithms require Assumption 1D about parallel processors having identical arithmetic.

So, for example, Table 4 says that Algorithms SER_BISEC, SER_ALLEIG, PAR_ALLEIG2 and PAR_ALLEIG3 will be correct when used with any of the implementations of Floating-

Algorithms	Description	Where
`bisect`	algorithm used in the Eispack routine; most floating point exceptions avoided by tests and branches	See section 5.2 and [21]
IEEE	IEEE standard floating point arithmetic used to accommodate possible exceptions; tridiagonals only	See section 5.3 and [4, 16]
`sstebz`	algorithm used in the Lapack routine floating point exceptions avoided by tests and branches	See section 5.4 and [1]
Best Scaling Routine	like `sstebz`, but prescales for optimal error bounds	See section 5.5 and [4, 16]

Table 1: Different implementations of FloatingCount()

Algorithms	Description	Where
Ser_Bisec	Serial bisection algorithm that finds all the eigenvalues of T in a user-specified interval	See section 4.4
Ser_AllEig	Serial bisection algorithm that finds all the eigenvalues of T	See section 4.4
Par_AllEig1	Parallel bisection algorithm that finds all the eigenvalues of T, load balancing by equally dividing the Gerschgorin interval into p equal subintervals; needs monotonic arithmetic	See section 4.5
Par_AllEig2	Similar to Par_AllEig1, but monotonic arithmetic unneeded	See section 4.5
Par_AllEig3	Parallel bisection algorithm that finds all the eigenvalues of T, load balancing by making each processor find an equal number of eigenvalues; monotonic arithmetic unneeded	See section 4.5

Table 2: Different implementations of Bisection

Assumptions about Arithmetic and Input Matrix	Results	Proofs
T is symmetric tridiagonal $\wedge$ (1A(ii) $\vee$ 1A(iii)) $\wedge$ 1B $\wedge$ 2A $\wedge$ 2B(ii)	For `bisect`'s FloatingCount(x), Assumption 4 holds but FloatingCount(x) can be nonmonotonic	See section 4.2 and section 5.2
T is symmetric tridiagonal $\wedge$ (1A(ii) $\vee$ 1A(iii)) $\wedge$ 1B $\wedge$ 2A $\wedge$ 2B	For IEEE routine's FloatingCount(x), Assumption 4 holds and FloatingCount(x) is monotonic	See section 5.3 and section 6
T is symmetric acyclic $\wedge$ 1A $\wedge$ 1B $\wedge$ 1C $\wedge$ 2A $\wedge$ 2B(ii)	For `sstebz`'s FloatingCount(x), Assumption 4 holds and FloatingCount(x) is monotonic	See section 5.4 and section 6
T is symmetric acyclic $\wedge$ 1A $\wedge$ 1B $\wedge$ 1C $\wedge$ 2A	For Best Scaling's FloatingCount(x), Assumption 4 holds and FloatingCount(x) is monotonic	See section 5.5 and section 6

Table 3: Results of Roundoff Error Analysis and Monotonicity

Assumptions	Results	Proofs
3, 4	Algorithm SER_BISEC is correct	See section 7
3, 4	Algorithm SER_ALLEIG is correct	See section 4
1D, 3, 4, and FloatingCount(x) is monotonic	Algorithm PAR_ALLEIG1 is correct	See section 7.2
1D, 3, 4	Algorithm PAR_ALLEIG2 is correct	See section 7.2
1D, 3, 4	Algorithm PAR_ALLEIG3 is correct	See section 7.2

Table 4: Correctness Results

Count in Table 1, provided the corresponding conditions in Table 3 are met. On the other hand, Algorithm PAR_ALLEIG1 cannot be used correctly with `bisect`'s FloatingCount, since that FloatingCount is not monotonic.

4 Correctness of Bisection Algorithms

We now investigate the correctness of bisection algorithms and exhibit some existing "natural" implementations of bisection that are incorrect. Table 4 summarizes all the correctness results that we prove in this paper.

4.1 An Incorrect Serial Implementation of Bisection

We give an example of the failure of EISPACK's `bisect` routine in the face of a nonmonotonic FloatingCount(x). Suppose we use IEEE standard double precision floating point arithmetic with $\varepsilon = 2^{-52} \approx 2.2 \cdot 10^{-16}$ and we want to find the eigenvalues of the following 2×2 matrix:

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

In exact arithmetic, A has eigenvalues near 1 and $-\varepsilon^2 \approx -4.93 \cdot 10^{-32}$. But `bisect` reports that the interval $[-10^{-32}, 0)$ contains -1 eigenvalues. The reason for this is `bisect`'s incorrect provision against division by zero (See Algorithm 3 in Section 5). In Section 6, by the proof of Theorem 6.1, we will show that this cannot happen for the LAPACK routine `dstebz(sstebz)` even for general symmetric acyclic matrices.

4.2 Nonmonotonicity of Parallel Prefix Algorithm

We now give another example of a nonmonotonic FloatingCount(x) when Count(x) is implemented using a fast parallel algorithm called parallel prefix [9]. Figure 1 shows the FloatingCount(x) of a 64×64 matrix of norm near 1 with 32 eigenvalues very close to $5 \cdot 10^{-8}$ computed both by the conventional bisection algorithm and the parallel prefix algorithm in the neighborhood of the eigenvalues; see [19, 12] for details.

4.3 A Correct Serial Implementation of the Bisection Algorithm

As we saw in Section 4.1, the EISPACK implementation of the bisection algorithm fails in the face of nonmonotonicity of the function FloatingCount(x). We now present an imple-

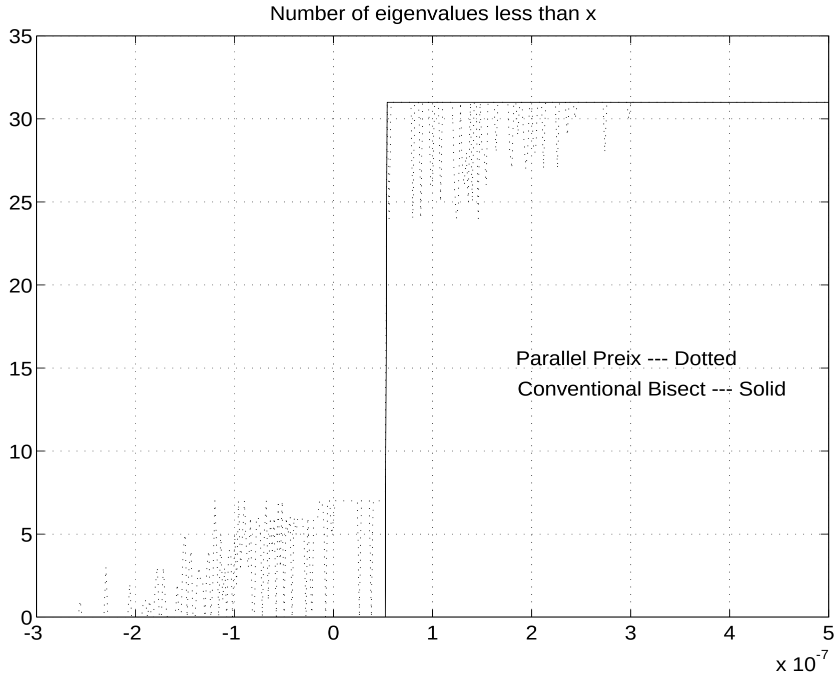

Figure 1: Parallel Prefix vs Conventional Bisection

mentation which works correctly irrespective of whether FloatingCount(x) is monotonic or not.

All the intervals referred to in the following discussion will be half-open intervals of the form $[\alpha, \beta)$. We define a *task* to be a 5-tuple $T = (\alpha, \beta, n_\alpha, n_\beta, O)$, where $[\alpha, \beta)$ is a non-empty interval, n_α and n_β are the counts associated with α and β respectively, and O is the set of indices corresponding to the eigenvalues being searched for in this interval. We obviously require $O \subseteq I_{n_\alpha}^{n_\beta}$, where $I_{n_\alpha}^{n_\beta} = \{n_\alpha + 1, \ldots, n_\beta\}$ ($I_{n_\alpha}^{n_\beta} = \emptyset$ when $n_\alpha \geq n_\beta$). We do not insist that $n_\alpha =$ FloatingCount(α), and $n_\beta =$ FloatingCount(β), only that $n_\alpha \geq$ FloatingCount(α) and $n_\beta \leq$ FloatingCount(β). In most implementations $O = I_{n_\alpha}^{n_\beta}$ and the index set O is not explicitly maintained by the implementation.

Algorithm SER_BISEC (see Figure 2) is a correct serial implementation of bisection, that finds all the eigenvalues specified by the given initial task $(left, right, n_{left}, n_{right}, I_{n_{left}}^{n_{right}})$, the i^{th} eigenvalue being found to the desired accuracy τ_i (note that we allow different tolerances to be specified for different eigenvalues, τ_i being the i^{th} component of the input tolerance vector τ). The difference between this implementation and the EISPACK implementation is the initial check to see if $n_{left} \geq n_{right}$, and the *forcing* of monotonicity on FloatingCount(x) by executing statement 10 at each iteration. Statement 10 has no effect if FloatingCount(x) is monotonic, whereas it forces n_{mid} to lie between n_α and n_β if FloatingCount(x) is nonmonotonic.

Theorem 4.1 *Algorithm* SER_BISEC *is correct if Assumptions 3 and 4 hold.*

PROOF. The theorem is proved in Section 7. □

```
subroutine Ser_Bisec($n$,$T$,$left$,$right$,$n_{left}$,$n_{right}$,$\tau$) /* computes the eigenvalues of $T$
            in the interval $[left, right]$ to the desired accuracy $\tau$ */
    if ($n_{left} \geq n_{right}$ or $left > right$) return;
    if (FloatingCount($left$) $> n_{left}$ or FloatingCount($right$) $< n_{right}$) return;
    enqueue $(left, right, n_{left}, n_{right}, I_{n_{left}}^{n_{right}})$ to $Worklist$;
    while ($Worklist$ is not empty)
        dequeue $(\alpha, \beta, n_\alpha, n_\beta, I_{n_\alpha}^{n_\beta})$ from $Worklist$;
        $mid = inside(\alpha, \beta)$;
        if ($\beta - \alpha < \min_{i=n_\alpha+1}^{n_\beta} \tau_i$) then
            print "Eigenvalue $\min(\max((\alpha+\beta)/2, \alpha), \beta)$ has multiplicity $n_\beta - n_\alpha$";
        else
            $n_{mid} = \min(\max(\text{FloatingCount}(mid), n_\alpha), n_\beta)$;
            if ($n_{mid} > n_\alpha$) then
                enqueue $(\alpha, mid, n_\alpha, n_{mid}, I_{n_\alpha}^{n_{mid}})$ to $Worklist$;
            end if
            if ($n_{mid} < n_\beta$) then
                enqueue $(mid, \beta, n_{mid}, n_\beta, I_{n_{mid}}^{n_\beta})$ to $Worklist$;
            end if
        end if
    end while
end subroutine
```

Figure 2: **Algorithm** Ser_Bisec

```
    subroutine SER_ALLEIG($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ */
        $(gl, gu)$ = COMPUTE_GERSCHGORIN($n$,$T$);
        call SER_BISEC($n$,$T$,$gl$,$gu$,0,$n$,$\tau$);
    end subroutine
    function COMPUTE_GERSCHGORIN($n$,$T$) /* returns the Gerschgorin Interval $(gl, gu)$ */
1:      $gl = \min_{i=1}^{n}(T_{ii} - \sum_{j \neq i} |T_{ij}|)$;        /* Gerschgorin left bound */
2:      $gu = \max_{i=1}^{n}(T_{ii} + \sum_{j \neq i} |T_{ij}|)$;        /* Gerschgorin right bound */
3:      $bnorm = \max(|gl|, |gu|)$;
4:      $gl = gl - bnorm * 2n\varepsilon - \xi_0$; $gu = gu + bnorm * 2n\varepsilon + \xi_n$; /* see Table 9 */
5:      return($gl$,$gu$);
    end function
```

Figure 3: **Algorithm** SER_ALLEIG computes all the eigenvalues of T

Note that in all our theorems about the correctness of various bisection algorithms, unless stated otherwise, we do not require FloatingCount(x) to be monotonic.

Algorithm SER_ALLEIG (see figure 3) is designed to compute all the eigenvalues of T to the desired accuracy. COMPUTE_GERSCHGORIN is a subroutine that computes the Gerschgorin interval of the symmetric acyclic matrix T. Note that in line 4 of the pseudocode, the Gerschgorin interval is widened to ensure that FloatingCount(gl) $= 0$ and FloatingCount(gu) $= n$ and hence is guaranteed to contain all the eigenvalues (this is proved in Section 4). Due to the correctness of Algorithm Ser_Bisec, we have the following corollary :

Corollary 4.1 *Algorithm* SER_ALLEIG *is correct if Assumptions 3 and 4 hold.*

4.4 Parallel Implementation of the Bisection Algorithm

We now discuss parallel implementations of the bisection algorithm. The bisection algorithm offers ample opportunities for parallelism, and many parallel implementations exist [6, 13, 18, 14]. We first discuss a natural parallel implementation which gives the false appearance of being *correct*. Then, we give a *correct* parallel implementation that has been tested extensively on the Connection Machine CM-5 supercomputer.

4.4.1 A Simple, Natural and Incorrect Parallel Implementation

A natural way to divide the work arising in the bisection algorithm among p processors is to partition the initial Gerschgorin interval into p equal subintervals, and assign to processor i the task of finding all the eigenvalues in the i^{th} subinterval. Algorithm PAR_ALLEIG0 (see figure 4) is a simple and natural parallel implementation based on this idea that attempts to find all the eigenvalues of T (the p processors are assumed to be numbered $0, 1, 2, \ldots, p-1$). However, algorithm PAR_ALLEIG0 fails to find the eigenvalue of a 1×1 identity matrix when

```
subroutine Par_Alleig0($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = My_Processor_Number();
    ($gl$,$gu$) = Compute_Gerschgorin($n$,$T$);
    $average\_width = (gr - gl)/p$;
    $\alpha(i) = gl + i * average\_width$;
    $\beta(i) = \alpha(i) + average\_width$;
    $n_{\alpha(i)}$ = FloatingCount($\alpha(i)$);
    $n_{\beta(i)}$ = FloatingCount($\beta(i)$);
    call Ser_Bisec($n$,$T$,$\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$\tau$);
end subroutine
```

Figure 4: **Algorithm** Par_Alleig0 executed by processor i — An *Incorrect* Parallel Implementation

implemented on a 32 processor CM-5! The reason is that when $p = 32$, *average_width* is so small that $\alpha(i) = \beta(i)$ for $i = 0, \ldots, p-1$. Thus, none of the processors are able to find the only eigenvalue. (This would happen on any machine with IEEE arithmetic, not just the CM-5.)

The error in algorithm Par_Alleig0 is in the way $\beta(i)$ is computed. Algorithm Par_Alleig1 (see figure 5) fixes the problem by computing $\beta(i)$ as $gl + (i+1) * average\_width$. This results in the following theorem, which we prove in Section 7.

Theorem 4.2 *Algorithm* Par_Alleig1 *is correct if Assumptions 1D, 3 and 4 hold and* FloatingCount(x) *is monotonic.*

However, when FloatingCount(x) is nonmonotonic, Algorithm Par_Alleig1 is still *incorrect*! The error in the algorithm is that when FloatingCount(x) is nonmonotonic, a desired eigenvalue may be computed more than once. For example, suppose $n = p = 3$, $n_{\alpha(0)} = 0$, $n_{\beta(0)} = n_{\alpha(1)} = 2$, $n_{\beta(1)} = n_{\alpha(2)} = 1$, and $n_{\beta(2)} = 3$. In this case, the second eigenvalue will be computed both by processors 0 and 2.

Algorithm Par_Alleig2 (see figure 6) corrects this problem by making processor 0 find all the initial tasks that are input to algorithm Ser_Alleig. These initial tasks are formed such that $n_{\alpha(i)} \leq n_{\beta(i)}$, for $i = 0, \ldots, p-1$, and are then communicated to the other processors. Note that these initial tasks may also be formed by computing $n_{\alpha(i)}$ and $n_{\beta(i)}$ in parallel on each processor, making sure $n_{\alpha(i)} \leq n_{\beta(i)}$, and then doing two max scans replacing $n_{\alpha(i)}$ by $\max_{j \leq i} n_{\alpha(j)}$ and $n_{\beta(i)}$ by $\max_{j \leq i} n_{\beta(j)}$. This would take $\log(p)$ steps on p processors. The function send(i,n_γ) sends the number n_γ to processor i, while receive(0,$n_{\alpha(i)}$) results in $n_{\alpha(i)}$ being set to the number sent by processor 0. Thus, we have the following theorem:

Theorem 4.3 *Algorithm* Par_Alleig2 *is correct if Assumptions 1D, 3 and 4 hold.*

Proof. The theorem is proved in Section 7. □

```
subroutine Par_Alleig1($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = My_Processor_Number();
    $(gl,gu)$ = Compute_Gerschgorin($n$,$T$);
    $average\_width = (gr - gl)/p$;
    $\alpha(i) = gl + i * average\_width$;
    $\beta(i) = \max(gl + (i+1) * average\_width, \alpha(i))$;
    if $(i = p - 1)$ $\beta(i) = gu$;
    $n_{\alpha(i)}$ = FloatingCount($\alpha(i)$);
    $n_{\beta(i)}$ = FloatingCount($\beta(i)$);
    call Ser_Bisec($n$,$T$,$\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$\tau$);
end subroutine
```

Figure 5: **Algorithm** Par_Alleig1 executed by processor i — *Correct* when FloatingCount(x) is monotonic

4.4.2 A Practical Correct Parallel Implementation

Although correct, Algorithm Par_Alleig2 is very sensitive to the eigenvalue distribution in the Gerschgorin interval, and does not result in high speedups on massively parallel machines when the eigenvalues are not distributed uniformly. We now give a more practical parallel implementation, and prove its correctness. Algorithm Par_Alleig3 (see figure 7) partitions the work among the p processors by making each processor find an almost equal number of eigenvalues (for ease of presentation, we assume that p divides n). This static partitioning of work has been observed to give good performance on parallel machines like the CM-5 [13], for almost all eigenvalue distributions.

In algorithm Par_Alleig3, processor i attempts to find eigenvalues $i(n/p)+1$ through $(i+1)n/p$. The function Find_Init_Task invoked on processor i finds a floating point number $\beta(i)$ such that FloatingCount($\beta(i)$) $= (i+1)n/p$, unless $\lambda_{(i+1)n/p}$ is part of a cluster of eigenvalues. In the latter case, Find_Init_Task finds a floating point number $\beta(i)$ such that FloatingCount($\beta(i)$) is bigger than $(i+1)n/p$ and $\lambda_{(i+1)n/p}, \ldots, \lambda_{\mathrm{FloatingCount}(\beta(i))}$ form a cluster relative to the user specified error tolerance, τ. If the cluster is so big that FloatingCount($\beta(i)$) is larger than $(i+2)n/p$, processor i sets $n_{\beta(i)}$ to $(i+1)n/p$ in order to ensure that each desired eigenvalue is computed just once. Each processor i computes $\beta(i)$, and communicates with processor $i-1$ to receive $\alpha(i)$, and then calls algorithm Ser_Bisec with the initial task ($\alpha(i)$, $\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$I_{\alpha(i)}^{\beta(i)}$) returned by the function Find_Init_Task. When the eigenvalues are well separated, each processor finds an equal number of eigenvalues.

Theorem 4.4 *Algorithm* Par_Alleig3 *is correct if Assumptions 1D, 3 and 4 hold.*

Proof. The theorem is proved in Section 7. □

Note that we used some communication between processors to guarantee correctness of

```
subroutine PAR_ALLEIG2($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = MY_PROCESSOR_NUMBER();
    ($gl$,$gu$) = COMPUTE_GERSCHGORIN($n$,$T$);
    $average\_width = (gr - gl)/p$;
    $\alpha(i) = gl + i * average\_width$;
    $\beta(i) = gl + (i+1) * average\_width$;
    if $(i = p-1)$ $\beta(i) = gu$;
    if $(i = 0)$ then
        $n_\gamma = 0$;
        for $(i = 1; i < p; i = i+1)$ do        /* does a max scan */
            $\gamma = gl + i * average\_width$;
            $n_\gamma$ = max(FloatingCount($\gamma$), $n_\gamma$);
            send($i$,$n_\gamma$);
        end for
    else
        receive(0,$n_{\alpha(i)}$);
    end if
    if $(i \neq 0)$ then
        send($i-1$,$n_{\alpha(i)}$);
    end if
    if $(i \neq p-1)$ then
        receive($i+1$,$n_{\beta(i)}$);
    end if
    call SER_BISEC($n$,$T$,$\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$\tau$);
end subroutine
```

Figure 6: **Algorithm** PAR_ALLEIG2 executed by processor i — A *Correct* Parallel Implementation

```
subroutine PAR_ALLEIG3($n$,$T$,$\tau$) /* computes all the eigenvalues of $T$ in parallel */
    $i$ = MY_PROCESSOR_NUMBER();
    ($gl$,$gu$) = COMPUTE_GERSCHGORIN($n$,$T$);
    ($\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$) = FIND_INIT_TASK($n$,$T$,$gl$,$gu$,0,$n$,$\tau$);
    call SER_BISEC($n$,$T$,$\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$\tau$);
end subroutine

function FIND_INIT_TASK($n$,$T$,$\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$\tau$) /* returns initial task */
    $save = \alpha(i)$; $n_{save} = n_{\alpha(i)}$;
    while ( $(n_{\beta(i)} \neq (i+1)n/p)$ and $(\beta(i) - \alpha(i) > \min_{i=n_{\alpha(i)}+1}^{n_{\beta(i)}} \tau_i)$)
        $mid = inside(\alpha(i), \beta(i))$; $n_{mid} = \min(\max(\text{FloatingCount}(mid), n_{\alpha(i)}), n_{\beta(i)})$;
        if $(n_{mid} \geq (i+1)n/p)$ then
            $\beta(i) = mid$; $n_{\beta(i)} = n_{mid}$;
        else
            $\alpha(i) = mid$; $n_{\alpha(i)} = n_{mid}$;
        end if
    end while
    $\gamma(i) = \beta(i)$;
    if $(n_{\beta(i)} > (i+2)n/p)$ then
        $n_{\beta(i)} = (i+1)n/p$;
        $\gamma(i) = \alpha(i)$;
    end if
    if $(i \neq p-1)$ then
        send($i+1$,$\gamma(i)$);
        send($i+1$,$n_{\beta(i)}$);
    end if
    if $(i \neq 0)$ then
        receive($i-1$,$\alpha(i)$);
        receive($i-1$,$n_{\alpha(i)}$);
    else
        $\alpha(i) = save$; $n_{\alpha(i)} = n_{save}$;
    end if
    return($\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$);
end function
```

Figure 7: **Algorithm** PAR_ALLEIG3 executed by processor i when each processor finds an (almost) equal number of eigenvalues

the parallel algorithms. It is much harder (and less elegant) to construct and prove correctness of similar *correct* and efficient parallel algorithms that do not use any communication.

5 Roundoff Error Analysis

As we mentioned before, Algorithm 1 was recently extended to the symmetric acyclic matrices. In [8] the following implementation of Count(x) for acyclic matrices was given. The algorithm refers to the tree $G(T)$, 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.

Algorithm 2: Count(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

```
call TreeCount($1, x, d, s$)
Count = $s$

procedure TreeCount($i, x, d, s$)        /* $i$ and $x$ are inputs, $d$ and $s$ are outputs */
    $s = 0$
    $sum = 0$
    for all children $j$ of $i$ do
        call TreeCount($j, x, d', s'$)
        $sum = sum + T_{ij}^2/d'$
        $s = s + s'$
    endfor
    $d = (T_{ii} - x) - sum$
    if $d < 0$ then $s = s + 1$
end TreeCount
```

In [8] it is also shown that barring over/underflow, the floating point version of Algorithm 2 has the same attractive backward error analysis as the floating point version of Algorithm 1: 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:

$$|T_{ij} - T'_{ij}| \leq f(C/2 + 2, \varepsilon)|T_{ij}| \text{ if } i \neq j \text{ and } T_{ii} = T'_{ii}, \tag{5.2}$$

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 ($n\varepsilon \leq .1$), we have [22]:

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

(Strictly speaking, the proof of this bound is a slight modification of the one in [8], and requires that d be computed exactly as shown in TreeCount. The analysis in [8] makes no assumption about the order in which the sum for d is evaluated, whereas the bound (5.2) 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 quite accurately, as we now discuss. Suppose the backward error in (5.2) can change eigenvalue λ_k by at most ξ_k. For example, Weyl's Theorem [20] implies that $\xi_k \leq \|T - T'\|_2 \leq 2f(C/2+2,\varepsilon)\|T\|_2$, i.e. that each eigenvalue is changed by an amount small compared to the largest eigenvalue. If $T_{ii} = 0$ for all i, then $\xi_k \leq (1-(C+4)\varepsilon)^{1-n}|\lambda_k|$, i.e. each eigenvalue is changed by an amount small relative to itself. See [16, 5, 10] for more such bounds.

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

$$i = \text{FloatingCount}(x, T) < \text{FloatingCount}(y, T) = j \quad . \tag{5.3}$$

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

$$x - \xi_{i+1} \leq \lambda_{i+1}(T) \leq \lambda_j(T) < y + \xi_j \quad . \tag{5.4}$$

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

$$x - \xi_j \leq \lambda_j(T) < y + \xi_j \quad . \tag{5.5}$$

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 (5.3). Thus, in principle $\lambda_j(T)$ can be computed nearly as accurately as the inherent uncertainty ξ_j permits.

Accounting for over/underflow is done as follows. We first discuss the way it is done in EISPACK's `bisect` routine [21], then the superior method in LAPACK's `sstebz` routine [1, 16]. The difficulty arises because if d' is tiny or zero, the division T_{ij}^2/d' can overflow. In addition, T_{ij}^2 can itself over/underflow. We would like to account for this by modifying the algorithm to avoid over/underflow (not necessary if we have IEEE arithmetic), and slightly increasing the backward error bound (5.2).

We denote the *pivot* d computed when visiting node i by d_i. The floating point operations performed while visiting node i are then

$$d_i = fl((T_{ii} - x) - (\sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{d_j})). \tag{5.6}$$

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

5.1 Model 1: Barring Overflow, Acyclic Matrix

Barring overflow, (5.6) and Assumption 2B(i) 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

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

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

$$\tilde{d}_i = T_{ii} + (2C+1)\eta_i - x - \sum_{\substack{\text{all children} \\ j \text{ of } i}} \frac{T_{ij}^2}{\tilde{d}_j}(1 + \varepsilon_{ij})^{C+4}. \qquad (5.7)$$

Remark 5.1 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 (5.7) 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 (5.2), since the $\tilde{d}_i$ are the exact pivots correspond to T' where T' satisfies (5.2) and $\text{sign}(\tilde{d}_i) = \text{sign}(d_i)$.

Remark 5.2 We need to bar overflow *in principle* for symmetric acyclic matrix with IEEE arithmetic, because if in (5.6), 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.

5.2 Models 2 and 3: EISPACK's bisect routine, Tridiagonal Matrix

EISPACK's `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 5.2). In this subsection, 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 `bisect`, we need the following assumptions:

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

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

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

Algorithm 3: EISPACK **bisect.** Count(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix** T **that are less than** x**.**

```
Count = 0;
$d_0 = 1$;
for $i = 1$ to $n$
    if ($d_i = 0$) then
        $v = |b_{i-1}|/\varepsilon$
    else
        $v = b_{i-1}^2/d_i$
    endif
    $d_i = a_i - x - v$
    if $d_i < 0$ then Count = Count + 1
endfor
```

Under Models 2 and 3, our error expression (5.7) 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, `bisect`'s provision against division by zero can drastically increase the backward error bound (5.2). When $d_j = 0$ for some j in (5.6), 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 error, because $\varepsilon|b_j|$ maybe 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 corresponds 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 `bisect`:

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

and

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

5.3 Models 2 and 3: IEEE routine, Tridiagonal Matrix

The following code can work only for symmetric tridiagonal matrices under Models 2 and 3 for the same reason as `bisect`: otherwise we could get $T^2_{ij_1}/d_{j_1} + T^2_{ij_2}/d_{j_2} = \infty - \infty = \text{NaN}$. So in this subsection, 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 [11]. To describe the error analysis, we again make Assumptions 1A(ii), 1A(iii) and 2B(ii) as in Section 5.2, and Assumption 2B(i), which is $\bar{B} \equiv \min_{i \ne j} T^2_{ij} \ge \omega$.

Algorithm 4: IEEE routine. Count(x) **returns the number of eigenvalues of a real symmetric tridiagonal matrix** T **that are less than** x**.**

```
Count = 0;
$d_0 = 1$;
for $i = 1$ to $n$
    /* note that there is no provision against overflow and division by zero */
    $d_i = (a_i - x) - b^2_{i-1}/d_{i-1}$
    Count = Count + 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 4 is quite similar to `bisect` except there is no provision against division by zero, and the SignBit(± 0) function (= 0 or 1) is used to count eigenvalues. 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 ∞, therefore Count also increases by 1 after two steps, which is correct. Using an analysis analogous to the last section, if we use Model 2 (gradual underflow), T' differs from T by

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

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

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

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

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

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

5.4 Models 1, 2 and 3: LAPACK's sstebz routine, Acyclic Matrix

In contrast to EISPACK's `bisect` and IEEE routines, LAPACK's `sstebz` 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 subsection, 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 `sstebz`). In this subsection, we need the Assumptions 1A (model i, ii or iii) 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 5: Count(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

```
call TreeCount($1, x, d_1, s_1$)
Count = $s_1$

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 ($|d_i| \leq \hat{p}$)   $d_i = -\hat{p}$
    if $d_i < 0$ then $s_i = s_i + 1$
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 `sstebz` 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

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

5.5 Models 1,2 and 3: Best Prescaling Algorithm, Acyclic Matrix

Following Kahan[16], 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 subsection, T is a symmetric acyclic matrix. To describe the error analysis, we only need Assumption 1A (i, ii or iii). Again because of the Gerschgorin Disk Theorem, the shifts are restricted to those x such that $|x| \leq (n+1)M$. The Best Scaling Algorithm is almost the same as the `sstebz` except $\hat{p} = -\sqrt{\omega}$, so we do not repeat the code here.

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

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

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

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

5.6 Error Bounds For Eigenvalues

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

Lemma 5.1 *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

$$\xi_k \leq 2\alpha(\varepsilon) \parallel T \parallel_2 + \beta. \tag{5.8}$$

Proof. By Weyl's Theorem [20],

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

and

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

Algorithms	Model 1		Model 2		Model 3	
	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β
bisect	—	—	$f(2.5,\varepsilon)$	$\varepsilon\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$\varepsilon\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$
sstebz	$f(2.5,\varepsilon)$	$2\hat{p}+3\bar{\omega}$	$f(2.5,\varepsilon)$	$2\hat{p}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$2\hat{p}+3\omega$
best scaling	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+3\bar{\omega}$	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$2\sqrt{\omega}+3\omega$
IEEE	—	—	$f(2.5,\varepsilon)$	$\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$f(2.5,\varepsilon)$	$\frac{\bar{M}^2}{\Omega}+3\omega$

Table 5: Backward Error Bounds for Symmetric Tridiagonal Matrices

Algorithms	Model 1	Model 2	Model 3
bisect	—	$[2f(2.5,\varepsilon)+\varepsilon]\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+\varepsilon\omega$	$[2f(2.5,\varepsilon)+\varepsilon]\|\|T\|\|_2+\frac{\bar{M}^2}{\Omega}+3\omega$
sstebz	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\hat{p}+3\bar{\omega}$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\hat{p}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\hat{p}+3\omega$
best scaling	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+3\bar{\omega}$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+\varepsilon\omega$	$2f(2.5,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+3\omega$
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 6: Error Bounds ξ_k of Eigenvalues for Symmetric Tridiagonal Matrices

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

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

□

According to this lemma, we present the tables of backward errors $\alpha(\varepsilon)$ and β, and the corresponding error bounds ξ_k of the eigenvalues, for different algorithms under different models (Tables 5 through 8)

5.7 Correctness of the Gerschgorin Bound

In this subsection, we will prove the correctness of the Gerschgorin bound computed by the routine COMPUTE_GERSCHGORIN (see Figure 3).

Since

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

So, $bnorm = \max(|gl|, |gu|) = \|T\|_\infty$. 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,

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

Algorithms	Model 1		Model 2		Model 3	
	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β	$\alpha(\varepsilon)$	β
bisect	—	—	—	—	—	—
sstebz	$f(C/2+2,\varepsilon)$	$2\hat{p}+(2C+1)\bar{\omega}$	$f(C/2+2,\varepsilon)$	$2\hat{p}+C\varepsilon\omega$	$f(C/2+2,\varepsilon)$	$2\hat{p}+(2C+1)\omega$
best scaling	$f(C/2+2,\varepsilon)$	$2\sqrt{\omega}+(2C+1)\bar{\omega}$	$f(C/2+2,\varepsilon)$	$2\sqrt{\omega}+C\varepsilon\omega$	$f(C/2+2,\varepsilon)$	$2\sqrt{\omega}+(2C+1)\omega$
IEEE	—	—	—	—	—	—

Table 7: Backward Error Bounds for Symmetric Acyclic Matrices

Algorithms	Model 1	Model 2	Model 3
bisect	—	—	
sstebz	$2f(C/2+2,\varepsilon)\|\|T\|\|_2+2\hat{p}+(2C+1)\bar{\omega}$	$2f(C/2+2,\varepsilon)\|\|T\|\|_2+2\hat{p}+C\varepsilon\omega$	$2f(C/2+2,\varepsilon)\|\|T\|\|_2+2\hat{p}+(2C+1)\omega$
best scaling	$2f(C/2+2,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+(2C+1)\bar{\omega}$	$2f(C/2+2,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+C\varepsilon\omega$	$2f(C/2+2,\varepsilon)\|\|T\|\|_2+2\sqrt{w}+(2C+1)\omega$
IEEE	—	—	

Table 8: Error Bounds ξ_k of Eigenvalues for Symmetric Acyclic Matrices

Algorithms	Matrix	Additional Assumptions	Bounds of ξ_k
bisect	Tridiagonal	Assumption 2C(i)	$11\varepsilon * bnorm$
sstebz	Acyclic	Assumptions 2C(i), 2C(ii)	$(8n+6)\varepsilon * bnorm$
best scaling	Acyclic	—	$(4n+8)\varepsilon * bnorm$
IEEE	Tridiagonal	Assumption 2C(i)	$10\varepsilon * bnorm$

Table 9: Upper Bounds for ξ_k for Different Algorithms under Different Models

Similarly, $|fl(gu) - gu| \le 2n\varepsilon * bnorm$. This proves that if we let

$$gl = gl - 2n\varepsilon * bnorm - \xi_0; \;\; gu = gu + 2n\varepsilon * bnorm + \xi_n.$$

then we can claim

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

For the algorithms we mentioned in the previous subsections, 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 9). For instance, the error bound of `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): $\bar{M} \ge \omega/\varepsilon$, we have

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

According to Table 9, if we let

$$gl = gl - (10n+6)\varepsilon * bnorm; \;\; gu = gu + (10n+6)\varepsilon * bnorm. \tag{5.9}$$

Then we have

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

in all situations, which shows the Gerschgorin Bound (5.9) is correct for EISPACK's `bisect`, LAPACK's `sstebz`, IEEE routine and Best Prescaling Algorithm.

5.8 The Splitting Criterion

The splitting criterion asks if an offdiagonal entry b_i is small enough in magnitude to set to zero without making unacceptable perturbations in the eigenvalues. This is useful because setting b_i to zero splits T into two independent subproblems which can be solved faster (serially or in parallel). If we are only interested in absolute accuracy, then Weyl's Theorem [20] guarantees that the test

$$\text{if } |b_i| \le tol \text{ then } b_i = 0 \tag{5.10}$$

will not change any eigenvalue by more than *tol*. An alternative test for setting b_i to zero is

$$\begin{array}{rcl} \delta & = & (a_{i+1} - a_i)^2/4 \\ \rho & = & (1 - 2^{-1/2})(b_{i-1}^2 + b_{i+1}^2) \\ \tau & = & \dfrac{b_i^2}{\delta + \rho}(2\rho + \dfrac{\delta b_i^2}{\delta + \rho}) \\ \text{if} & & \tau < tol^2 \text{ then } b_i = 0 \end{array}$$

This also guarantees that no eigenvalue will change by more than *tol* (in fact it guarantees that the square root of the sum of the squares of the changes in all the eigenvalues is bounded by *tol*) [15, 20]. Although it sets b_i to zero more often than the simpler test (5.10), it is much more expensive.

To guarantee relative accuracy, we need the following new result:

Lemma 5.2 *Let T be a tridiagonal matrix where for a fixed i*

$$|b_i| \le tol \cdot (|a_i a_{i+1}|)^{1/2}$$

Let $T' = T$ except for setting $b_i' = 0$. Then there exist other tridiagonal matrices T_1 and T_2 with the following properties:

i. $\lambda_{1i} \le \lambda_i' \le \lambda_{2i}$, *where T_j has eigenvalues $\lambda_{j1} \le \cdots \le \lambda_{jn}$.*

ii. $\lambda_{1i} \le \lambda_i \le \lambda_{2i}$,

iii. $T_1 = T_2 = T$ *except for entries* (i, i) *and* $(i+1, i+1)$ *which differ from those of T by factors $1 \pm tol$.*

In other words, the eigenvalues of T' and T lie between the eigenvalues of matrices T_1 and T_2, where the entries of T_j approximate those of T with relative accuracy *tol*. This is a nearly unimprovable backward error bound. Combined with [5, Theorem 4], this easily yields

Corollary 5.1 *Let T be γ-s.d.d., and suppose $tol < (1-\gamma)/(1+\gamma)$ (see [5] for definitions). Suppose $|b_i| \le tol \cdot (|a_i a_{i+1}|)^{1/2}$, and let $T' = T$ except for $b_i' = 0$. Then*

$$\exp\left(\frac{-tol}{1 - \gamma\frac{1+tol}{1-tol}}\right) \le \frac{\lambda_i'}{\lambda_i} \le \exp\left(\frac{tol}{1 - \gamma\frac{1+tol}{1-tol}}\right)$$

When $tol \ll 1 - \gamma$ then

$$1 - \frac{tol}{1-\gamma} \le \frac{\lambda_i'}{\lambda_i} \le 1 + \frac{tol}{1-\gamma}$$

PROOF OF LEMMA 5.2. By the Courant Minimax Theorem [20] it suffices to construct T_1 and T_2 satisfying condition iii in the Lemma such that for all vectors x

$$x^T T_1 x \leq x^T T' x \leq x^T T_2 x \;\text{ and }\; x^T T_1 x \leq x^T T x \leq x^T T_2 x \tag{5.11}$$

Since all the matrices differ only in the i-th and $i+1$-st rows and columns, it suffices to consider two by two matrices only:

$$T = \left[\begin{array}{cc} a_i & b_i \\ b_i & a_{i+1} \end{array} \right] , \; T' = \left[\begin{array}{cc} a_i & 0 \\ 0 & a_{i+1} \end{array} \right] ,$$

We claim that the following matrices satisfy condition (5.11):

$$T_1 = \left[\begin{array}{cc} a_i - tol|a_i| & b_i \\ b_i & a_{i+1} - tol|a_{i+1}| \end{array} \right] , \; T_2 = \left[\begin{array}{cc} a_i + tol|a_i| & b_i \\ b_i & a_{i+1} + tol|a_{i+1}| \end{array} \right]$$

To prove this requires us to verify that

$$\left[\begin{array}{cc} \pm tol|a_i| & 0 \\ 0 & \pm tol|a_{i+1}| \end{array} \right] \text{ and } \left[\begin{array}{cc} \pm tol|a_i| & b_i \\ b_i & \pm tol|a_{i+1}| \end{array} \right]$$

are positive (negative) semidefinite, which is immediately implied by the assumption of the lemma. □

This leads us to recommend the splitting criterion

$$\text{if } |b_i| \leq tol|a_i a_{i+1}|^{1/2} \text{ then } b_i = 0 \tag{5.12}$$

since this does not change the eigenvalues any more than relative perturbations of size *tol* in the matrix entries. Note that it will never be applied to tridiagonals with zero diagonal unless b_i is exactly zero.

This criterion is more stringent than the criterion in the EISPACK code bisect [21], which essentially is

$$\text{if } |b_i| \leq tol(|a_i| + |a_{i+1}|) \text{ then } b_i = 0. \tag{5.13}$$

Note that this is at least about as stringent as the absolute accuracy criterion (5.10) but less stringent than the relative accuracy criterion (5.12). To see that it can fail to deliver high relative accuracy when (5.12) will succeed, consider the example

$$\left[\begin{array}{cc} 10^{20} & 5 \cdot 10^9 \\ 5 \cdot 10^9 & 1 \end{array} \right]$$

which is γ-s.d.d. with $\gamma = .5$. Let $tol = 10^{-10}$. Then EISPACK would set the offdiagonals to zero, returning eigenvalues 10^{20} and 1. The true eigenvalues (to about 20 digits) are 10^{20} and .75.

Note that criterion (5.12) could possibly be used as the stopping criterion in a QR algorithm [5] in the hopes of attaing high relative accuracy. However, the rounding errors in any existing QR algorithm generally cause far more inaccuracy in the computed eigenvalues than the currently used stopping criteria (which are generally identical to (5.13)).

6 Proof of Monotonicity of Count(x)

In 1966 Kahan proved but did not publish the following result [16]: 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 FloatingCount($x$) $= 1 > 0 =$ FloatingCount(x'). In this section, we will extend this proof of monotonicity of FloatingCount(x) to symmetric acyclic matrices.

In section 2.1, we mentioned that we will number the rows and columns of T in preorder, as they are accessed by Algorithm 2 (see section 5). This means node 1 is the root of the tree, since it is accessed first, and children are numbered higher than their parents. This lets us relabel the variables in Algorithm 2 as follows, where we also introduce roundoff:

Algorithm 6: Count(x) **returns the number of eigenvalues of the symmetric acyclic matrix T that are less than x.**

```
call TreeCount($1, x, d_1, s_1$)
Count = $s_1$

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($j, x, d_j, s_j$)
        $d_i = fl(d_i - fl(T_{ij}^2/d_j))$
        $s_i = s_i + s_j$
    endfor
    if $d_i < 0$ then $s_i = s_i + 1$
end TreeCount
```

(Without loss of generality we ignore roundoff in computing T_{ij}^2.) Thus s_i is the total number of negative d_j in the subtree rooted at i (including d_i). We may summarize Algorithm 6 more briefly by

$$d_i = fl(fl(T_{ii} - x) - fl(\sum_{j \in C(i)} fl(\frac{T_{ij}^2}{d_j}))) \tag{6.14}$$

$$s_i = \sum_{j \in C(i)} s_j + \begin{cases} 0 & \text{if } d_i \geq 0 \\ 1 & \text{if } d_i < 0 \end{cases} \tag{6.15}$$

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 6 for different x we will write $s_i(x)$ and $d_i(x)$. The theorem we wish to prove is:

Theorem 6.1 *If the floating point arithmetic used to implement Algorithm 6 is monotonic, then $s_i(x) \leq s_i(x')$.*

We introduce some more definitions. In these definitions, y is always a floating point number. The number y is a *zero* of d_i if $d_i(y) \geq 0 > d_i(y')$. 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 $d_i(y') = 0$.

Suppose that some s_i is decreasing; we want to find a contradiction.

Lemma 6.1 *Let m be the largest m such that s_m is decreasing. 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 (6.15) 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 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

$$fl(\frac{T_{ij}^2}{d_j(y)}) > fl(\frac{T_{ij}^2}{d_j(y')}) \tag{6.16}$$

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 (6.16) 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 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 6; this is done in practice to avoid overflow and division by zero; see Algorithm 5 and [1, 16]. However, the proof does *not* work for the algorithm used to avoid overflow in the subroutine `bisect` [21]. This is because `bisect` tests if a computed d_i is exactly zero, and increases if it is; this can increase $d_i(y')$ past $d_i(y)$ even if inequalities (6.16) are satisfied. The example in section 4.2 shows that monotonicity can indeed fail in practice.

Lemma 6.3 *If y is a pole of d_i, then there must be a 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 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 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. □

Now use Lemma 6.1 to conclude that there is an m such that y is a negative pole of d_m, and

$$\sum_{k\in C(m)} s_k(y) = \sum_{k\in C(m)} s_k(y') \quad . \tag{6.17}$$

Use Lemma 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 (6.17) than to the left hand side. So to maintain (6.17) there must be another p in the tree rooted at m with $d_p(y) < 0 \leq d_p(y')$, i.e. y is a negative pole of d_p. By Lemma 6.3, p cannot lie on the path from m to l, since only positive poles lie on this path. Therefore, again by Lemma 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 (6.17), and so cannot balance d_l. By the same argument, any other negative pole which would balance d_l has a counterbalancing zero. Therefore (6.17) cannot be satisfied. This contradiction proves Theorem 6.1.

7 Proofs of Correctness

We now present the proofs of the various theorems stated in Section 4.

Proof of Theorem 4.1 To prove the theorem, we first prove the inductive assertion that for every task $(\alpha, \beta, n_\alpha, n_\beta, I_{n_\alpha}^{n_\beta})$ in the $Worklist$, FloatingCount$(\alpha) \le n_\alpha < n_\beta \le$ FloatingCount(β). The inductive assertion is clearly true for the initial task in the $Worklist$ for which FloatingCount$(\alpha) \le n_\alpha < n_\beta \le$ FloatingCount(β). Suppose that the inductive assertion holds for some task. We prove that the assertion holds for the subtasks created by this task. Statement 10 in Figure 2 ensures that $n_\alpha \le n_{mid} \le n_\beta$. A new subtask is added to the $Worklist$ if

i. $n_\alpha < n_{mid}$. Statement 10 implies that $n_{mid} = \min(\text{FloatingCount}(mid), n_\beta)$ and therefore, $n_{mid} \le$ FloatingCount(mid). Thus, FloatingCount$(\alpha) \le n_\alpha < n_{mid} \le$ FloatingCount(mid), and the inductive assertion holds for the new subtask $(\alpha$,mid,n_α, n_{mid},$I_{n_\alpha}^{n_{mid}})$.

ii. $n_{mid} < n_\beta$, which implies that $n_{mid} \ge$ FloatingCount(mid). Thus, FloatingCount(mid) $\le n_{mid} < n_\beta \le$ FloatingCount(β), and the inductive assertion holds for the new subtask $(mid$,β,n_{mid},n_β,$I_{n_{mid}}^{n_\beta})$.

Hence our inductive assertion holds for any task in the $Worklist$. Let $\lambda' = \min(\max(fl(\frac{\alpha+\beta}{2}), \alpha), \beta)$ be an eigenvalue output by Algorithm SER_BISEC. For simplicity, we assume that the eigenvalue is of multiplicity 1, i.e., $n_\beta = n_\alpha + 1$. The inductive assertion implies that there exists at least one n_β^{th} jump-point of FloatingCount(x), λ''_{n_β}, in $(\alpha, \beta]$. By the working of the algorithm, $|\lambda' - \lambda''_{n_\beta}| \le \tau_{n_\beta}$, and by the assumption about FloatingCount(x), $|\lambda''_{n_\beta} - \lambda_{n_\beta}| \le \xi_{n_\beta}$. By the triangle inequality, $|\lambda' - \lambda_{n_\beta}| \le \tau_{n_\beta} + \xi_{n_\beta}$, and hence the computed eigenvalues are correct to within the user specified error tolerance. The proof when $n_\beta > n_\alpha + 1$ is similar. By Assumption 1, *fl(inside*$(\alpha, \beta)) \in (\alpha, \beta)$ for all α, β that arise (note that $\alpha \le \beta$ for all tasks in the $Worklist$). Thus, all subtasks have strictly smaller intervals, and the algorithm must terminate. At any stage of the algorithm, the index sets corresponding to all the tasks in the $Worklist$ form a partition of the initial index set, $I_{n_{left}}^{n_{right}}$. Each index i is contained in exactly one index set and hence, each desired eigenvalue is computed exactly once. It is also clear that the computed eigenvalues are in sorted order. □

7.1 A Necessary and Sufficient Condition for Correctness

Having found a rather simple fix to the problem of nonmonotonicity in serial bisection, we now prove a necessary and sufficient condition for the correctness of any serial or parallel implementation of the bisection algorithm. First, we define a few terms to help us in the ensuing discussion. Consider a task $T = (\alpha, \beta, n_\alpha, n_\beta, O)$. We say that this task *covers* the index set O. If tasks $T_1, \ldots, T_k$ cover the index sets $O_1, \ldots, O_k$ respectively, then the set of tasks $\{T_1, \ldots, T_k\}$ is said to *cover* the index set $O_1 \cup O_2 \cdots \cup O_k$. We define an *Index Cover* to be a set of tasks which covers the user index set U. A *Disjoint Index Cover* is an index cover such that the index sets covered by any pair of tasks in the index cover are disjoint.

We assume that all the bisection algorithms discussed below maintain a set of tasks (either explicitly or implicitly). Each task in this set, $(\alpha, \beta, n_\alpha, n_\beta, O)$, is assumed to be such that $\alpha \leq \beta$ and FloatingCount$(\alpha) \leq n_\alpha < n_\beta \leq$ FloatingCount(β). When the width of an interval corresponding to some task becomes smaller than $\min_{i=n_\alpha+1}^{n_\beta} \tau_i$, this task is marked as a *final* task, and is not further refined. We will refer to an interval corresponding to a *final* task as a *final* interval. At the end of the algorithm, the midpoints of the *final* intervals are output as the eigenvalues corresponding to the index sets covered by the respective *final* tasks.

Consider the two tasks $T_1 = (\alpha_1, \beta_1, n_{\alpha_1}, n_{\beta_1}, O_1)$, and $T_2 = (\alpha_2, \beta_2, n_{\alpha_2}, n_{\beta_2}, O_2)$. Suppose that $\min(\max(fl((\alpha_1 + \beta_1)/2), \alpha_1), \beta_1) \leq \min(\max(fl((\alpha_2 + \beta_2)/2), \alpha_2), \beta_2)$. We say that the pair T_1 and T_2 is ordered if $\forall i \in O_1, \forall j \in O_2, i \leq j$. A set of tasks is said to be ordered if every pair of tasks in this set is ordered.

Theorem 7.1 *A bisection algorithm is correct iff its final tasks form an ordered disjoint index cover.*

PROOF. Suppose the *final* tasks form an ordered disjoint index cover. Consider a *final* task $(\alpha, \beta, n_\alpha, n_\beta, O)$ where $O \subseteq I_{n_\alpha}^{n_\beta}$. Then $\forall i \in O$, the eigenvalues output from this task are $\lambda_i' = \min(\max(fl((\alpha+\beta)/2), \alpha), \beta)$. Since $i \in I_{n_\alpha}^{n_\beta}$, the interval (α, β) contains an i^{th} jump-point,λ_i'', of FloatingCount(x). Hence the reported eigenvalue λ_i' is such that $|\lambda_i' - \lambda_i''| \leq \tau_i$. By our assumptions about FloatingCount(x), $|\lambda_i - \lambda_i''| \leq \xi_i$. Thus $|\lambda_i - \lambda_i'| \leq \tau_i + \xi_i$, and every eigenvalue output is computed correctly. Since the *final* tasks form a disjoint index cover, every desired eigenvalue is output exactly once. All *final* tasks are ordered, hence the desired eigenvalues are output in sorted order.

If the *final* tasks do not form an index cover then at least one of the desired eigenvalues will not be output. If any two *final* tasks cover intersecting index sets, then some eigenvalue will be output more than once, and if some pair of *final* tasks is not ordered, then the eigenvalues output will not be in sorted order. Hence, it is necessary for the *final* tasks to form an ordered disjoint index cover. □

It is easy to verify that Algorithm SER_BISEC satisfies the sufficient conditions of Theorem 7.1. Note that the eigenvalues output will be correct if the *final* tasks form an index cover.

7.2 Further Proofs

We now use the above characterization of correct bisection algorithms to prove the correctness of the parallel algorithms given in Section 4.4

Proof of Theorems 4.2 and 4.3 The initial interval $(\alpha(i), \beta(i)]$ computed by each processor is such that $\alpha(i) \leq \beta(i)$. Also $\beta(i) = \alpha(i+1)$, $n_{\beta(i)} = n_{\alpha(i+1)}$ for $i = 0, \ldots, p-2$, $\alpha(0) = gl$, $n_{\alpha(0)} = 0$, $\beta(p-1) = gu$ and $n_{\beta(p-1)} = n$. Thus, the initial tasks that are input to Algorithm SER_BISEC on all p processors, $(\alpha(i)$,$\beta(i)$,$n_{\alpha(i)}$,$n_{\beta(i)}$,$I_{\alpha(i)}^{\beta(i)})$, form an ordered index cover.

In algorithm PAR_ALLEIG1, $\alpha(i) \leq \beta(i)$ implies that $n_{\alpha(i)} \leq n_{\beta(i)}$ if FloatingCount(x) is monotonic. By the way in which processor 0 computes these quantities, $n_{\alpha(i)} \leq n_{\beta(i)}$ in algorithm PAR_ALLEIG2. Thus, the index cover produced by both the above algorithms is disjoint.

The correctness of algorithm Ser_Bisec implies that all the *final* tasks form an ordered disjoint index cover, and hence proves the correctness of algorithms Par_AllEig1 and Par_AllEig2 (by theorem 7.1). □

Proof of Theorem 4.4 We consider the initial tasks input to Algorithm Ser_Bisec on each processor. We first observe that $\alpha(0) = gl$, $n_{\alpha(0)} = 0$, and $\beta(p-1) = gu$, $n_{\beta(p-1)} = n$. For $i = 0, \ldots, p-2$, it can be seen that $n_{\beta(i)} = n_{\alpha(i+1)}$. Also, $\alpha(i) = \gamma(i-1) \leq \beta(i)$, and $n_{\alpha(i)} \leq n_{\beta(i)}$ for $i = 0, \ldots, p-1$. This is because FloatingCount(x) is *forced* to be monotonic in function Find_Init_Task. Thus $\beta(i)$ is no smaller than $\alpha(i)$, which is communicated by processor $i-1$. Thus, all the initial tasks input to Algorithm Ser_Bisec form a disjoint index cover. It can also be checked that this index cover is ordered. The above statements rely on the assumption that identical floating point operations on different processors yield bitwise identical results (see Assumption 1). Hence by the correctness of Algorithm Ser_Bisec, Algorithm Par_AllEig3 is correct. □

7.3 A Family of Correct Parallel Bisection Implementations

We now prove the correctness of a family of bisection algorithms $\mathcal{F}$. Every algorithm in this family starts out with one task which covers the user index set U. All tasks are obtained by refining existing tasks in the task set. A task $T = (\alpha, \beta, n_\alpha, n_\beta, O)$ with $\alpha \leq \beta$ is removed from the task set and is refined to form the k subtasks $(\alpha_1, \alpha_2, n_{\alpha_1}, n_{\alpha_2}, O_1), \ldots, (\alpha_k, \alpha_{k+1}, n_{\alpha_k}, n_{\alpha_{k+1}}, O_k)$, where $\alpha_0 = \alpha$, $\alpha_{k+1} = \beta$, $\alpha_i \leq \alpha_{i+1}$ and $n_{\alpha_i} \leq n_{\alpha_{i+1}}$, for $i = 1, \ldots, k$. Furthermore, $\cup_{i=1}^{k} O_i = O$, and $O_i = I_{n_{\alpha_i}}^{n_{\alpha_{i+1}}} \cap O$, for $i = 1, \ldots, k$. Note that a task may be refined by doing k-way multisection or by a single iteration of a fast root finder, such as Newton's or Laguerre's iteration. These subtasks are now added to the task set(some of them being marked *final* and not further refined). The tasks in the task set may be processed by one or more processors. By the manner in which the tasks are refined, it is easy to see that at each step of the algorithm, the tasks in the task set form an ordered disjoint index cover. In particular, the *final* tasks form an ordered disjoint index cover. Hence, using Theorem 7.1 we get :

Theorem 7.2 *Every bisection algorithm from the family $\mathcal{F}$ is correct if Assumption 1 holds.*

Algorithms Ser_Bisec and Par_AllEig1 are easily seen to belong to the above family. Algorithm Ser_Bisec may be modified simply to yield a parallel algorithm, where all the enqueuing and dequeing of tasks is done from a global *Worklist* that is distributed across all the processors. Such an algorithm has been implemented on the CM-5 [13] — the work is initially partitioned among the processors (as in algorithms Par_AllEig1 and Par_AllEig3), and load imbalance is reduced by enqueuing and dequeing tasks from other processors. This algorithm has been observed to give good performance even when the initial partitioning of work is not good. Such an algorithm also belongs to the family $\mathcal{F}$, and hence is *correct*.

8 Practical Implementation Issues

In section 5.3, we introduced the IEEE routine (Algorithm 4) which has no explicit tests and branches in the inner loop. However, there are some practical issues we need to consider. In this section, we will briefly discuss three topics: SignBit, division by zero and over/underflow.

- SignBit.

  An acceptable way to compute SignBit(d) in Fortran would be

  $$\text{SignBit}(d) = 0.5 - \text{SIGN}(0.5, d)$$

  except that we need not a REAL number but an INTEGER to add to Count, so extra time would have to be spent upon REAL-to-INTEGER conversion. In the language C, the expression "($d < 0.0$)" could be used in place of "SignBit(d)". However, both of these expedients produce SignBit(-0.0) = 0, and that can cause function Count(x) to malfunction on a few aberrant computer designs.

  However, if we do some preprocessing work, like $a_i = a_i + 0$, before entering the function Count(x), then all the d_i's will never become -0 no matter whether by exact cancellation or underflow, on any commercially significant computer regardless of whether it conforms to IEEE standard 754 or 854 for Floating-PointArithmetic. But on machines that almost conform to such a standard, departing from it only in that they may force underflowed subtractions to -0.0, function Count requires that SignBit(-0.0) = 1 in order to account properly for the sign of $\pm\infty$ produced subsequently by division by -0.0. On such computers SignBit must be implemented in one of the following ways, which are optional for other computers.

  IEEE Standards 754 & 854 recommend that conforming computers provide a function CopySign which we may use safely in place of Fortran's SIGN function to implement

  $$\text{SignBit}(d) = 0.5 - \text{CopySign}(0.5, d).$$

  Through an unfortunate accident, the arguments of CopySign have been reversed on Apple computers, which otherwise conform conscientiously to IEEE 754; they require SignBit(d) = 0.5 − CopySign(d, 0.5).

  Hardly any other computer's compilers' libraries offer CopySign.

  All computers can compute SignBit(d) quickly by shifting the sign bit of d *logically* into the rightmost bit position of an integer register leaving zeros in all the other bits. Equally good is a *twos-complement arithmetic* right shift that fills the register with copies of the sign bit, thereby producing −SignBit(d). Can either of these shifts be expressed in a higher-level language in such a way as will achieve the desired effect on every computer? Two obstacles get in the way.

  The first obstacle is a disparity of widths. The INTEGER variables Count and n are likely to be 2 or 4 bytes wide. (The algorithm for Count(x) can cope with matrices T of as big a dimension n as memory capacity allows.) The REAL variable d may be 4 bytes wide but is most likely 8. INTEGERs 8 bytes wide are not in common use, so d

# of Bytes	Fortran declarations	C declaration[2]
4	REAL, SINGLE PRECISION, REAL*4	float
8	DOUBLE PRECISON, REAL*8	double
10	EXTENDED, TEMPREAL, REAL*10	long double

Table 10: Width of real variable

is probably wider than the widest INTEGERs supported by the compiler. Therefore the leading(leftmost) 2 or 4 bytes of d must be first located and then extracted as an integer before the shift. Most computers, with separate registers for INTEGERs and REALs, must first store d in memory and then reload it into an integer register.

Unfortunately, different computers order the bytes of d differently. Motorola 68040s give d and the byte with its sign the same address. Intel 486s must add ((width of d) -1) to the address of d to find the byte with d's sign; the width of d in bytes can be found in table 9.

MIPS microprocessors can match either of the first two above. (Who gets to choose?) DEC VAXs do something a little bit different again. These diverse byte orderings constitute a second obstacle impeding efficient and portable programming of the SignBit function.

Both obstacles can be overcome to a degree by Conditional Compilation in C using #define and #ifdef commands in its preprocessor to find out whether the computer to which the program is being compiled belongs to a previously recognized family for which an efficient sequence of instructions has been prepared. This expedient fails to cope with new computers whose C compilers proclaim conformity with all applicable standards but whose arithmetic properties and memory-register mappings were unknown at the time the program for Count(x) was promulgated. A better solution to this problem is to include appropriately defined CopySign functions in language standards; CopySign should reveal the sign of -0.0 on a computer whose arithmetic respects it, and hide that sign on a computer whose arithmetic ignores it, and return both REAL and INTEGER values according to the type of its first argument.

- Division by Zero
  IEEE 754 & 854 require by default (unless the programmer explicitly requests otherwise) that "nonzero/zero" quotients be computed as appropriately signed infinities. Of course, "finite/infinite" quotients must produce appropriately signed zeros. Function IEEE Count(x) works perfectly under these conventions; that is why its program contains no test to avert division by zero. A test like that is necessary on computers that can not tolerate division by zero, but wastes time because division by zero is unlikely to occur by accident as often as once in a million passes around the inner loop, and is certain to be noticed by the computer if it does occur.

  A little known alternative has long existed for users of proprietary Fortran compilers on IBM /370s and DEC VAXs; their programs may request that "nonzero/zero" quotients deliver the computer's biggest floating-point magnitude with the numerator's sign. This works almost as well as ∞ would in the program above for Count(x).

Unfortunately, most computers that do conform to IEEE 754/854 treat division by zero no better than nonconforming computers. The trouble is linguistic; language designers and compiler writers have yet to agree upon standard ways for programmers to request IEEE standards' default infinities or IBM's ot DEC's biggest magnitude. Instead, division by zero is left undefined or defined as an error; either way, computation stops.

- Over/Underflow
  Computers that abort computation when overflow occurs present the same problems as those that stop on division by zero. Once again, ways exist to tell any commercially significant computer to replace every overflow by either ∞ or the biggest finite floating-point number with an appropriate sign, but no higher level programming language provides a single way that works for every computer.

Inattention to troublesome details by designers and implementors of programming languages creates headaches for programmers would-be portable (reusable) programs. The details in question here are the CopySign function and humane exception-handling. To get around the lack of adequate language standards, programmers must avoid those details by inserting extra tests and branches into their programs. The annoyance at having to complicate so simple a program is compounded by the performance penalty incurred by data-dependent branches taken rarely, especially on massively parallel and vectorized computers.

9 Conclusions

We have proved necessary and sufficient conditions for a bisection algorithm to be *correct*. We have also seen examples of natural serial and parallel implementations that are *incorrect*, the errors arising from a nonmonotonic FloatingCount(x) and/or roundoff. Thus every bisection implementation must be carefully analyzed and proven to satisfy the sufficient conditions for *correctness*.

Acknowledgements

The authors acknowledge the many contributions of W. Kahan, especially to section 8.

References

[1] 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.

[2] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[3] ANSI/IEEE, New York. *IEEE Standard for Radix Independent Floating Point Arithmetic*, Std 854-1987 edition, 1987.

[4] M. Assadullah, J. Demmel, S. Figueroa, A. Greenbaum, and A. McKenney. On finding eigenvalues and singular values by bisection. LAPACK Working Note. in preparation.

[5] J. Barlow and J. Demmel. Computing accurate eigensystems of scaled diagonally dominant matrices. *SIAM J. Num. Anal.*, 27(3):762–791, June 1990.

[6] H. Bernstein and M. Goldstein. Parallel implementation of bisection for the calculation of eigenvalues of a tridiagonal symmetric matrices. Technical report, Courant Institute, New York, NY, 1985.

[7] J. Demmel. Underflow and the reliability of numerical software. *SIAM J. Sci. Stat. Comput.*, 5(4):887–919, Dec 1984.

[8] J. Demmel and W. Gragg. On computing accurate singular values and eigenvalues of acyclic matrices. *Lin. Alg. Appl.*, 185:203–218, 1993.

[9] 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.

[10] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990.

[11] J. Demmel and X. Li. Faster numerical algorithms via exception handling. In M. J. Irwin E. Swartzlander and G. Jullien, editors, *Proceedings of the 11th Symposium on Computer Arithmetic*, Windsor, Ontario, June 29 – July 2 1993. IEEE Computer Society Press. to appear in IEEE Trans. Comp.; available as all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/cs/csd-93-728; software is csd-93-728.shar.Z.

[12] J. Demmel and H. Ren. The instability and nonmonotonicity of the parallel prefix algorithm. in preparation, 1994.

[13] I. S. Dhillon and J. W. Demmel. A parallel algorithm for the symmetric tridiagonal eigenproblem and its implementation on the CM-5. In progress, 1993.

[14] Y. Huo and R. Schreiber. Efficient, massively parallel eigenvalue computations. preprint, 1993.

[15] W. Kahan. When to neglect offdiagonal elements of symmetric tridiagonal matrices. Computer Science Dept. Technical Report CS42, Stanford University, Stanford, CA, July 1966.

[16] W. Kahan. Accurate eigenvalues of a symmetric tridiagonal matrix. Computer Science Dept. Technical Report CS41, Stanford University, Stanford, CA, July 1966 (revised June 1968).

[17] W. Kahan. Analysis and refutation of the International Standard ISO/IEC for Language Compatible Arithmetic. SIGNUM Newsletter and SIGPLAN Notices, 1991.

[18] S.-S. Lo, B. Phillipe, and A. Sameh. A multiprocessor algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):155–165, March 1987.

[19] R. Mathias. The stability of parallel prefix matrix multiplication with applications to tridiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 1993. submitted.

[20] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[21] 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.

[22] J. H. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford University Press, Oxford, 1965.