LAPACK WORKING NOTE 172: BENEFITS OF IEEE-754 FEATURES IN MODERN SYMMETRIC TRIDIAGONAL EIGENSOLVERS

OSNI A. MARQUES[*], E. JASON RIEDY[‡], AND CHRISTOF VÖMEL[*]

Technical Report UCB//CSD-05-1414
Computer Science Division
UC Berkeley
September 30, 2005

Abstract. Bisection is one of the most common methods used to compute the eigenvalues of symmetric tridiagonal matrices. Bisection relies on the *Sturm count*: for a given shift σ, the number of negative pivots in the factorization $T - \sigma I = LDL^T$ equals the number of eigenvalues of T that are smaller than σ. In IEEE-754 arithmetic, the value ∞ permits the computation to continue past a zero pivot, producing a correct Sturm count when T is unreduced. Demmel and Li showed in the 90s that using ∞ rather than testing for zero pivots within the loop could improve performance significantly on certain architectures.

When eigenvalues are to be computed to high relative accuracy, it is often preferable to work with LDL^T factorizations instead of the original tridiagonal T, see for example the MRRR algorithm. In these cases, the Sturm count has to be computed from LDL^T. The differential stationary and progressive qds algorithms are the methods of choice.

While it seems trivial to replace T by LDL^T, in reality these algorithms are more complicated: in IEEE-754 arithmetic, a zero pivot produces an overflow, followed by an invalid exception (NaN), that renders the Sturm count incorrect.

We present alternative, safe formulations that are guaranteed to produce the correct result.

Benchmarking these algorithms on a variety of platforms shows that the original formulation without tests is always faster provided no exception occurs. The transforms see speed-ups of up to 2.6× over the careful formulations.

Tests on industrial matrices show that encountering exceptions in practice is rare. This leads to the following design: First, compute the Sturm count by the fast but unsafe algorithm. Then, if an exception occurred, recompute the count by a safe, slower alternative.

The new Sturm count algorithms improve the speed of bisection by up to 2× on our test matrices. Furthermore, unlike the traditional tiny-pivot substitution, proper use of IEEE-754 features provides a careful formulation that imposes no input range restrictions.

AMS subject classifications. 15A18, 15A23.

Key words. IEEE-754 arithmetic, NaN arithmetic, IEEE-754 performance, differential qds algorithms, MRRR algorithm, LAPACK.

1. Introduction.

In the early 90's, Demmel and Li showed how adequate exception handling can improve the performance of certain numerical algorithms [8]. One such algorithm is bisection, using Sturm counts to compute the eigenvalues of a symmetric tridiagonal matrix T. Computing the Sturm counts can encounter a zero pivot while factoring $T - \sigma I = LDL^T$. A conventional bisection implementation like LAPACK's DSTEBZ substitutes a small number for zero and continues the computation [3]. This procedure has two disadvantages. Testing imposes a performance penalty, and ensuring correctness requires a range restriction on the input data.

[*]Lawrence Berkeley National Laboratory, 1 Cyclotron Road, MS 50F-1650, Berkeley, CA 94720, USA. {oamarques,cvoemel}@lbl.gov

[‡]Computer Science Division, University of California, Berkeley, CA 94720, USA. ejr@cs.berkeley.edu

[§]This work has been supported by a grant from the National Science Foundation (Cooperative Agreement no. ACI-9619020) and from the Department of Energy's SciDAC program (DE FC02 01ER25478).

IEEE-754 arithmetic [4, 21] provides the *special value* ∞ ('Infinity') to continue past the exception of an overflow or division by zero. This quantity propagates like any other floating point value, ideally at full speed. Hence, the capabilities of IEEE-754 arithmetic allow a faster way of implementing tridiagonal bisection.

Since [8], there has been significant progress in algorithms for the symmetric eigenvalue problem. In particular, LAPACK now includes DSTEGR, an implementation of the MRRR algorithm [23, 24, 25, 26, 9]. Two key concepts of MRRR are its use of shifted LDL^T factorizations instead of the original tridiagonal matrix T, and the *differential stationary* and the *progressive qds* algorithms that transform one shifted factorization into another. These transformations have several useful applications in MRRR, including computing some eigenvalues by bisection. This paper focuses on using the qds algorithms in bisection. Originally, the differential qd (dqd) algorithm was proposed by Rutishauser [28, 29]. Parlett and Fernando [14, 15, 22] discovered the shifted variant (dqds) as an improvement to Demmel and Kahan's zero shift QR [7] for computing singular values of a bidiagonal matrix to high relative accuracy. It is implemented as LAPACK's DLASQ1, see also [27].

Bisection on the factored matrix LDL^T using these stationary and progressive differential qds transformations is more complicated than the traditional version using the tridiagonal T. A qds implementation in IEEE-754 arithmetic can encounter not only ∞ but also an invalid operation ∞/∞ (producing NaN, or 'Not A Number') that renders the Sturm count invalid. Demmel and Li [8] suggested a paradigm for similar cases.

- Use a fast (but possibly incorrect) algorithm to compute a result.
- Assess the correctness of the computed result.
- If the answer is not correct, use a slower more careful algorithm to recompute the result.

The goal of this paper is to quantify the usefulness of the aforementioned paradigm in this novel context and on today's computer architectures. This allows us to select the best bisection variant for use in the MRRR algorithm.

The paper is organized as follows. Section 2 gives an overview of the various formulations of Sturm counts. We analyze three always correct alternatives to the basic version. One of the alternative formulations computes a correct Sturm count without any additional input range restrictions.

Section 3 is the heart of this paper. We benchmark our algorithms for inputs resulting in both exceptional (∞ followed by NaN) and unexceptional data and quantify the benefits of using IEEE-754 features and the penalties for encountering exceptions. As long as no exception occurs, the basic version is always faster. Overall, the careful variants' times are around $1.12\times$ the time of the the basic version. We refer to this as a speed-up of 1.12. When an exception occurs, the combined plain/careful computation times are about $1.63\times$ that of the careful loop alone, which we call a slow-down of 1.63. There is one remarkable exception, the Pentium 4 Xeon, whose performance drops by factors of 18-60$\times$ in the exceptional case.

Section 4 compares these formulations when applied within bisection on real world test examples. The median speed-up of the fast algorithm when running bisection to determine the eigenvalues fairly accurately is moderate, around 1.08. The Cray X1 greatly benefits from moving exceptional tests out of the inner loop, seeing speed-ups from 1.2 to almost 2.

The potential for huge slow-downs on aberrant architectures implies that our paradigm may not be suitable for portable code. However, Section 5 describes a technique to limit the penalty on almost all cases: blocking. The Sturm count computation can be divided into blocks, and the NaN test is performed on each block. Thus, if a NaN occurs, only the computation of the previous block need be repeated.

The Appendix A gives a short overview of important implementation details in our algorithms.

2. Sturm counts with and without IEEE-754 arithmetic. Bisection is one of the most popular methods for computing the eigenvalues of a symmetric tridiagonal matrix. It can be used to find k selected eigenvalues with $\mathcal{O}(kn)$ work given a tridiagonal matrix of dimension n. Furthermore, it is easily parallelizable.

Bisection is based on Sturm counts. This section presents various methods for computing Sturm counts. We require that the input matrix is unreduced; that is, no off-diagonal entry is zero. This implies that the eigenvalues are simple. Additionally, we assume no input is NaN or $\pm\infty$.

2.1. Sturm counts for a tridiagonal matrix. Sylvester's Law of Inertia states that the (shifted) symmetric tridiagonal matrix $T - \sigma I = LDL^T$, with L being unit-bidiagonal and D being diagonal, has the same number of positive, negative, and zero eigenvalues as D, see for example [16] .

This result can be used to find, by suitable variation of the shifts, small enclosing intervals for each eigenvalue of T. Starting with an initial interval $[VL, VU]$ containing the eigenvalue, we compute the number of negative pivots at the midpoint $(VL + VU)/2$. If this number is smaller than the index of the wanted eigenvalue, the eigenvalue lies in the right half of the interval, otherwise in the left half. This bisection process can be repeated until an interval of suitably small size is found. Its implementation in finite precision arithmetic is correct provided the Sturm count is monotonic with σ, see [19, 6].

The well known recurrence for the pivots in D is given by

$$d(i) = (T_{ii} - \sigma) - \frac{T_{ii+1}^2}{d(i-1)}. \tag{2.1}$$

In finite precision arithmetic, the pivot $d(i-1)$ may be tiny or exactly zero. In this case, the succeeding pivot $d(i)$ overflows (provided that $T_{ii+1} \neq 0$).

A possible cure to this situation was first developed in [19]. At the beginning of the algorithm, a minimum pivot threshold *pivmin* is computed. It represents the smallest divisor so that T_{ii+1}^2/pivmin does not overflow. Then, whenever $|d(i-1)| < \text{pivmin}$, the algorithm replaces $d(i-1)$ by pivmin. (The convention in LAPACK's DSTEBZ is actually to replace the pivot by negative pivmin.) This substitution limits the input range. To prevent overflows, the inputs must satisfy $|T_{ij}| \leq \Omega^{1/2}\eta^{1/4}$, where Ω is the arithmetic's overflow threshold and η is the smallest positive normalized number.

IEEE-754 arithmetic allows us to avoid the pivmin threshold [8]. With $d(i-1) = 0$, the next pivot becomes $-\infty$, and the computation continues normally afterwards with $d(i+1) = (T_{i+1i+1} - \sigma)$. In [8], the authors report a speed-up of 1.2 to 1.3 of Sturm counts using the IEEE-754 version over the version based on the pivmin threshold. The impact on bisection is reported to be a speed-up of 1.14 to 1.24. Also, the IEEE-754 version does *not* impose any range restrictions.

2.2. Sturm counts for a tridiagonal matrix in factored form LDL^T. Instead of applying Sylvester's Law of Inertia to a tridiagonal matrix T, it also can be applied to its LDL^T factorization. This idea leads to differential qds transformations that are introduced in the following. They are of great importance to the MRRR algorithm [23, 24, 25, 26, 9] that requires the reliable computation of Sturm counts from shifted LDL^T factorizations instead of the original tridiagonal matrix.

In order to compute Sturm counts, MRRR uses two principal algorithms, the *differential stationary qds* factorization $LDL^T - \sigma I = L_+ D_+ L_+^T$ and the *differential progressive qds* factorization $LDL^T - \sigma I = U_- D_- U_-^T$. These are variants of an algorithm originally proposed by Rutishauser [28, 29] and are stable in a mixed sense, see [26].

While a Sturm count on T does not look very different from one on LDL^T, there are subtle and important differences between the two approaches that are summarized in the following:

	tridiagonal T	Factored $T = LDL^T$
Sturm sequence algorithm	$T - \sigma I = LDL^T$	differential qds algorithms
Possible exceptional values	∞	∞, NaN
Is the final Sturm count valid?	Always	If NaN has not occurred

Algorithm 2.1 presents both the stationary (a) and progressive (b) algorithms. The variable *negcount* holds the Sturm count. Note the following:

- Both algorithms use the quantities $d(i)$ and $l(i)^2 d(i)$. If the Sturm count is repeated several times as in bisection, precomputing $lld(i) \equiv l(i)^2 d(i)$ is more efficient than repeatedly computing $l(i)^2 d(i)$. We use $lld(i)$ through the rest of the paper.
- For the purposes of the Sturm sequence, it is not necessary to store the pivots $d_+(i)$ and $d_-(i)$.
- The division in both algorithms is probably the most costly operation. We place the sign tests after the division to encourage optimizations that overlap the two operations.

Algorithm 2.1 The differential stationary and progressive qds algorithms compute $L_+ D_+ L_+^T(\text{stationary}) = U_- D_- U_-^T(\text{progressive}) = LDL^T - \sigma I$ and count the number of negative pivots. The diagonal elements of D are $d(1), \ldots, d(n)$ and the off-diagonal entries of L are $l(1), \ldots, l(n-1)$. The quantity *negcount* denotes the number of eigenvalues of LDL^T that are less than σ.

negcount $= 0$
$t = -\sigma$
for $i = 1 : n-1$ **do**
  $d_+(i) = d(i) + t$
  $t = \Big(t/d_+(i)\Big) * lld(i) - \sigma$
  Increase negcount if $d_+(i) < 0$
end for
$d_+(n) = d(n) + t$
Increase negcount if $d_+(n) < 0$

(a) Stationary qds

negcount $= 0$
$p = d(n) - \sigma$
for $i = n-1 : 1$ **do**
  $d_-(i+1) = lld(i) + p$
  $p = \Big(p/d_-(i+1)\Big) * d(i) - \sigma$
  Increase negcount if $d_-(i+1) < 0$
end for
$d_-(1) = p$
Increase negcount if $d_-(1) < 0$

(b) Progressive qds

Sturm counts on the factored matrix LDL^T via differential stationary or progressive qds transforms can produce a NaN during the computation. In the case of differential stationary qds, if $d_+(i-1) = 0$, then $|d_+(i)| = \infty$ and $d_+(i+1) = \text{NaN}$, see also Table 2.1(a). For the differential progressive qds, if $d_-(i+1) = 0$, then

$|d_-(i)| = \infty$ and $d_-(i-1) = \text{NaN}$, see also Table 2.1(b).

	$i-1$	i	$i+1$
t	$-d(i-1)$	$\pm\infty$	NaN
d_+	0	$\pm\infty$	NaN

(a) Stationary qds

	$i-1$	i	$i+1$
d_-	0	$\pm\infty$	NaN
p	$\pm\infty$	NaN	NaN

(b) Progressive qds

Table 2.1: Variable values after a zero pivot is encountered in step $i-1$ of the stationary and the progressive variant of Algorithm 2.1.

If a NaN occurs, that NaN propagates through all subsequent computations. The resulting Sturm count is *incorrect* in this case; the computation needs to be repeated with a NaN-free variant of the respective algorithm. This is in contrast to a Sturm sequences using T that is robust even when ∞ occurs.

2.3. NaN-free variants of differential qds transforms. There are different possible approaches to avoid invalid operations in the differential stationary and progressive qds algorithms. In this section, we discuss three such variants of Algorithm 2.1 that are based on appropriate substitutions. A summary of the possible exceptional values and range restrictions of the various alternative formulations is given in Table 2.2. These are explained in the later sections.

Algorithm	Exceptional values	Range restrictions
2.2: pivmin	**none**	$\lvert T_{ij}\rvert \leq \Omega^{1/2}\eta^{1/4}$
2.3: $\infty/\infty \Rightarrow 1$	$\pm\infty$	**none**
2.4: $\pm\infty \Rightarrow \pm\Omega$	$\pm\infty$	$\lvert d(i)\rvert < \Omega\,\varepsilon$

Table 2.2: Range restrictions and possible exceptional values for the alternative formulations of the differential qds algorithms. The quantity Ω is the overflow threshold, η is the smallest positive, normalized number, and ε denotes the smallest positive number such that $1+\varepsilon$ does not evaluate to 1.

2.3.1. Substitution of pivmin for tiny pivots. The first approach mimics the tridiagonal Sturm count procedure presented in Section 2.1. Algorithm 2.2 replaces tiny and zero pivots with a minimum threshold *pivmin* to prevent division by a zero pivot. It entirely avoids exceptional values. The choice of *pivmin* is justified by [19] and used in LAPACK. In our tests, we use pivmin $= \eta \cdot \max_{i=1\ldots n-1} T^2_{i,i+1}$. To avoid overflows, the initial data must satisfy $|T_{ij}| \leq \Omega^{1/2}\eta^{1/4}$, where Ω is the overflow threshold.

Algorithm 2.2 Substituting pivmin for very small pivots.

```
negcount = 0
t = −σ
for i = 1 : n − 1 do
   d+(i) = d(i) + t
   if |d+(i)| < pivmin then
      d+(i) = −pivmin
   end if
   t = (t/d+(i)) ∗ lld(i) − σ
   Increase negcount if d+(i) < 0
end for
d+(n) = d(n) + t
Increase negcount if d+(n) < 0
```

(a) Stationary qds

```
negcount = 0
p = d(n) − σ
for i = n − 1 : 1 do
   d−(i + 1) = lld(i) + p
   if |d−(i + 1)| < pivmin then
      d−(i + 1) = −pivmin
   end if
   p = (p/d−(i + 1)) ∗ d(i) − σ
   Increase negcount if d−(i+1) < 0
end for
d−(1) = p
Increase negcount if d−(1) < 0
```

(b) Progressive qds

2.3.2. Substitution of ∞/∞ by 1. The second approach mimics the tridiagonal Sturm count by allowing ∞ but preventing the subsequent NaN. For the differential stationary qds algorithm (Algorithm 2.1(a)), substituting of an alternate value for ∞/∞ can avoid producing a NaN. Using L'Hospital's limit formula, we find $\lim_{t\to\infty} t/d_+(i) = \lim_{t\to\infty} t/(d(i)+t) = 1$. The same argument holds for the quotient $p/d_-(i+1)$ in the differential progressive qds algorithm (Algorithm 2.1(b)). The qds variants in Algorithm 2.3 explicitly substitute 1 for ∞/∞. Note that the infinite t (or p) from a tiny or zero pivot produces a pivot d_+ (or d_-) with the same sign, so the substitution is always 1. This variant does not restrict the input range but does produce $\pm\infty$ values.

Algorithm 2.3 Substituting 1 for ∞/∞.

```
negcount = 0
t = −σ
for i = 1 : n − 1 do
   d+(i) = d(i) + t
   if |t| = ∞ and |d+(i)| = ∞ then
      q = 1
   else
      q = (t/d+(i))
   end if
   t = q ∗ lld(i) − σ
   Increase negcount if d+(i) < 0
end for
d+(n) = d(n) + t
Increase negcount if d+(n) < 0
```

(a) Stationary qds

```
negcount = 0
p = d(n) − σ
for i = n − 1 : 1 do
   d−(i + 1) = lld(i) + p
   if |p| = ∞ and |d−(i + 1)| = ∞
   then
      q = 1
   else
      q = (p/d−(i + 1))
   end if
   p = q ∗ d(i) − σ
   Increase negcount if d−(i+1) < 0
end for
d−(1) = p
Increase negcount if d−(1) < 0
```

(b) Progressive qds

2.3.3. Substitution of Overflow and Divide-by-Zero by $\pm\Omega$. Following a suggestion of W. Kahan [19, 20], we present an alternative version which emulates saturation arithmetic, an arithmetic where all overflows produce largest representable number rather than ∞. Algorithm 2.4 replaces an infinite t or p with the correctly signed largest representable number $\pm\Omega$. As long as the subsequent addition in the next iteration does not overflow, division with the new pivot evaluates to 1 as in Algorithm 2.3. In true saturation arithmetic, all overflows would produce a correctly signed Ω as a result. Because we only emulate saturation arithmetic, the subsequent addition overflows and yields an incorrect result if the initial $d(i) \geq \Omega\epsilon$. Thus we must restrict the input range slightly. This variant also produces infinities as intermediate values. The function application $\text{sign}(t)$ in Algorithm 2.4 returns 1 if $t = +\infty$ and -1 if $t = -\infty$.

Algorithm 2.4 Substituting the largest magnitude, finite number Ω for infinite quotients (correctly signed).

(a) Stationary qds

```
negcount = 0
t = −σ
for i = 1 : n − 1 do
   d₊(i) = d(i) + t
   t = (t/d₊(i)) ∗ lld(i) − σ
   if |t| = ∞ then
      t = sign(t) ∗ Ω
   end if
   Increase negcount if d₊(i) < 0
end for
d₊(n) = d(n) + t
Increase negcount if d₊(n) < 0
```

(b) Progressive qds

```
negcount = 0
p = d(n) − σ
for i = n − 1 : 1 do
   d₋(i + 1) = lld(i) + p
   p = (p/d₋(i + 1)) ∗ d(i) − σ
   if |p| = ∞ then
      p = sign(p) ∗ Ω
   end if
   Increase negcount if d₋(i + 1) < 0
end for
d₋(1) = p
Increase negcount if d₋(1) < 0
```

3. A benchmark of the various differential qds algorithms. We devise two benchmarks to compare the various algorithms. For a given tridiagonal matrix V_n in factored form LDL^T, both benchmarks compute differential qds factorizations of $LDL^T - \sigma I$. In the first case, we choose σ such that no zero pivot (and hence no IEEE exception) occurs. In the second case, we choose σ to force an exception. These benchmarks allow us to compare the performance of the algorithms in the unexceptional case, and to quantify the penalties when an IEEE exception occurs. For given dimension n, the matrix V_n is a tridiagonal with all off-diagonal entries equal one; its diagonal consists of the vector $1 : n$.

3.1. Benchmark details. The occurrence of NaN always follows a zero (or tiny) pivot in $LDL^T - \sigma I$, see Table 2.1(a and b). Furthermore, a pivot can only be zero when the shift is chosen as an eigenvalue of a leading submatrix in the stationary case, and as an eigenvalue of a trailing submatrix in the progressive case.

For the first benchmark, the shift σ is chosen well outside the Gershgorin bounds, this guarantees that no exceptional operation occurs.

The second benchmark must induce an invalid operation. For the differential stationary qds algorithms, we choose as the shift the first diagonal element $d(1)$, and a NaN occurs in step 2. For the differential progressive qds algorithm, setting

$lld(n-1) = -(d(n)-\sigma)$ forces the pivot $d_-(n)$ to zero for any shift σ. However, in order to prevent the matrix from becoming reducible, we must choose $\sigma \neq d(n)$. A reducible matrix would produce NaNs even in the alternative formulations of the progressive algorithm. The choice $\sigma = d(n)/2.0$, and $lld(n-1) = -d(n)/2.0$ produces a NaN regardless of any compiler optimizations or rounding settings.

We run each benchmark in double precision for matrices of dimensions from 500 to 6 000 by steps of 500. Individual Sturm counts compute very quickly, so we time a loop of 50 000 Sturm counts on the same data. Table 3.1 summarizes the architectures, compilers, and timers used for our experiments.

Architecture	Clock (MHz)	OS	Compiler	Timer
Athlon-XP	1200	GNU/Linux	gfortran (Debian 4.0.1-6) -O3 -fno-trapping-math -march=athlon-xp	`gettimeofday`
Itanium 2	1300	Linux	Intel ifort 9.0 -O2	`gettimeofday`
Pentium 4 Xeon, SSE	3000	Linux	Intel ifort 9.0 -O3 -xP	`gettimeofday`
Pentium 4 Xeon, x87	3000	Linux	Intel ifort 9.0 -O3 -xP -mp	`gettimeofday`
MIPS R12000	600	IRIX	SGI MIPS pro 7.3.1.3m -O2	`ETIME`
Power3	375	AIX	IBM xlf90 v8r11 -O3	PAPI
UltraSparc 2i	650	Solaris	SUN f90 forte 7.0 -O4	PAPI
Cray X1	800	UNICOS	Cray ftn 5.4.0.4 -O2	`cpu_time`
Opteron	2200	Linux	Pathscale pathf90 2.1 -O3	`cpu_time`

Table 3.1: Platforms and timers used for testing.

All tested platforms support the default IEEE-754 execution mode, producing and propagating ∞ and NaN without user-visible traps. We tested both the Pentium 4 Xeon's 80-bit x87 floating-point unit and its 64-bit SSE3 floating-point unit. We tested only the SSE2 unit on the Opteron. SSE2 and SSE3 floating-point units can perform vector operations where each operand is a pair of double-precision values. The Intel and Pathscale compilers did not vectorize the Sturm counting loops and used only the first 64-bit entry in each vector register. The Athlon platform tested supports only SSE1, so our tests use its x87 floating-point unit.

3.2. Benchmark results. Both benchmarks time the same Sturm count routines. Algorithms 2.2, 2.3, and 2.4 each are applied to $T = LDL^T$ for the two shifts above. To test the paradigm of computing quickly and fixing failures, we time three additional versions. Each runs the basic qds loop in Algorithm 2.1, tests the final pivot for NaN, and re-computes with runs of Algorithms 2.2, 2.3, or 2.4 if the pivot is NaN.

Figures 3.1, 3.2, and 3.3 present the timings for every algorithm and platform.

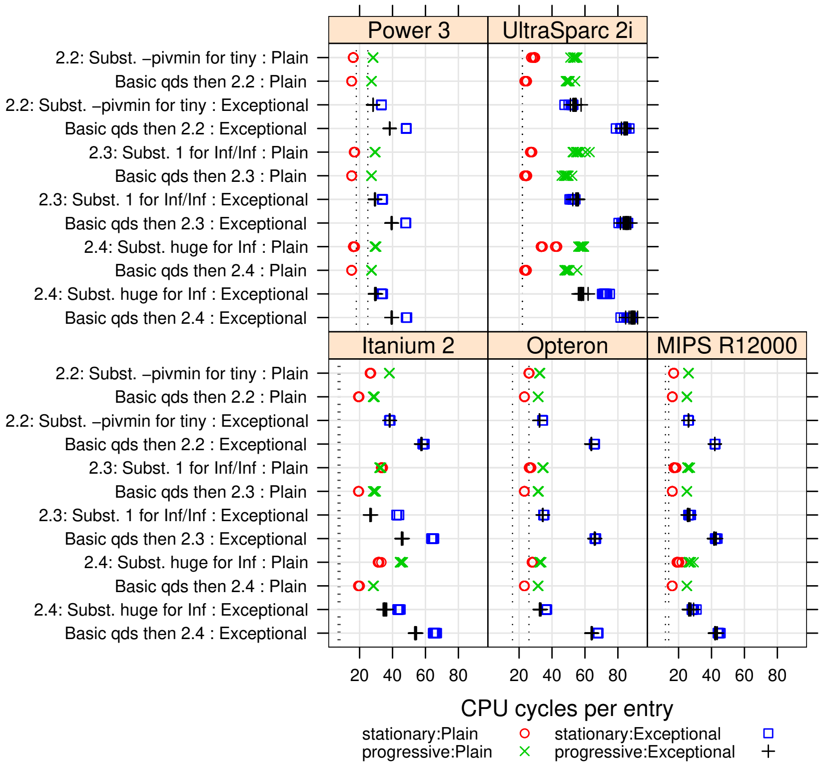

Fig. 3.1: CPU cycles per entry for each qds algorithm. Vertical dashed lines show the range of documented division latencies, if known.

The times are normalized into CPU cycles per entry examined by

$$\text{CPU cycles per entry} = \frac{\text{total time}}{50\,000 \text{ iterations} \cdot n \text{ entries}} \cdot \text{clock speed in Hz} \tag{3.1}$$

given the CPU clock speeds in Table 3.1. Note that the figures have different horizontal scales, and Figure 3.3 provides two views of the Pentium 4 Xeon's performance when using the x87 floating-point unit. The markers have different shapes to distinguish both between progressive and stationary loops and also between plain and exceptional execution. The dashed vertical lines denote the range of division latencies given in platform documentation and tuning guides [1, 17, 18, 30, 32, 31, 2]. No such information is available for the Cray X1.

3.3. Examining the benchmark results. On an unexceptional run, a basic qds Sturm count achieves a speed-up of 1.12 over the careful algorithms of Section 2, where speed-up is measured by the time of the careful algorithm divided by the time of the basic qds algorithm on the same inputs. When exceptions occur, the median slow-down of the combined basic/careful qds loops over the careful loop alone is 1.63. We measure slow-down as the time of the combined loop divided by the time of the careful loop on the same inputs. The only slow-downs over 2 occur for the x87 unit

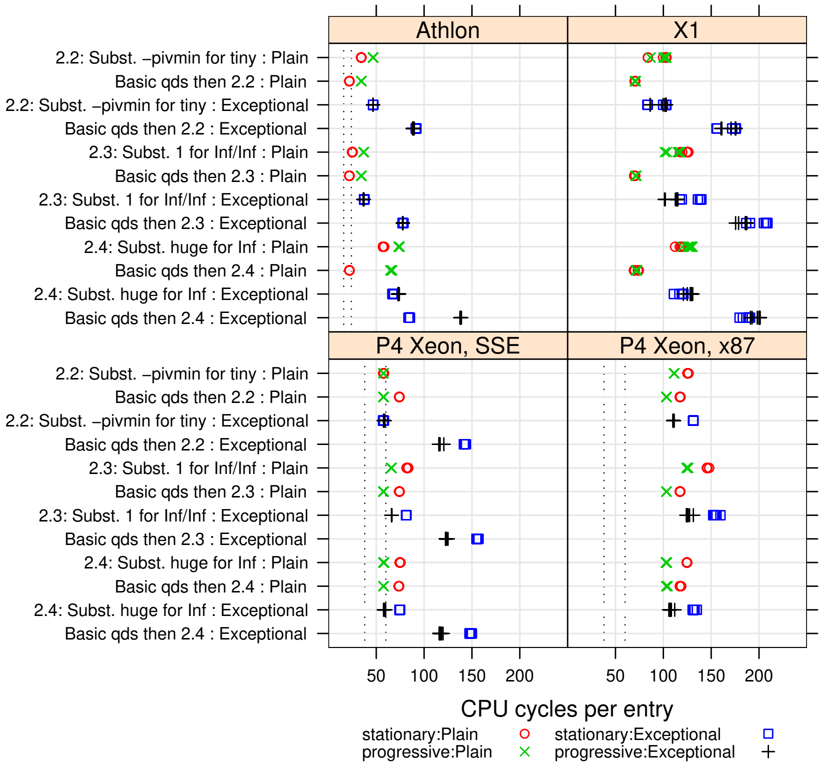

Fig. 3.2: CPU cycles per entry for each qds algorithm for platforms needing a wider scale than Figure 3.1. Xeon values with the x87 unit are clipped to fit. Vertical dashed lines show the range of documented division latencies, if known.

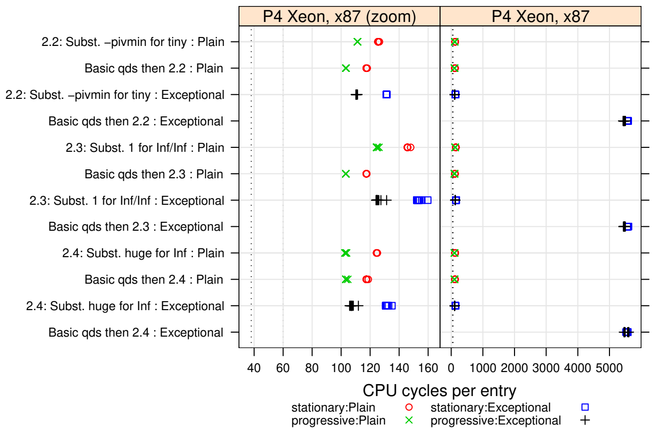

Fig. 3.3: CPU cycles per entry for each qds algorithm on the Xeon with the x87 unit. Vertical dashed lines show the range of documented division latencies, if known.

on the Pentium 4 Xeon, which drastically penalizes exceptional values and suffers slow-downs from 18.93 to 66.53. No one careful algorithm's performance dominates the others, but we recommend implementors use Algorithm 2.3 ($\infty/\infty \Rightarrow 1$) because it imposes no special range restrictions on the input data.

Unexceptional execution. Figure 3.4 compares the basic qds loops' times to the times of their careful counterparts when no exceptions occur. As expected, the basic qds loop runs faster. The speed-up of the basic qds loop over the corresponding careful algorithms across all platforms is given in Table 3.2.

Variant	Min	1^{st} Quartile	Median	Mean	3^{rd} Quartile	Max
All variants	0.775	1.063	1.118	1.235	1.355	2.638
2.2: pivmin	0.775	1.039	1.078	1.166	1.355	1.565
2.3: $\infty/\infty \Rightarrow 1$	1.016	1.091	1.129	1.218	1.206	1.805
2.4: $\pm\infty \Rightarrow \pm\Omega$	0.984	1.055	1.133	1.321	1.580	2.638

Table 3.2: Ratio of careful loops' times to corresponding basic loops' times, or the speed-up of the basic loop over the careful variants, for unexceptional inputs across all platforms and both progressive and stationary qds.

Note that the stationary loops sometimes run more slowly than the progressive loops; some platforms appear to prefer sequential forward memory access over sequential reverse access. And the x87 unit on the Pentium 4 Xeon prefers Algorithm 2.2 (pivmin) over Algorithm 2.1's simple qds loop.

Exceptional execution. Figure 3.5 shows the slow-downs from attempting a basic qds Sturm count, finding an exception, and re-counting with a careful loop. The x87 unit on the Xeon processors shows an excessive slow-down, detailed in Figure 3.6. This is evidence that the Xeon implements IEEE-754 special values in microcode for its x87 unit. In other words, every time the Xeon's x87 unit encounters an ∞ or NaN *value*, it flushes its internal pipelines and jumps to a special handler program inside the processor. In contrast, the Athlon's x87 unit processes these special values at full speed.

Variant	Min	1^{st} Quartile	Median	Mean	3^{rd} Quartile	Max
All variants	1.088	1.526	1.634	6.417	1.963	52.77
2.2: pivmin	1.356	1.566	1.708	6.660	1.972	49.60
2.3: $\infty/\infty \Rightarrow 1$	1.343	1.544	1.712	5.956	1.917	44.13
2.4: $\pm\infty \Rightarrow \pm\Omega$	1.088	1.493	1.569	6.636	1.960	52.77

Table 3.3: Ratio of the failed basic loop plus the careful loop's times to the corresponding careful loops' times, or the slow-down of the attempted basic loop over the careful variants, for exceptional inputs across all platforms and both progressive and stationary qds.

Assuming exceptions occur rarely in practice, most slow-downs in Figure 3.5 and Table 3.3 are acceptable. The median is 1.63, less than the factor of two that we expected. Section 5 details how breaking the loops into shorter blocks can limit impact of the worst slow-downs.

Fig. 3.4: As long as no exception occurs, the basic qds loop almost always runs faster than the careful loops that test for exceptional operations. A point at 1.2 means the careful loop's running time was 1.2 times the basic loop's time; the basic loop sees a speed-up of 20%.

Choosing a backup qds loop. To help choose one among the careful qds loops to support the basic loop, Figure 3.7 and Table 3.4 compare times basic loop followed by the two algorithms with range restrictions, 2.2 and 2.4, to the time of the same loop followed by the algorithm without range restrictions, 2.3. No algorithm completely dominates the performance of the others. We suggest that users adopt Algorithm 2.3 because it imposes no range restrictions on the data.

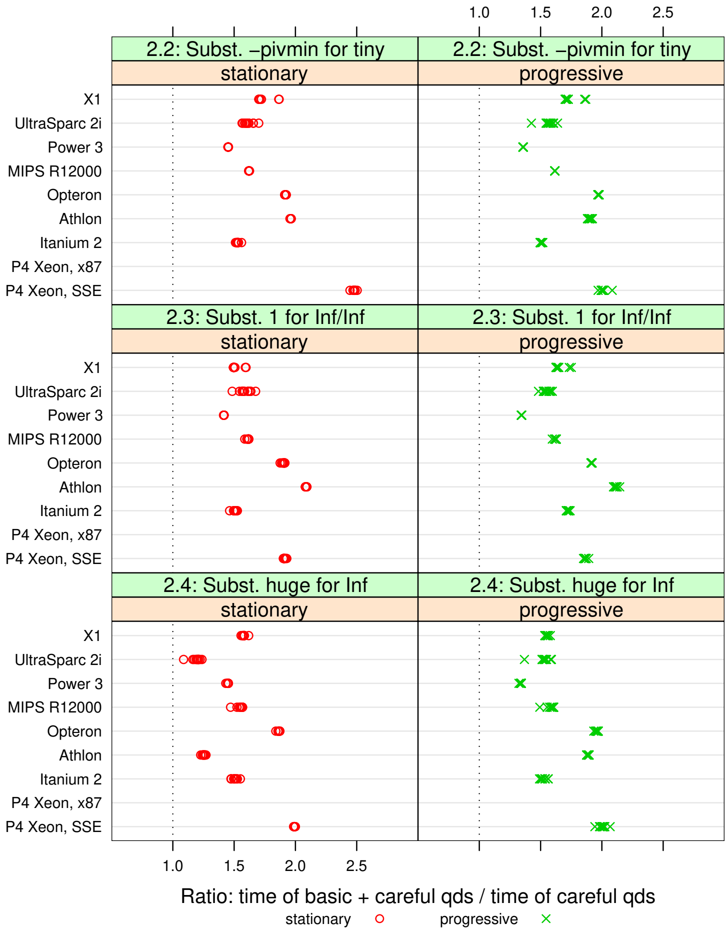

Fig. 3.5: When an exception occurs, however, many platforms suffer a penalty for carrying the NaN through $O(n)$ operations. Here, a point at 1.5 means that the basic loop followed by the careful loop ran 1.5 times as long as the careful loop alone. The values for the x87 Xeon runs are clipped.

Variant	Min	1st Quartile	Median	Mean	3rd Quartile	Max
2.2: pivmin	0.820	0.945	0.992	1.003	1.009	1.273
2.4: $\pm\infty \Rightarrow \pm\Omega$	0.901	0.995	1.014	1.061	1.048	1.788

Table 3.4: Ratio of the listed careful loops' times to the corresponding times for variant B ($\infty/\infty \Rightarrow 1$) for exceptional inputs across all platforms and both progressive and stationary qds. To evaluate performance as a "backup" method, the times compared are those of the basic qds loop followed by the careful loop. We attribute the wide spread to wide architectural and compiler differences.

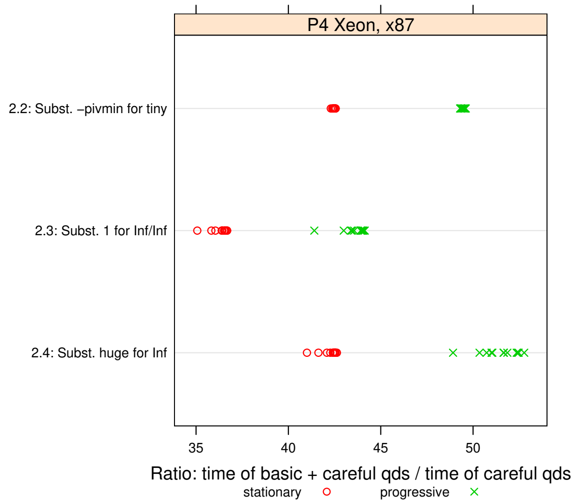

Fig. 3.6: The x87 unit on Xeons suffers an extreme penalty for computing with NaNs, from over 18× to almost 70× slow-down.

4. Impact on bisection. Does performance of the Sturm count impact eigensolvers? We answer this question in part by using Section 2.3's algorithms in bisection. We apply bisection to the tridiagonal matrices from matrix collections and applications described in Section 4.1. The bisection routine tested is a modification of the auxiliary routine DLARRB from LAPACK. We test each version of our Sturm counts individually by computing the eigenvalues to within a relative error of 4ε. Overall, the basic qds loop provides a median speed-up of only 1.08. Particular platforms like the Cray X1 see significant improvements.

4.1. Test Matrices. The symmetric tridiagonal matrices used in our experiments come from a wide range of real applications, as follows:

(A) Applications in chemistry. The matrices are related to the modeling of biphenyl using Moller-Plesset theory, density functional theory applied to determine the bulk properties of $SiOSi_6$, and the self-consistent Hartree-Fock method for solving a non-linear Schroedinger problem. The matrices were obtained with the NWChem [13, 12] computational chemistry package and were provided by George Fann.

(B) Matrices from the Harwell-Boeing Collection [10, 11]. The matrices are related to the modeling of power system networks, a finite-difference model for the shallow wave equations for the Atlantic and Indian Oceans, a finite element approximation to the biharmonic operator on a beam and on a plate, and structural engineering finite element analysis.

(C) Matrices from the University of Florida Sparse Matrix Collection [5]. The matrices are related to finite-element problems.

The matrices from (B) and (C) were tridiagonalized by means of LAPACK's DSYTRD or a simple Matlab implementation of the Lanczos algorithm, with a starting vector filled with ones and no reorthogonalization, so as to provoke the appearance of very close eigenvalues in the resulting tridiagonal. In this case, Lanczos was run for

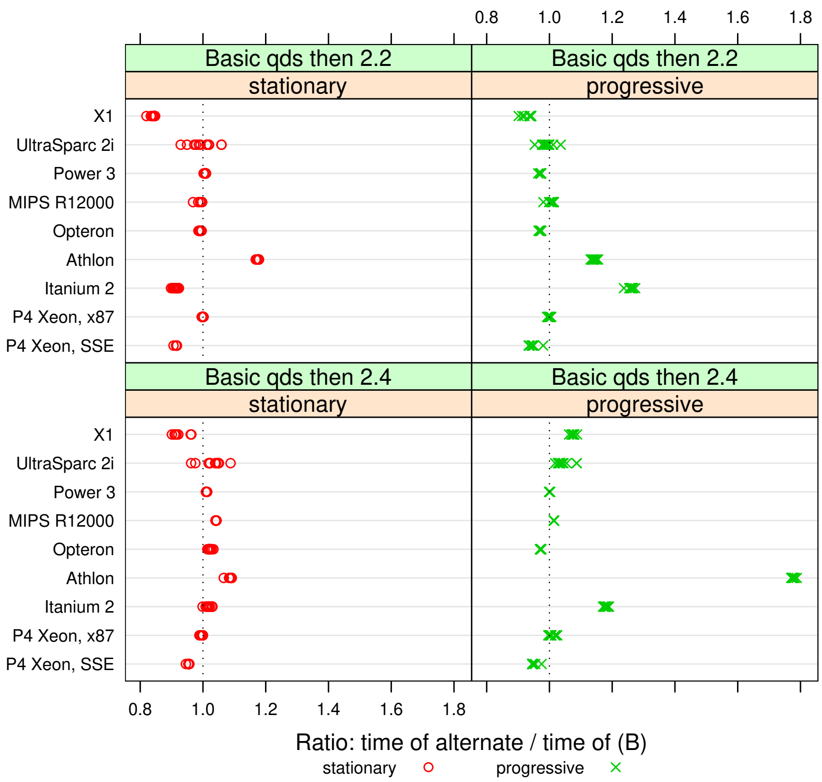

Fig. 3.7: Comparing Algorithms 2.2 and 2.4, those with input range restrictions, to Algorithm 2.3 shows no all-around performance winner.

$N \cdot k$ steps, where N is the the dimension of the problem and k an integer at most 4. Table 4.1 gives more information about the matrices used in the tests.

Matrix	Application	Source	# of matrices	Dimensions
fann	Chemistry	NWChem	3	966–2053
bus	Power system networks	HB	4	494–1138
plat1919	Shallow wave equations	HB	1	1919
nos	Biharmonic operator	HB	2	957–960
bcsstk	Structural engineering	HB	18	1074–8012
alemdar, nasa, sts	Finite-element problems	UF	17	1824–6438

Table 4.1: Matrices used in the tests were drawn from NWChem, the Harwell-Boeing collection (HB), and the University of Florida collection (UF).

4.2. Bisection performance results. Figures 4.1 and 4.2 show the CPU cycles per entry for each qds algorithm and platform. Because bisection does not run a fixed number of Sturm counts, calculating the cycles per entry requires replacing the

50 000 in Equation (3.1) with the number of calls to the qds routine in each individual execution. Section 4.1's test matrices generated no exceptions. Figure 4.3 provides the speed-ups for each algorithm and platform, and Table 4.2 summarizes those speed-ups across all platforms.

Variant	Min	1^{st} Quartile	Median	Mean	3^{rd} Quartile	Max
All variants	0.536	1.029	1.084	1.158	1.322	2.006
2.2: pivmin	0.734	1.024	1.038	1.112	1.350	1.543
2.3: $\infty/\infty \Rightarrow 1$	0.557	1.036	1.077	1.140	1.150	1.984
2.4: $\pm\infty \Rightarrow \pm\Omega$	0.536	1.034	1.119	1.223	1.432	2.006

Table 4.2: Ratio of bisection's time using careful qds loops to corresponding bisection time with basic loops, or the bisection speed-up of the basic loop over the careful variants, for unexceptional inputs across all platforms and both progressive and stationary qds.

The median improvement of the basic qds loop over the careful algorithms is a modest 1.08, with a few notable slow-downs using the P4 Xeon's x87 unit. However, some platforms see substantial speed-ups. The Itanium 2 and Cray X1 both see speed-ups of at least 1.2. The Cray X1 benefits the most, achieving a maximum speed-up of 1.98 and a median speed-up of 1.66.

5. Blocking. For larger matrix sizes, the potential slow-downs shown in Section 3.3 become unacceptable. Blocking the loops limits the slow-down for most cases without surrendering the basic loop's performance.

The idea is simple: instead of performing the whole computation with the fast algorithm for the full loop, the computation is split into blocks $1 : n_1$, $n_1 + 1 : n_2$, and so forth. After the computation for the current block has finished, its correctness is assessed immediately. Thus, in the case of an invalid result, only the computation of the current block is repeated.

In our context of computing the Sturm count from a differential qds factorization, blocking results in an outer loop over the blocks, and an inner loop that computes the Sturm count *increment*, the number of negative pivots in the factorization of the current block. The increment is added to the count of the factorization encounters no exceptions. Otherwise, the increment is re-computed with a careful algorithm.

The block size is a tuning parameter. Its setting has to address the trade-off between the cost for assessing the correctness of the intermediate solution, in our case the NaN test, and the potential overhead for redoing the computation. The optimal block size may depend not only on the computation costs but also on caching effects.

The benchmark test from Section 3 motivates the blocking approach. The basic stationary differential qds algorithm produces a NaN in step 2 when the shift is chosen to be $d(1)$. Once that exception is handled, no further exceptions occur. Thus, the earlier the NaN is detected, the fewer unnecessary operations are performed. At the end of the first block, the NaN is detected and only the first block's contribution is recomputed. The remainder of the blocks will proceed at full speed.

Blocking has no benefits in those very rare cases in which the computation has to be redone for *every* block. One such example is an LDL^T factorization of a symmetric diagonally dominant tridiagonal Toeplitz matrix (e.g. the 1-2-1 matrix). We leave it to the reader to verify that for any shift in the stationary dqds algorithm for each

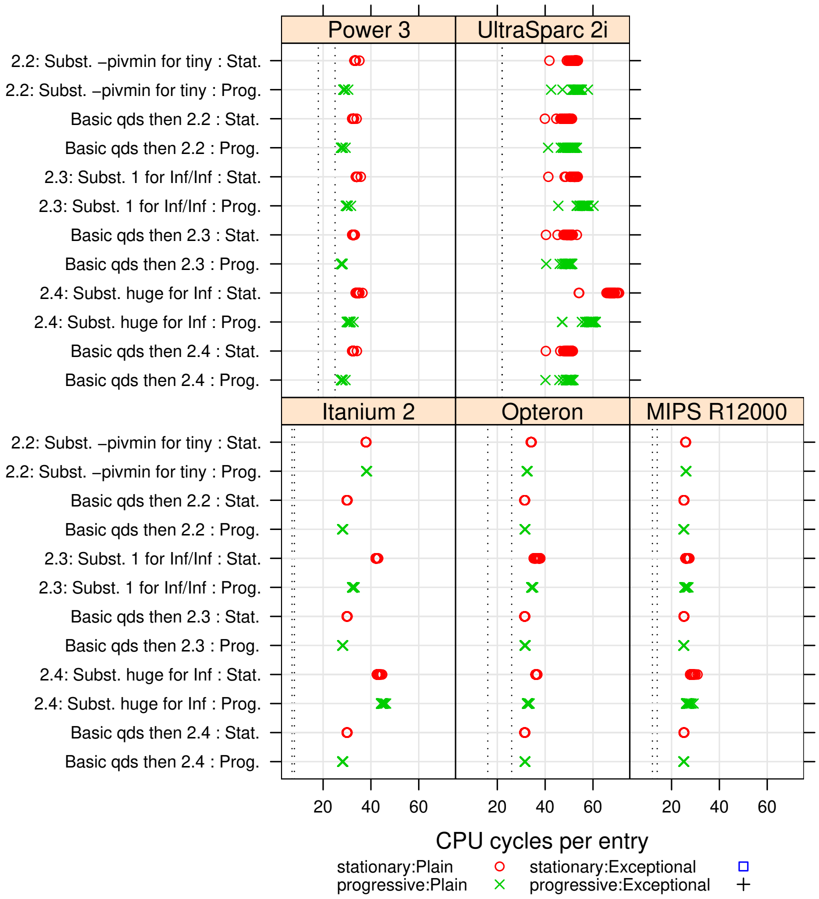

Fig. 4.1: CPU cycles per entry for each qds algorithm during bisection. Vertical dashed lines show the range of documented division latencies, if known.

block size greater than two, the first pivot d_+ in each block is zero. Thus a NaN occurs in every block.

Conclusion. Evaluating the Sturm count with a qds factorization risks failure when a tiny or zero pivot causes an overflow, but avoiding those rare failures exacts a cost in performance. Following Demmel and Li [8]'s example, we achieve better performance by attempting a fast but potentially erroneous qds factorization, testing for failure afterwards, and recomputing the factorization more carefully only when necessary. As shown in Section 3.3, that strategy typically yields benchmark speed-ups over careful factorizations of around 1.12 across many platforms. However, a remarkable slowdown of up to 66× can occur for the x87 unit on the Pentium 4 Xeon on exceptional cases. For bisection on the matrices in Section 4.1, we find the speed-ups vary greatly by platform and algorithm. While the median is a modest 1.08, some platforms benefit greatly. For example, bisection on the Cray X1 improves by factors of 1.2 to up to 2.

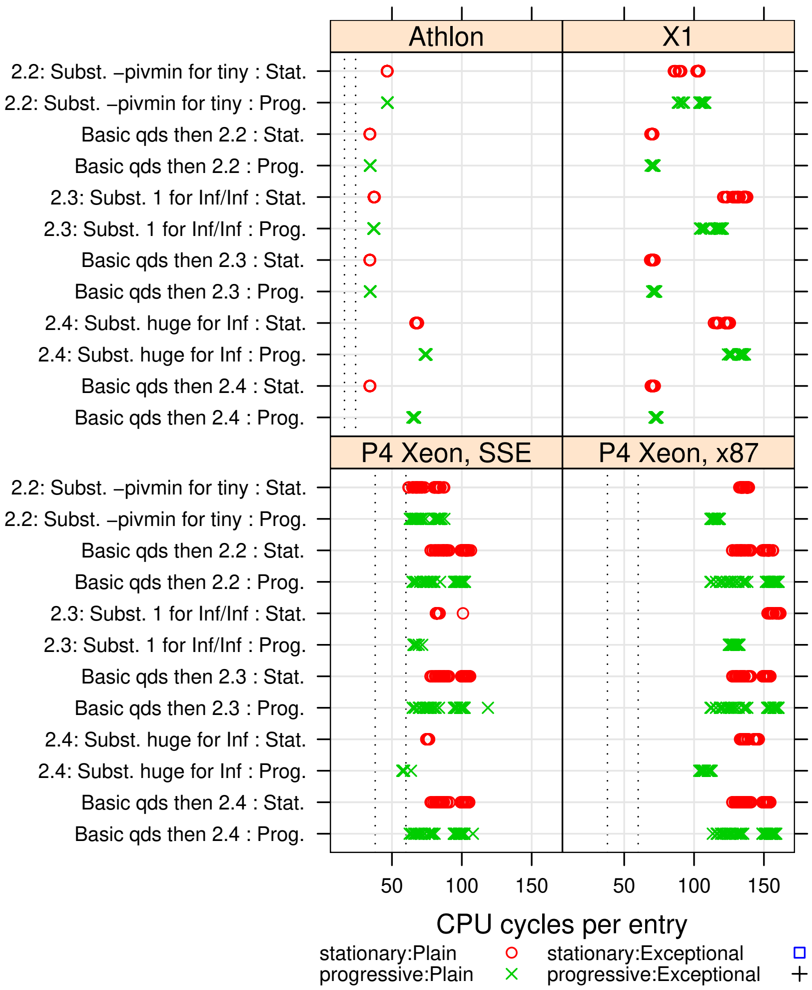

Fig. 4.2: CPU cycles per entry for each qds algorithm during bisection for platforms needing a wider scale than Figure 4.1. Xeon values with the x87 unit are clipped to fit. Vertical dashed lines show the range of documented division latencies, if known.

This paper has focused on the performance benefits of IEEE-754 features in the context of bisection. However, there are other interesting applications. We mention two related examples which should find similar benefits.

First, consider the dqds algorithm for computing the singular values of a bidiagonal matrix [14, 15, 22, 27]. There an invalid operation signals that stability has been lost. The shift to accelerate the convergence has been chosen too aggressively and needs to be replaced by a more prudent shift.

Second, consider the MRRR algorithm's method of computing eigenvectors [23, 24, 25, 26, 9]. An invalid operation during the twisted factorization indicates that an eigenvector entry is tiny or zero. In this case, the fast product formula used to

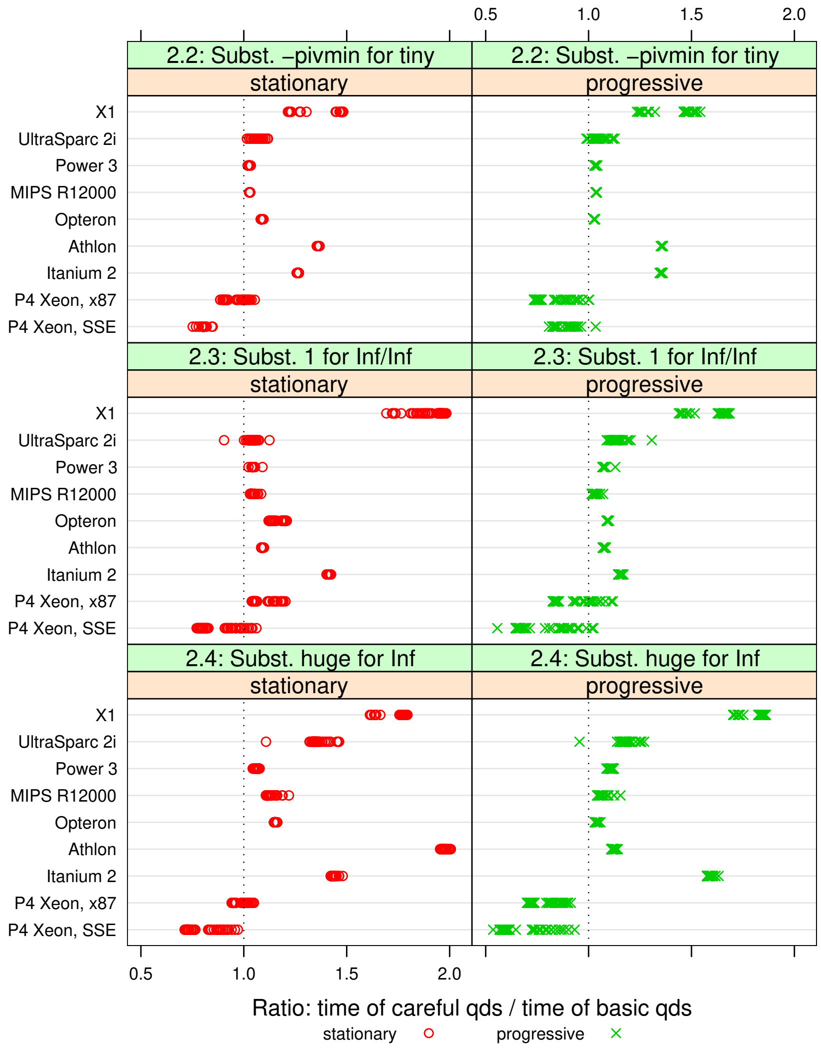

Fig. 4.3: Similar improvements occur for bisection. A point at 1.2 means that bisection's running time using the careful loop is 1.2 times as long as bisection's time using the basic loop.

compute the eigenvector yields an incorrect result and must be replaced by a more careful formula.

Acknowledgments. We would like to thank Prof. Kahan for Algorithm 2.4 and many comments. We acknowledge the contribution from Intel Corporation, Hewlett-Packard Corporation, IBM Corporation, and the National Science Foundation grant EIA-0303575 for providing hardware and software to the CITRIS Cluster used in producing these research results. We also thank Cray Inc. for giving us access to the

X1, and NERSC for our use of the IBM SP3.

Appendix A. Important implementation details.

This section describes briefly important implementation details for our benchmark of the qds variants.

We describe the effort needed to test for NaN to ensure a valid Sturm count from the unsafe loops in Algorithm 2.1. Only recently have language systems provided portable tests for NaN. Since we cannot yet rely on those systems' availability, we provide our own test described in Section A.1.

Also, Algorithms 2.3 and 2.4 must test for infinite values. Thankfully such tests can be implemented portably, see Section A.2

Section A.3 discusses possible tests for the sign of a pivot.

And finally, Section A.4 discusses the potential benefits and unfortunate drawbacks of using system trap handlers to treat exceptions when they occur.

A.1. Testing for NaNs. While NaN is the only IEEE-754 value for which X.EQ.X is false, many compilers in higher optimization levels replace this logical expression by a *constant* true value. In such circumstances, we have no in-line, reliable test for the presence of a NaN at run time.

For this reason, our experiments use a function of the form

```
LOGICAL FUNCTION DISNAN( DIN1, DIN2 )
DOUBLE PRECISION DIN1, DIN2
DISNAN = (DIN1.NE.DIN2)
END FUNCTION
```

that is compiled separately. Baring interprocedural analysis, the compiler must assume DIN1 and DIN2 are not related. DISNAN(X,X) thus returns true if and only if X is NaN.

Future languages will provide better support for IEEE-754 facilities. For example, the Fortran 2003 standard provides a NaN test with the IEEE_IS_NAN() function in the IEEE_ARITHMETIC module.

There is a tradeoff in *when* to test for NaN. We could detect a NaN early by investigating the pivot in every step of the Sturm count, or we could use NaN propagation and wait and test only the final pivot. Including a test in every iteration penalizes unexceptional instances, while testing only after the loop requires loading all the data twice. Between these two extremes lies a range of blocking approaches, see Section 5.

A.2. Testing for ∞. The substitutions in Algorithms 2.3 and 2.4 require a test for the presence of ∞. This can be done portably by the following code:

```
OMEGA = DLAMCH('Overflow')
ABS(X).GT.OMEGA
```

Here OMEGA (also Ω in the text) represents the overflow threshold. LAPACK provides Ω through _LAMCH('O'). In Fortran 90, the enquiry function HUGE(X) returns the overflow threshold for the precision of its argument X. And Fortran 2003 provides IEEE_IS_FINITE() in the IEEE_ARITHMETIC module to test for non-finite IEEE-754 values directly.

A.3. Testing the sign of a pivot. The straightforward way to count negative pivots in Fortran is to compare the pivot to zero, as in the following:

```
IF( PIVOT.LT.ZERO ) NEGCOUNT = NEGCOUNT + 1
```

The Cray compiler recognizes the sign test inside the inner loop and performs an interesting optimization to partially vectorize it. The pivots from a portion of the loop are copied into intermediate vector. Then the entire vector is tested and converted into mask bits, and those mask bits are counted to provide a NEGCOUNT increment.

Another possible implementation copies the sign bit of PIVOT to an integer as 0 or 1, then adds that integer to NEGCOUNT. We lack a clear, direct way of expressing that implementation in Fortran.

A.4. Handling exceptional cases through traps. Algorithms 2.3 and 2.4 both explicitly test for exceptional operations. However, many platforms support implicit tests through *trapping*. A trap in this context is an implicit transfer of control from the exceptional operation to a *trap handler*. The handler examines the exceptional operation and its operands then returns a new result for the exceptional operation.

Trapping allows algorithms to run without explicit branches to test for exceptional cases; the tests are performed in hardware during floating-point operations. However, traps are only an optional feature in IEEE-754 [4]. While being widely implemented, each operating system and processor combination uses a different programming interface. This makes trap-handling non-portable. Furthermore, invoking a floating-point trap usually incurs expensive bookkeeping in both the processor and the operating system.

We experimented with trap handlers on three platforms with reasonable but very different interfaces: Solaris on an UltraSparc 2i, AIX on a Power3, and Linux on a Pentium 3. A single exceptional operation on any of these platforms increased the running time by a factor of at least a thousand.

REFERENCES

[1] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. 22007 revision K.

[2] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Athlon 64 and AMD Opteron*, November 2004. 25112 Rev. 3.05.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide.* SIAM, Philadelphia, 3. edition, 1999.

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

[5] T. A. Davis. University of Florida sparse matrix collection. `http://www.cise.ufl.edu/research/sparse/matrices/`, NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997.

[6] J. W. Demmel, I. S. Dhillon, and H. Ren. On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *Electronic Trans. Num. Anal.*, 3:116–140, 1995. (LAPACK working note #70).

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

[8] J. W. Demmel and X. S. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comp.*, 43(8):983–992, 1994. (Also: LAPACK Working Note 59).

[9] I. S. Dhillon, B. N. Parlett, and C. Vömel. LAPACK working note 162: The design and implementation of the MRRR algorithm. Technical Report UCBCSD-04-1346, University of California, Berkeley, 2004.

[10] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (release I). Technical Report RAL-TR-92-086, Atlas Centre, Rutherford Appleton Laboratory, 1992.

[11] I. S. Duff, R. G. Grimes, and J. G. Lewis. The Rutherford-Boeing sparse matrix collection. Technical Report RAL-TR-97-031, Atlas Centre, Rutherford Appleton Laboratory, 1997. Also Technical Report ISSTECH-97-017 from Boeing Information & Support Services and Report TR/PA/97/36 from CERFACS, Toulouse.

[12] E. Apra et al. NWChem, a computational chemistry package for parallel computers, version 4.7. Technical report, Pacific Northwest National Laboratory, Richland, WA. USA, 2005.

[13] R. A. Kendall et al. High performance computational chemistry: An overview of NWChem a distributed parallel application. *Computer Phys. Comm.*, 128:260–283, 2000.

[14] K. Fernando and B. N. Parlett. Accurate singular values and differential qd algorithms. *Numerische Mathematik*, 67:191–229, 1994.

[15] K. V. Fernando and B. N. Parlett. Implicit Cholesky algorithms for singular values and vectors of triangular matrices. *Numerical Linear Algebra with Applications*, 2(6):507–531, 1995.

[16] G. H. Golub and C. van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, Maryland, 3. edition, 1996.

[17] Intel Corporation. *Intel Itanium 2 Processor Reference Manual For Software Development and Optimization*, May 2004. 251110-003.

[18] Intel Corporation. *IA-32 Intel Architecture Optimization Reference Manual*, June 2005. 248966-012.

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

[20] W. Kahan. A demonstration of presubstitution for ∞/∞. unpublished, 2005.

[21] M. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, Philadelphia, PA, USA, 2001.

[22] B. N. Parlett. *Acta Numerica*, chapter The new qd algorithms, pages 459–491. Cambridge University Press, 1995.

[23] B. N. Parlett and I. S. Dhillon. Fernando's solution to Wilkinson's problem: an application of double factorization. *Linear Algebra and Appl.*, 267:247–279, 1997.

[24] B. N. Parlett and I. S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Linear Algebra and Appl.*, 309(1–3):121–151, 2000.

[25] B. N. Parlett and I. S. Dhillon. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and Appl.*, 387:1–28, 2004.

[26] B. N. Parlett and I. S. Dhillon. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, 2004.

[27] B. N. Parlett and O. Marques. An implementation of the dqds algorithm (positive case). *Linear Algebra and Appl.*, 309:217–259, 2000.

[28] H. Rutishauser. Der Quotienten-Differenzen-Algorithmus. *Z. Angew. Math. Phys.*, 5:233–251, 1954.

[29] H. Rutishauser. *Vorlesungen über Numerische Mathematik*. Birkhäuser, Basel, 1976.

[30] Silicon Graphics, Inc. *Origin 2000 and Onyx2 Performance Tuning and Optimization Guide*, August 2002. 007-3430-003.

[31] Sun Microsystems, Inc. *UltraSPARC-IIi User's Manual*, 1997. 805-0087-01.

[32] Scott Vetter, Stefan Andersson, Ron Bell, John Hague, Holger Holthoff, Peter Mayes, Jun Nakano, Danny Shieh, and Jim Tuccillo. *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*. International Business Machines, Inc., 1st edition, October 1998.