Design, Implementation and Testing of Extended and Mixed Precision BLAS [*]

X. S. Li[†] J. W. Demmel[‡] D. H. Bailey[†] G. Henry[§] Y. Hida[‡] J. Iskandar[‡]
W. Kahan[‡] A. Kapur[‡] M. C. Martin[‡] T. Tung[‡] D. J. Yoo[‡]

October 20, 2000

Abstract

This paper describes the design rationale, a C implementation, and conformance testing of a subset of the new Standard for the BLAS (Basic Linear Algebra Subroutines): Extended and Mixed Precision BLAS. Permitting higher internal precision and mixed input/output types and precisions permits us to implement some algorithms that are simpler, more accurate, and sometimes faster than possible without these features. The new BLAS are challenging to implement and test because there are many more subroutines than in the existing Standard, and because we must be able to assess whether a higher precision is used for internal computations than is used either for input or output variables. So we have developed an automated process of generating and systematically testing these routines. Our methodology is applicable to languages besides C. In particular, our algorithms used in the testing code would be very valuable to all the other BLAS implementors. Our extra precision routines achieve excellent performance—close to half of the machine peak Megaflop rate even for the Level 2 BLAS, when the data access is stride one.

[*]This research was supported in part by the National Science Foundation Cooperative Agreement No. ACI-9619020, NSF Grant No. ACI-9813362, the Department of Energy Grant Nos. DE-FG03-94ER25219 and DE-FC03-98ER25351, and gifts from the IBM Shared University Research Program, Sun Microsystems, and Intel. This project also utilized resources of the National Energy Research Scientific Computing Center (NERSC) which is supported by the Director, Office of Advanced Scientific Computing Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

[†]NERSC, Lawrence Berkeley National Laboratory, 1 Cyclotron Rd, Berkeley, CA 94720 (xiaoye@nersc.gov, dhbailey@lbl.gov).

[‡]Computer Science Division, University of California, Berkeley, CA 94720 ({demmel, wkahan, yozo, anil, djyoo}@cs.berkeley.edu, {iskandar, mcmartin, teresat}@cory.eecs.berkeley.edu}).

[§]Intel Corp, ESG-SSPD, Bldg. CO1-01, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006-5733 (greg.henry@intel.com).

Contents

1 Introduction

The Basic Linear Algebra Subprograms (BLAS) [36, 22, 21] have been widely used by both the linear algebra community and the application community to produce fast, portable and reliable software. Highly efficient machine-specific implementations of the BLAS are available for most modern high-performance computers. Because of the success of these earlier BLAS, a BLAS Technical Forum was established to consider expanding the BLAS in the light of modern software, language, and hardware developments. The BLAS Technical Forum meetings began with a workshop in November 1995 at the University of Tennessee. Meetings were hosted by universities and software and hardware vendors. Various working groups were established to consider such issues as the overall functionality; language interfaces for Fortran 77, Fortran 95, and C; dense, banded, and sparse BLAS; and extended and mixed precision BLAS. As a result, a new BLAS Standard is being established, and the working groups have started implementing different parts of the Standard. The document for the new BLAS Standard is available on-line [1].

In this paper, we describe the design rationale, a reference C implementation, and testing code for a representative subset of the routines in the Extended and Mixed Precision BLAS [1, Chapter 4]. (This also contains the corresponding routines without extended and mixed precision that are part of the Dense and Banded BLAS [1, Chapter 2].) Extended and mixed precision, which we define more carefully below, permit us to implement some algorithms that may be simpler, more accurate, and sometimes even faster than without it. We give numerous examples in section 2.

Extended precision is only used internally in the BLAS; the input and output arguments remain just Single or Double as in the existing BLAS. For the internal precision, we allow *Single*, *Double*, *Indigenous*, or *Extra*. In our reference implementation we assume that Single and Double are the corresponding IEEE floating point formats [3]. *Indigenous* means the widest hardware-supported format available. Its intent is to let the machine run close to top speed, while being as accurate as possible. On some machines this would be a 64-bit (Double) format, but on others such as Intel machines it means the 80-bit IEEE format (which has 64 fraction bits). Our reference implementation currently supports machines on which Indigenous is the same as Double. *Extra* means anything at least 1.5 times as accurate as Double, and in particular wider than 80 bits. An existing quadruple precision format could be used to implement Extra precision, but we cannot assume that the language or compiler supports any format wider than Double. So for our reference implementation, we use a technique called *double-double* in which extra precise numbers are represented by a pair of double precision numbers [6], providing about 106 bits of precision. Section 3 discusses a variety of implementation techniques for precision beyond double.

In particular, our design goal does *not* include a general extended precision capability as in systems like Mathematica [51], Maple [40] and others, where all variables, arithmetic operations, and intrinsic functions can have arbitrarily high precision. Instead, we are interested in enhancing conventional floating point algorithms written in C or Fortran, with support only for the conventional single and double precision data types. In other words, we are interested in applications where performance remains a primary concern, and our job is either to improve that performance, or make algorithms more accurate or simpler without lowering performance significantly if at all.

The subroutines implementing extra precision look like their conventional counterparts, but have an an additional argument PREC at the end. PREC is a runtime variable specifying whether the internal precision used for the computation should be Single, Double, Indigenous, or Extra.

Mixed precision permits some input/output arguments to be of different mathematical types, meaning real and complex, or different precisions, meaning single and double. This permits such operations as real-by-complex matrix multiplication, which can rather faster than using alternatives that do not mix precision. A parsimonious subset of the many possible combinations of arguments types and precisions is mandated. The mixed precision routines are similar to their conventional counterparts, except that they include versions for each permitted input type combination. Routines that combine mixed and extended precision (the PREC argument) are also included.

The multitude of precisions and types means that the Extended and Mixed Precision BLAS contains many more routines than their conventional counterparts. This motivated us to generate them automatically from a single template. Since the BLAS Technical Forum voted to support C and Fortran 77 interfaces, and these languages do not permit us to write a single source to handle all desired type combinations, we decided to use the M4 macro package for automatic program generation. We note that conventional operator overloading mechanisms in Fortran 95 and C++ that choose the precision of an operation based only on on the input arguments are *not* adequate to generate our algorithms. This is described in section 5.

It is a challenge to design portable and reliable test routines that confirm whether a routine with arguments of one type and precision uses a higher internal precision, indeed a precision (like Extra) that is higher than any language and compiler supported format. Most of our design effort went into designing this test code, which we describe in detail in section 6. We use a variety of algebraic identities and tricks to automatically generate test data that will make the internal precision visible through black box testing. These techniques should be valuable for other testing efforts as well.

The complete list of BLAS routines for which we decided Extended and Mixed Precision counterparts were worthwhile is given below. (Some routines, like computing $\sqrt{\sum_{i=1}^{n} |x_i|^2}$, hardly benefit from extra precision and so are excluded.) The routines not included in our current reference implementation are marked "omitted". The omitted routines offer no more technical challenges than those already included, and will be released at a later date. For simplicity, formulas are only given for the real versions; the complex ones permit some arguments to be conjugated.

- Level 1
  - DOT (inner product) $r := \alpha \cdot \sum_{i=1}^{n} x_i \cdot y_i + \beta \cdot r$
  - SUM (sum) $s := \sum_{i=1}^{n} x_i$
  - AXPBY (Scaled vector accumulation) $y_i := \alpha \cdot x_i + \beta \cdot y_i$
  - WAXPBY (Scaled vector addition) $w_i := \alpha \cdot x_i + \beta \cdot y_i$
- Level 2
  - GEMV (General matrix vector product) $y := \alpha \cdot A \cdot x + \beta \cdot y$ or $y := \alpha \cdot A^T \cdot x + \beta \cdot y$
  - GBMV (Banded matrix vector product) (same mathematical formulas as GEMV)
  - SPMV (Symmetric matrix vector product, packed format) (same mathematical formulas as GEMV)
  - omitted: SYMV (Symmetric matrix vector product)
  - omitted: SBMV (Symmetric band matrix vector product)
  - TRSV (Triangular solve) $x := \alpha \cdot T^{-1} \cdot x$ or $x := \alpha \cdot T^{-T} \cdot x$
  - omitted: TBSV (Band triangular solve)

  - omitted: TPSV (Packed triangular solve)

- Level 3
  - GEMM (General matrix matrix product) $C := \alpha \cdot A \cdot B + \beta \cdot C$, either A and/or B may be transposed
  - SYMM (Symmetric matrix matrix product) $C := \alpha \cdot A \cdot B + \beta \cdot C$ or $C := \alpha \cdot B \cdot A + \beta \cdot C$
  - omitted: TRMM (Triangular matrix matrix product)
  - omitted: TRSM (Triangular matrix solve)
  - omitted: SYRK (Symmetric rank k update)
  - omitted: SYR2K (Symmetric rank $2k$ update)

The rest of the paper is organized as follows. Section 2 gives the rationale and motivating examples for standardizing the extended and mixed precision BLAS. This includes an extended example showing how linear systems can be solved more accurately using only an extra precise matrix-vector product routine. Section 3 surveys various implementation techniques for extended precision, both in hardware and software, including performance data for our reference implementation. When the data access is in stride one, our extra precision routines achieve sustained Megaflop rates of close to half of the machine peak for both Level 2 and Level 3 BLAS. Section 4 summarizes our design decisions based on the preceding discussions of the benefits and costs. Section 5 describes our automatic code generator that easily accommodates the large number of new routines. Section 6 presents the core algorithms to test the correctness of all the functions. Section 7 summarizes the issues in our design and discusses future work.

2 Rationale for Extended and Mixed Precision

2.1 Introduction

Our proposal to have extended and mixed precision in the BLAS is motivated by the following facts:

- A number of important linear algebra algorithms can become simpler, more accurate and sometimes faster if some internal computations carry more precision (and sometimes more range) than is used for the input and output arguments. These include linear system solving, least squares problems, and eigenvalue problems. Often the benefits of wider arithmetic cost only a small fractional addition to the total work.

- For single precision input, the computer's native double precision is a way to achieve these benefits easily on all commercially significant computers, at least when only a few extra-precision operations are needed. In our reference implementation, we assume single and double precision correspond to the IEEE 32-bit and 64-bit formats. [1]

[1] Some Crays and their emulators implement 64-bit single (`REAL` in Fortran) in hardware and much slower 128-bit double (`DOUBLE PRECISION` in Fortran) in software, so if a great many double precision operations are needed, these machines will slow down significantly.

- The overwhelming majority of computers on desktops, containing Intel processors or their AMD and Cyrix clones, are designed to run fastest performing arithmetic to the full width, 80-bits, of their internal registers. These computers confer some benefits of wider arithmetic at little or no performance penalty. Some BLAS on these computers already perform wider arithmetic internally but, without knowing this for sure, programmers cannot exploit it.

- All computers can simulate quadruple precision or something like it in software at the cost of arithmetic slower than double precision by at worst an order of magnitude. Less slowdown is incurred for a rough double-double precision on machines (IBM RS/6000, PowerPC/Mac, SGI/MIPS R8000, HP PA RISC 2.0) with special Fused Multiply-Accumulate instructions. Since some algorithms require very little extra precise arithmetic to get a large benefit, the slowdown is practically negligible.

Given the variety of implementation techniques hinted above, we need to carefully examine the costs and benefits of exploiting various arithmetic features beyond the most basic ones, and choose a parsimonious subset that gives as many benefits as possible with modest implementation cost.

We list the arithmetic features potentially available to us below:

Feature 1: Mixed precision. This means mixing single and double arguments, or real and complex arguments, in a single BLAS call. For example, it can be much cheaper to multiply a real by a complex matrix, than to convert the real matrix to complex and then multiply two complex matrices. This is straightforward to implement.

Feature 2: Extra internal precision. This means using higher precision within a BLAS call, with all high-precision variables hidden from the user. For example, a call to DGEMV might perform a matrix-vector multiplication on IEEE double-precision inputs (with 53-bit mantissas), but use an accumulator internally with a 64- or 106-bit mantissa. This is the simplest way to introduce extra precision.

Feature 3: Wider internal range. This means using floating point numbers with a wider overflow/underflow range within a BLAS call, again with all such variables hidden from the user. For example, a call to SNRM2 might compute the root-sum-of-squares of an IEEE single-precision vector (with nonzero magnitudes in the range $\approx 10^{\pm 38}$) by using an IEEE double precision accumulator (with range $\approx 10^{\pm 308}$). Wider range may not always be available at a reasonable price.

Feature 4: Extra-wide variables. Rather than hiding all uses of extra precision or extra range within BLAS calls, one could permit the user to declare extra-wide variables and use them as input/output arguments. By this we mean using double precision (or quadruple precision) variables when the working precision is single (or double), not the general mechanism in Mathematica and Maple. Use of extra-wide variables offers the most flexibility to the user, but if the program is already in double precision, quadruple (or other language and compiler-supported wide precision) may not always be available.

Feature 5: Exception handling. IEEE arithmetic defines precise responses to exceptional events like overflow, underflow, division-by-zero, invalid operations (like $\sqrt{-1}$), and inexact [3, 4]. In particular, it has rules for arithmetic with NaNs (Not-a-Number symbols) (produced by

$\sqrt{-1}$, 0/0, etc.) and $\pm\infty$ (produced by overflow, 1/0, etc.). It also defines *flags* that the user can reset and later test to see if any exception has occurred since resetting them. This feature can let us use simpler and faster algorithms that rarely generate exceptions, and afterwards check for these exceptions and recompute the answer slowly and carefully only if necessary [15]. Unfortunately, languages and compilers do not support access to exception handling in a uniform way.

The rest of this section lists a variety of compelling examples that exploit the arithmetic features above, and categorizes them according to which features they use. Using these examples, Section 4 will present and justify our design decisions.

2.2 Example 1: Iterative Refinement of Linear Systems and Least Squares Problems

Given a triangular factorization of a matrix A, we can solve $Ax = b$ by forward and back substitution. In practice the relative error of this algorithm is almost always bounded by

$$\frac{\|x_{computed} - x_{true}\|}{\|x_{true}\|} = O(N \cdot \kappa(A) \cdot \varepsilon)$$

where $\|\cdot\|$ denotes the norm of a matrix or vector, $\kappa(A) = \|A\| \cdot \|A^{-1}\| \geq 1$ is the *condition number* of A, N is its dimension, and ε is the *input/output* precision (for example $2^{-53} \approx 10^{-16}$ in double precision). We expect the error to be at least about ε, just from rounding the true solution to a vector $x_{computed}$ of floating point numbers. We expect a large error when $\kappa(A) \gg 1$, i.e. the problem is ill-conditioned, which is frequently the case for problems of interest. It turns out that using a little bit of extra precision, we can eliminate $N \cdot \kappa(A)$ from the error bound, and so get an accurate answer nearly independently of N and condition number $\kappa(A)$.

To achieve this, we use the following *iterative refinement* algorithm to improve the accuracy of the solution:

```
Repeat
    Compute the residual $r = b - A \cdot x$
    Solve $A \cdot d = r$ for $d$ using the existing factorization
    Update the solution $x = x + d$
Until {$r$ or $d$ is small enough or stops decreasing, or a maximum iteration count is exceeded}.
```

LAPACK currently implements this iterative refinement algorithm entirely in the input/output precision, such as in routine `xGERFS` [2]. What it accomplishes, according to a well understood error-analysis [16, 27], is essentially to replace the condition number $\kappa(A)$ in the error bound by another possibly smaller one, which can be as small as $\min_D \kappa(D \cdot A)$, where the minimum is over all diagonal matrices D. In effect, this algorithm compensates, up to a point, for bad row-scaling. The residual is never worsened but the solution x, though usually improved, frequently gets worse if this last condition number is very big.

An improved version of iterative refinement, which nearly eliminates $\kappa(A)$ from the final error bounds, differs from LAPACK's in two places. First, the residual $r = b - A \cdot x$ is accumulated in twice the input/output precision and then, after massive cancellation has taken place, is rounded to input/output precision. Second, refinement is stopped after x appears to have settled down about as much as it ever will. Now another classical error analysis [27] says that the ultimate relative error will be bounded by $O(\varepsilon)$, essentially independent of the condition number and dimension of A. This assumes that $N \cdot \kappa(A)$ is not comparable with or bigger than $1/\varepsilon$ (in which case A could be made exactly singular by a few rounding errors in its entries). In short, the improved iterative refinement almost always produces the correct solution x, and almost always gives fair warning otherwise.

If the improved algorithm's residual r is accumulated to sufficiently more than input/output precision but less than twice as much, its improvement over what LAPACK obtains currently becomes proportionately weakened. In extensive experiments on machines with 80-bit registers, i.e. 11 more bits of precision than double, iterative refinement either returned the solution correct to all but the last few bits, or with at least 10 more correct bits than without refinement [32].

The current LAPACK algorithm does not have to change much to benefit from a more accurate residual: ε has to be replaced in a few places by a roundoff threshold that reflects the possibly extra-precise accumulation of r; and if this latter threshold is sufficiently smaller than ε then the iteration's stopping criterion should be changed from testing r to testing x and d, and an estimate of the error in x should be obtained not from the current condition estimator but instead (at lower cost) from the last few vectors d.

The indispensable requirement is an extra-precise version of matrix-vector product, i.e. GEMV in the Level 2 BLAS (**Feature 2: Extra internal precision**). One iteration of refinement costs only $4 \cdot N^2$ flops per right-hand side column b. If, as usual, there are only a few columns b and at most a few iterations suffice, their cost is asymptotically negligible compared to the $O(N^3)$ operations required to compute the LU or other factorization in the first place, even if extra-precise operations run somewhat slowly. Therefore, we want to be able to insist that GEMV use extra precision, even if it is expensive.

An extended numerical example is given below in section 2.2.1.

The availability of extra-precise residuals would make it possible to solve many other linear algebra problem more accurately as well. The simplest example is the overdetermined least squares problem: choosing x to minimize $\|A \cdot x - b\|_2$. This can be formulated as the linear system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix} \cdot \begin{bmatrix} -r \\ x \end{bmatrix} = \begin{bmatrix} b \\ 0 \end{bmatrix}$$

and refined using the QR decomposition of A from which the initial solution x was obtained. As with general linear systems, the final error depends much less upon the condition number of A (though the situation is a bit more complicated than a simple linear system). Again, an extra precise matrix-vector product is indispensable (**Feature 2: Extra internal precision**), and its cost is asymptotically small provided that A is not too far from square.

Iterative refinement for eigenproblems is still a topic for research, but recent results are encouraging. The indispensable requirement is a matrix residual

$$R = A \cdot X - Y \cdot B = \begin{bmatrix} A & -Y \end{bmatrix} \cdot \begin{bmatrix} X \\ B \end{bmatrix}$$

accumulated extra-precisely and rounded back to input/output precision after massive cancellation has taken place. Sometimes $Y = X$; sometimes B is diagonal or triangular. Such residuals can be computed from one call to a matrix-matrix-product (GEMM) that accumulates extra-precisely, but a lot of memory traffic can be avoided if versions of GEMM that accept and produce extra-precise matrices are available too (**Feature 4: Extra-wide variables**).

Iterative refinement of nonsymmetric eigenproblems, other than Schur decompositions, depends crucially upon how well iterative refinement works for nearly singular linear systems. For best results, the LU factorization should ideally be performed by Crout factorization with extra-precisely accumulated scalar products, but how much good this would do is not yet clear. **Exception handling (Feature 5)** is important too because infinity and 0/0 turn up in the better mathematical algorithms mentioned above unless entirely algebraic computations are replaced in spots by invocations of transcendental functions elementwise upon arrays. The last question remaining is how much precision would be needed to perform an entire eigencalculation by conventional means, without refinement, to obtain results as good and as soon as are obtained from refinement with a little extra-precise arithmetic in the program; experience indicates that a high precision conventional algorithm without refinement is much more expensive than with refinement.

2.2.1 Numerical Example

In this section we study a (well-known) numerical example illustrating the benefits of extra-precise iterative refinement. We used the linear system $(L \cdot H_n) \cdot x = (L \cdot b)$, where H_n is the Hilbert matrix of dimension n, b is the j-th column of the identity matrix (which we denote by I_j in the Figures), and L is a scale factor chosen so that $L \cdot H_n$ and $L \cdot b$ have integer entries representable without roundoff. The Hilbert matrix H has elements $h_{ij} = 1/(i+j-1)$. We chose the Hilbert matrix because (1) it is very sensitive to roundoff errors, with condition numbers increasing rapidly (see Table 1), and (2) its inverse is known exactly and so we can compute the true solution. L is the least common multiple of all the $i+j-1$, $1 \le i,j \le n$.

n	2	3	4	5	6	7	8	9
$\kappa_2(H_n)$	1.9e01	5.2e02	1.6e04	4.8e05	1.5e07	4.8e08	1.5e10	4.9e11

Table 1: Condition numbers of Hilbert matrix [27, pp. 516].

Figures 1, 2 and 3 show the convergence history of iterative refinement, with progressively increasing n. We use both the LAPACK routine `SSPRFS` (single precision, symmetric and packed) and our modification to it with `SSPMV` replaced by our extended precision version. In this experiment, we let the code run 20 iterations, because the examples are very ill-conditioned with respect to single precision ($\varepsilon \approx 5.96\text{e-}08$). In practice, one or two iterations often suffice. In each figure, we plot five quantities ($\|v\|_\infty \equiv \max_i |v_i|$).

1. Forward error bound $FERR$ on x computed by LAPACK (see [16] for the derivation):

$$\frac{\|x - x_{true}\|_\infty}{\|x\|_\infty} \le FERR \equiv \frac{\|\, |A^{-1}| \cdot (|r| + n\varepsilon(|A| \cdot |x| + |b|))\, \|_\infty}{\|x\|_\infty}$$

2. Componentwise relative backward error [16], computed by LAPACK, which is the smallest relative perturbation in each matrix entry necessary to make the computed x the exact solution of the perturbed problem:

$$BERR \equiv \max_i \frac{|r|_i}{(|A|\ |x| + |b|)_i}$$

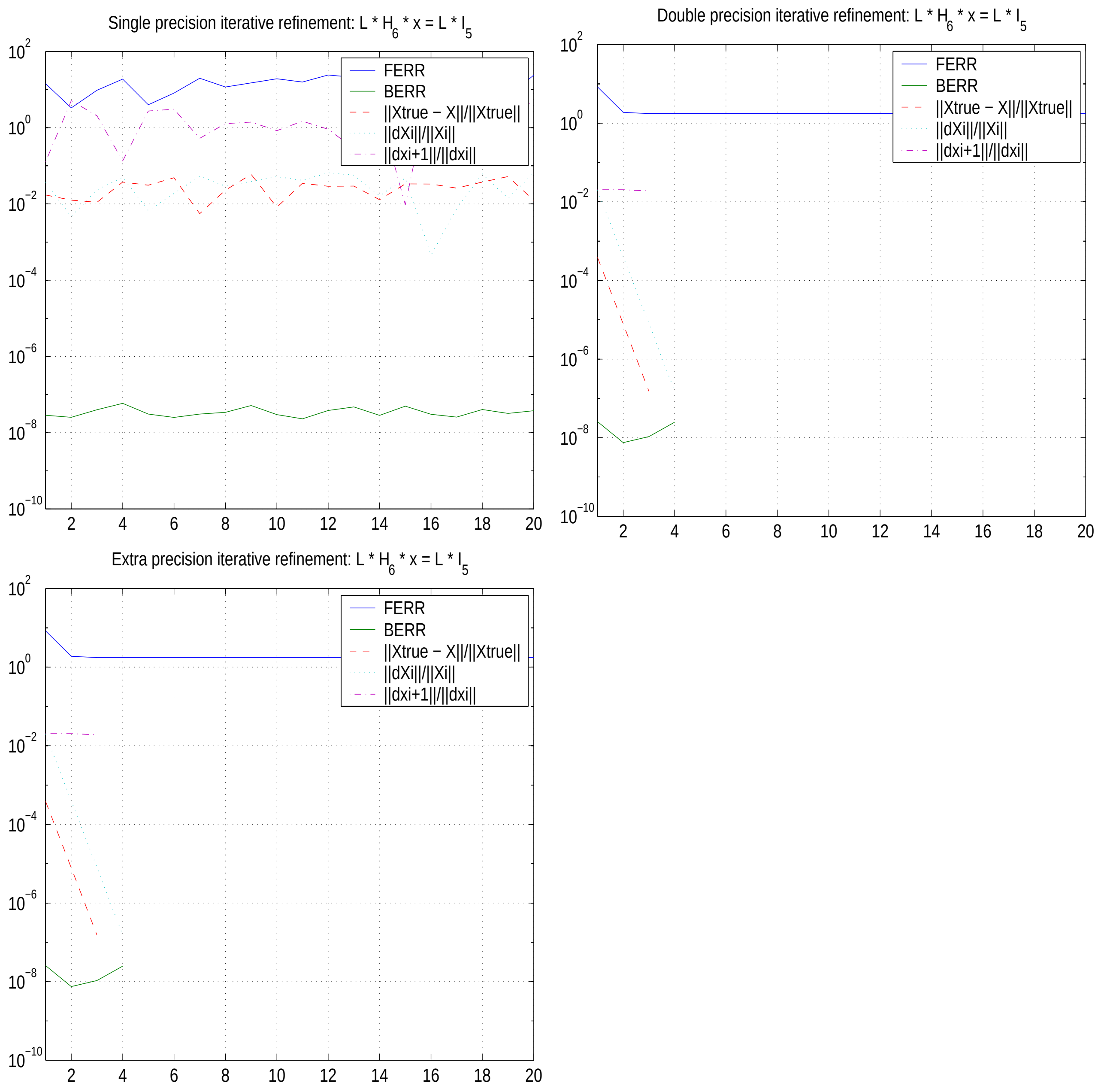

Figure 1: Iterative refinement results in Single, Double and Extra precision, $n = 6, j = 5$.

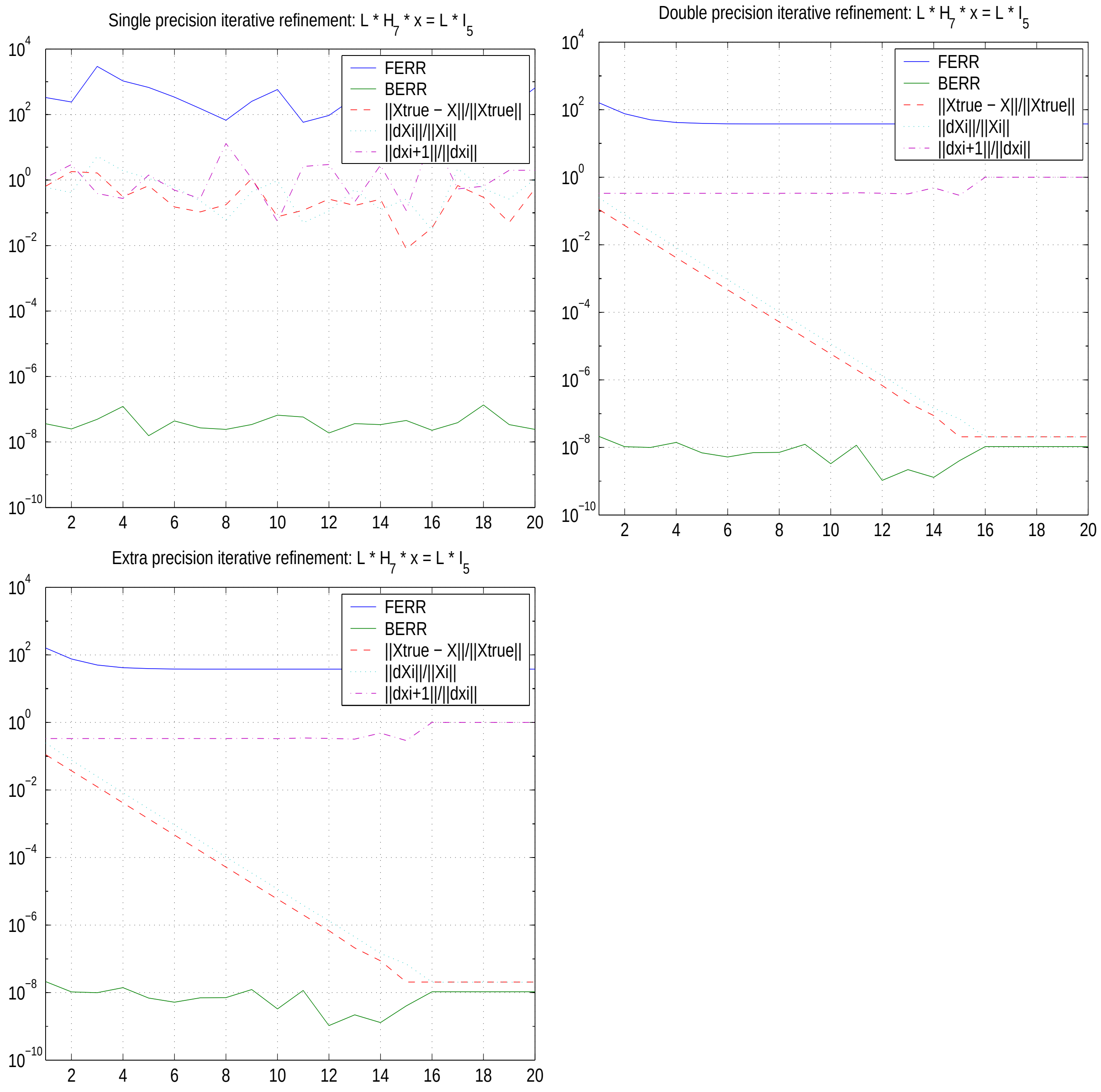

Figure 2: Iterative refinement results in Single, Double and Extra precision, $n = 7, j = 5$.

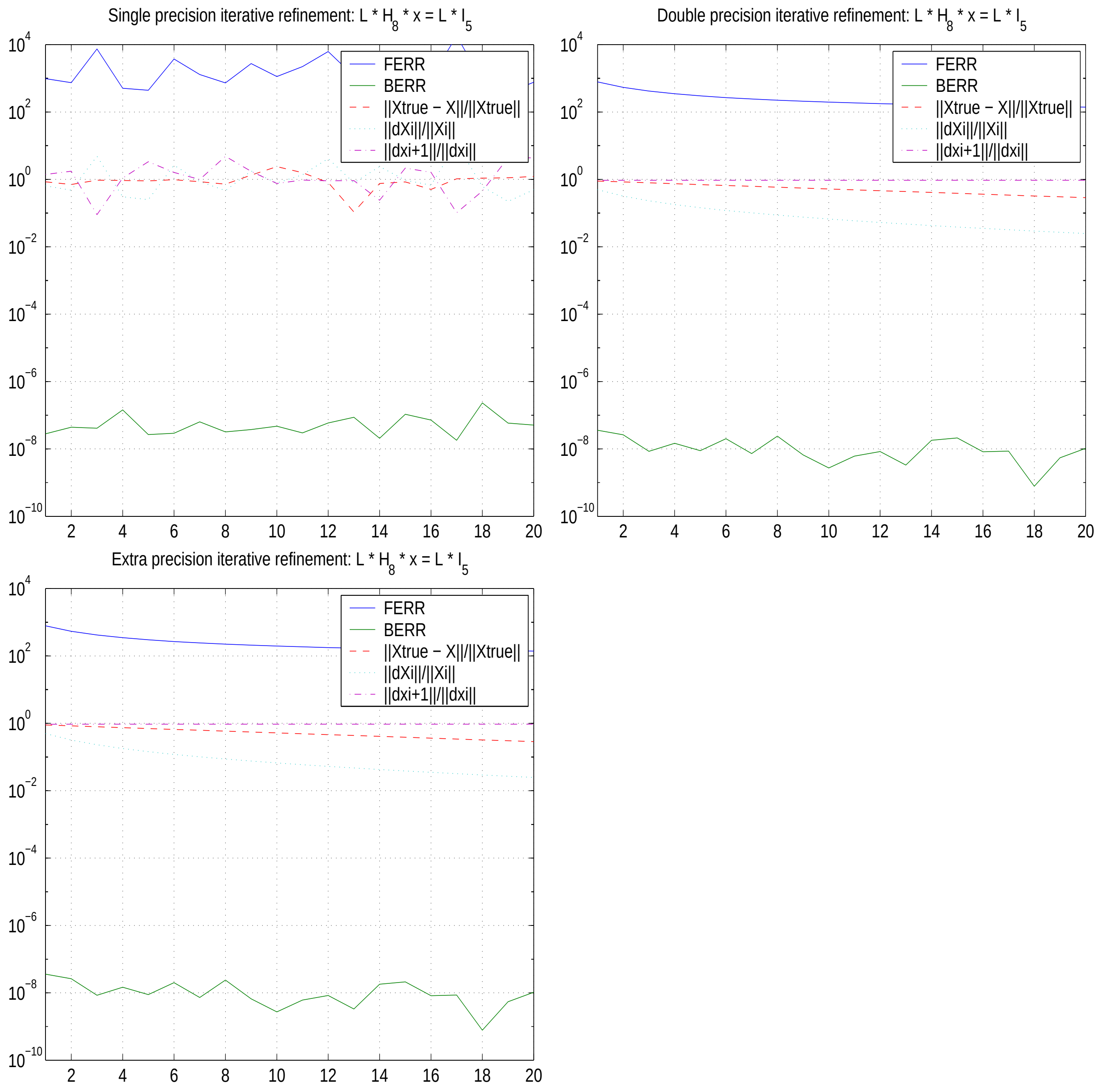

Figure 3: Iterative refinement results in Single, Double and Extra precision, $n = 8, j = 5$.

3. The true error $||x_{true} - x||_\infty / ||x_{true}||_\infty$,

4. the normalized change in x between successive iterations $||x_{i+1} - x_i||_\infty / ||x_i||_\infty$, and

5. the ratio of changes of x between successive iterations $||x_{i+2} - x_{i+1}||_\infty / ||x_{i+1} - x_i||_\infty$.

In the labels in the figures, we let $dx_i = x_{i+1} - x_i$.

In Figure 1, four error measures become exactly zero after 3 or 4 steps, so nothing is plotted afterwards. From this experiment, we made the following observations:

- The single precision iteration can effectively reduce BERR to near machine epsilon, although it is slightly worse than the extended precisions. But it cannot reduce the true error of the solution.

- Both double and extra precisions can reduce the true error to close to machine epsilon when the condition number is comparable to or less than $1/\varepsilon$ (when $n = 6$ or $n = 7$ but not $n = 8$). Using more than double the input/output precision gives comparable results to just using double the input/output precision.

- The error bound FERR is good only when the iterative refinement is performed in single precision. If extended precision is used, FERR may be arbitrarily pessimistic. Instead, $||dx_i||/||x_i||$ should be used as the error bound.

Figure 4 is the plot of the relative errors after 20 steps, when using double precision and varying the dimension n from 3 to 10. The vertical line in the figure corresponds to $1/\varepsilon$. This also confirms the theory that the improved iterative refinement always produces accurate solution as long as the condition number is not bigger than $1/\varepsilon$.

2.3 Example 2: Avoiding Pivoting in Sparse Gaussian Elimination

Sparse Gaussian Elimination is a challenging algorithm to implement efficiently because of the high cost of constructing, traversing and modifying the dynamic data structures that are needed to compute the pivot sequence and resulting fill-in of originally nonzero matrix entries. This cost is especially acute on distributed memory parallel machines because known algorithms require frequent fine-grained messages, and these are expensive. In contrast, sparse Cholesky is a special case of Gaussian elimination applicable to symmetric positive definite matrices, for which *any* pivot sequence is numerically stable. Sparse Cholesky has been satisfactorily parallelized because the pivot sequence can be chosen ahead of time (i.e. independent of the values of the nonzero matrix entries, or any intermediate results) to optimize load balance and fill-in, and the data structures and communication patterns can be determined statically and more simply [25].

We would like to have an algorithm as parallelizable as sparse Cholesky but applicable to completely general matrices. To do this we have designed a version of parallel sparse Gaussian elimination called SuperLU [37] that avoids dynamic pivoting, and so permits the same optimizations as parallel sparse Cholesky. To retain numerical stability, we use a variety of techniques including

1. prepivoting large matrix entries to the diagonal,

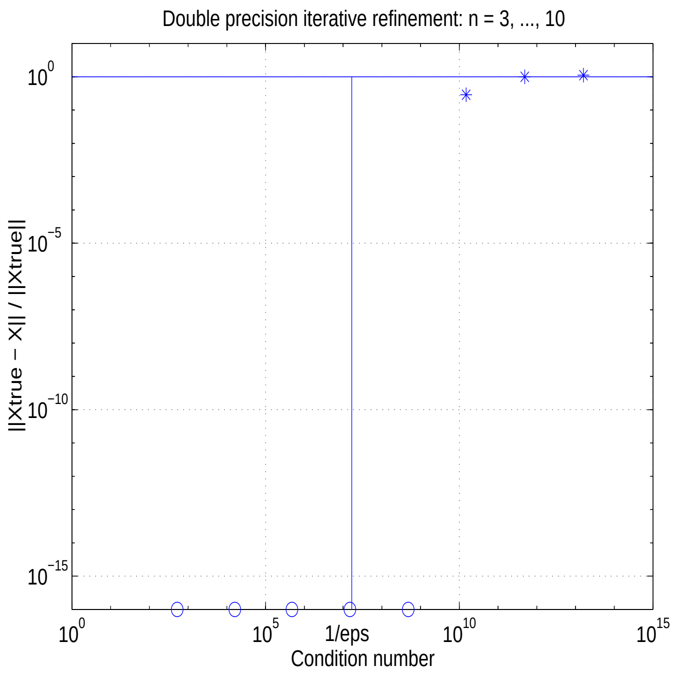

Figure 4: Iterative refinement results in Double precision, $n = 3, ..., 10$. Circles correspond to exact zeros.

2. substituting tiny diagonal pivots by $\sqrt{\varepsilon}\|A\|$ (which reduces pivot growth at the price of losing *half* the backward stability enjoyed by conventional pivoting), and

3. performing iterative refinement with extra precise residuals, as in Section 2.2 (**Feature 2: Extra internal precision**).

Preliminary results, where double precision is used internally to achieve single precision accuracy for the computed results [37], show that the above techniques do ensure satisfactory accuracy on a set of challenging test matrices drawn from applications, and also show large speedups on suitably large matrices.

If the above techniques do not get adequate accuracy (we encountered only one such matrix in our extensive testing), then the following further techniques can be applied. One approach is to organize the Gaussian elimination in a left-looking fashion, where each entry is updated by the dot product of the corresponding row of L and column of U. For this update, we can call the new DOT function (or GEMV/GEMM in a blocked algorithm). This only requires **Feature 2: Extra internal precision**. Another technique is to use this inaccurate LU factorization as a preconditioner to an iterative solver, such as QMR or GMRES. For example, for the matrix mentioned above, we achieved accurate solution with a few iterations of preconditioned GMRES.

2.4 Example 3: Accelerating Iterative Methods for $Ax = b$, like GMRES

The principle of iterative refinement can be applied to other equation solving methods as well, both linear and nonlinear. The paper [49] replaces the solution of $Ad = r$ via the LU factorization of A

by GMRES(m) [46] to get the correction d, and solves a linear system arising from a discretized elliptic PDE as accurately as though the entire computation were done in double, but while doing nearly all the work in single and so running much faster. The authors of [49] point out that their observation could have been applied to any other efficient, restarted iterative solver besides GMRES(m), such as GCR(m) [23], etc.

The algorithm in [49] differs slightly from the one in Section 2.2 in that the right hand side b, the computed solution x, and the product $A \cdot x$ are kept in extended precision (double in their experiments). In other words, they exploit **Feature 4: Extra-wide variables**, in addition to **Feature 2: Extra internal precision**. It is not clear from [49] whether they really need extra-wide variables, or whether the general scheme of Section 2.2 is adequate; we suspect that it is.

2.5 Example 4: Mixed Real and Complex Arithmetic

LAPACK auxiliary routine CLACRM multiplies a complex matrix A by a real matrix B. It is called by the LAPACK routines for computing eigenvalues and eigenvectors of Hermitian matrices using divide-and-conquer (CSTEDC, CLAED0 and CLAED7), currently one of two fastest available algorithms for large matrices. It is similarly used by the complex SVD based on divide-and-conquer. The alternative of promoting B to complex and calling the BLAS CGEMM would cost 3 times as many floating point operations (4 floating point multiplications and 2 additions per solution component, instead of 2 multiplications). Since this routine is the most time-consuming part of the divide-and-conquer algorithm, it is important that it go fast. Currently CLACRM is implemented by copying the real part of A to contiguous workspace, calling SGEMM, copying the result back, and repeating this for the imaginary part. It would be simpler and more efficient (by avoiding these data copies) to have a matrix-matrix-product routine that accepted a real and complex matrix (**Feature 1: Mixed precision**).

LAPACK auxiliary routines CLASR and CSROT apply one or more real 2-by-2 rotations R_i to a complex matrix A. They are called by the LAPACK routines for computing eigenvalues and eigenvectors of Hermitian matrices using QR iteration (CSTEQR), and by the complex SVD, and are the bottlenecks in these routines, which are currently the fastest available algorithms for small matrices. The alternative of promoting the R_i to complex would again triple the cost of the most time-consuming part of QR iteration. Currently these routines are implemented in straightforward (unoptimized) Fortran 77 but could be implemented as optimized BLAS with mixed precision inputs and outputs (**Feature 1: Mixed precision**).

It is possible to define a very large number of mixed precision routines, not all of which are useful. For example, the standard matrix-matrix product routine has five floating point arguments (it computes $\alpha AB + \beta C$). If each argument were independently permitted to have one of the 4 standard floating point types (single, double, single complex, double complex), then there would be $4^5 = 1024$ different combinations, most of which would never be used. Instead, we have identified a small subset of just 12 mixed precision versions (plus the 4 conventional versions) to implement; see section 4.

2.6 Example 5: Using Normal Equations Instead of QR for Least Squares

This example uses **Feature 4: Extra-wide variables**, to implement a faster algorithm for the overdetermined least squares problem, i.e. finding the x that minimizes $\|Ax - b\|_2$. The algorithm that uses the fewest floating point operations is to form the matrix $A^T \cdot A$ and then solve the so-called *normal equations*

$$(A^T \cdot A) \cdot x = A^T \cdot b$$

by Cholesky. This requires about half as many flops as the alternative QR-based approach when the number of rows m of A is much larger than the number of columns n. The trouble is that using the normal equations instead of QR can effectively square the condition number of the problem, and can lose twice as much accuracy. The remedy is to form and solve the normal equations in twice the input/output precision. Unless the condition number is near $1/\varepsilon$, this can be much more accurate than the QR-based approach in single precision. If we use more than input/output precision but less than twice as much, the error bound is proportionately more.

If m is sufficiently larger than n, and the cost of extra precise arithmetic is less than twice the cost of arithmetic in the input/output precision, then this algorithm will be faster than the existing QR based algorithm.

Preliminary benchmarks on a Pentium Pro indicate that forming and solving the normal equations using 80-bit extended arithmetic can be significantly faster than running LAPACK's implementation of QR, provided m is much larger than n. For example, with $m = 400$ and $n = 20$, the normal equation were 3 times faster than QR, i.e. faster than mere operation counts can explain. As n grows, the normal equations slow down with respect to QR, with about equal runtimes at $n = 45$, and with QR 20 times faster for $n = 400$. For $m = 500$ and $n = 5$, the normal equations were 39 times faster than QR!

Here is why the normal equations approach cannot be implemented solely with high precision restricted to internal precision for the BLAS. We need to be able to compute, return, and continue computing with $A^T \cdot A$ and $(A^T) \cdot b$ in higher precision. This means that we have to be able to declare variables of these types, not just have anonymous variables. Some languages do not permit declaring such high precision variables. Furthermore, Cholesky requires a square root, although LU could be used instead.

There is a family of Least Squares problems for which extended BLAS are necessary, but not the declaration of higher precision variables. These are simple three-dimensional nearest-point problems: Find the point in a given line or plane nearest another given line or point. These problems have explicit solutions expressed in terms of scalar- and cross-products [31, pp. 48–55]. Many another geometrical problem is solved by computing bilinear and quadratic forms during which massive cancellation is brought about by near-degenerate configurations that are incidental to the solution of a larger and well-behaved problem. The incidence of geometrically inconsistent results, or instead recourse to higher precision for all intermediate computations, can be diminished by orders of magnitude, sometimes to zero, if BLAS with internal extra precision (**Feature 2: Extra internal precision**) are used during the computation of those bilinear and quadratic forms. This approach is particularly attractive on computers that can evaluate extended precision BLAS about as fast as ordinary BLAS, as is the case for the overwhelming majority of computers on desktops containing, as they do, Intel or clones inside.

2.7 Example 6: Solving Ill-conditioned Triangular Systems

The LAPACK subroutine SLATRS solves a triangular linear system $T \cdot x = scale \cdot b$ with careful scaling to prevent over/underflow (here *scale* is an output argument chosen by the algorithm to avoid over/underflow). It is used instead of the Level 2 BLAS routine STRSV when we expect very ill-conditioned triangular systems, and want to get the solution vector (with the *scale* factor) despite the fact that it may be very large or very small. This is the case with condition estimation (SGECON, SPOCON, STRCON) and inverse iteration to find eigenvectors of nonsymmetric matrices (SLAEIN). SLATRS does careful scaling in the innermost loops to prevent over/underflow, and is significantly slower than an optimized STRSV. Also, it contains about 320 executable statements, (versus 160 for the unoptimized version of STRSV) a measure of the complexity of producing such code.

There are three ways to accelerate SLATRS. Since the standard interface to the BLAS routine for solving a triangular system does not allow for returning the *scale* factor, we believe that more than just **Feature 3: Wider internal range**, is needed.

The first way is to use **Feature 3: Wider internal range** along with **Feature 4: Extra-wide variables** to guarantee that over/underflow cannot occur for a given number of iterations of the outermost loop of (lower) triangular solve:

```
Solve $L \cdot x = b$ ... $L$ is lower triangular
  for $i = 1$ to $n$
    $x(i) = [b(i) - (\sum_{j=1}^{i-1} L(i,j) \cdot x(j))]/L(i,i)$
```

For example, if the entries of L and b are represented in IEEE single, and no diagonal entry of L is denormalized, and 80-bit Intel arithmetic is used to implement this algorithm, testing the scaling is needed only once every 64 values of i. In particular, this could be implemented by calling an optimized 80-bit GEMV on blocks of size 64, which would get much of the benefit of the optimized SGEMV. IEEE Double is not so good, since we could only do blocks of 8. Alternatively, we could use scaling code like that in SLATRS to compute the largest block size we could accommodate. This requires information about the norms of columns of L. In the case of SLAEIN, where we call SLATRS many times, the cost of this information is small. In the case of condition estimation, the cost is larger.

The second way to accelerate SLATRS uses **Feature 5: Exception handling**, and was explored in [15]. The IEEE exception flags can be used to hide over/underflow problems as follows: We first run using the optimized STRSV, and then check the exception flags. Since over/underflow is rare, this will almost always indicate that the solution is fine. Only in the rare cases that an exception occurs does one need to recompute more carefully. This is definitely the way to go on a Pentium. Speedups reported in [15] ranged from 1.4 to 6.5.

The third way to accelerate SLATRS, which could complement the above ways, is to more cleverly estimate how big a block we can call STRSV on, based on some data about sizes of off-diagonal entries of the matrix. SLATRS currently does this in a naive way, and we have explored better alternatives as well.

The complex analog CLATRS is used in the above routines as well as in some other LAPACK routines like CTREVC, which computes eigenvectors of triangular matrices. Speedups of CTREVC using exception handling reported in [15] ranged from 1.38 to 1.65. The overall routine CGEEV for the complex nonsymmetric eigenproblem sped up by 8%, since CTREVC is just part of the calculation.

2.8 Example 7: Eigenvalues and Eigenvectors of Symmetric Matrices

We discuss the acceleration of methods for computing the eigenvalues and eigenvectors of a symmetric matrix. In particular, they apply to the second phase of the algorithm, when the original matrix has been reduced to tridiagonal form. We use combinations of **Feature 3: Wider internal range**, **Feature 4: Extra-wide variables**, and **Feature 5: Exception handing**.

We begin with the recently released LAPACK code xSYEVR [2], which uses a new algorithm [18, 17, 42, 43] that is significantly faster than any previous algorithm, running in $O(n^2)$ time to find all the eigenvalues and all the eigenvectors of an n-by-n symmetric tridiagonal matrix. The core of the new algorithm is in subroutine xLAR1V, which computes the so-called *stationary qd transform* of a tridiagonal T in factored form $T = LDL^T$, producing a factored form of $T - \sigma I = L_+ D_+ L_+^T$, where $\sigma > 0$ is a shift, D and D_+ are diagonal, and L and L_+ are unit lower triangular and bidiagonal (i.e. L and L_+ have ones on the diagonal, nonzeros directly below the diagonal, and are zero elsewhere). The algorithm is

```
$s = -\sigma$
for $i$=1 to $n - 1$
    $D_+(i) = D(i) + s$
    $L_+(i+1, i) = L(i+1, i) * D(i)/D_+(i)$
    $s = s * L_+(i+1, i) * L(i+1, i) - \sigma$
end for
$D_+(n) = D(n) + s$
```

If the final value of s is Not-a-Number, which may occur if σ is larger than the smallest eigenvalue of T, this is detected (by asking if not($s > 0$ or $s < 1$)) and a slower version of the above loop is run with branches to avoid creation of $\pm\infty$ or NaN. The same idea is used to speed up the computation of eigenvalues the *differential qd transform* in LAPACK subroutine xLASQ3. The speedup of the overall routine for finding all the eigenvalues of T by exploiting arithmetic with NaNs and infinities this way ranges from 1.28 times faster on a Sun Ultra 30 to 1.8 times faster on an IBM RS-6000/590 [43]. This does not require the full power of **Feature 5: Exception handling**, just IEEE default arithmetic [3, 4].

Similarly, ScaLAPACK routine PDLAIECT assumes the availability of arithmetic with $\pm\infty$ (like $1/\infty = 0$) to remove branches from the inner loop in the computation of eigenvalues, and so go faster. The simplest version of the inner loop in PDLAIECT is as follows:

Count the number of eigenvalues less than x of the symmetric tridiagonal matrix T with diagonal entries $a(1 : n)$ and off diagonals $b(1 : n - 1)$ (we assume $b(0) = 0$).

```
$count = 0$
$d(0) = 1$
for $i = 1$ to $n$
        $d(i) = a(i) - x - b(i-1)^2/d(i-1)$
        if $d(i) < 0$, $count = count + 1$
```

In practice $d(i)$ overwrites $d(i-1)$, and we replace the inner loop by $d = a(i) - x - b(i-1)^2/d$. The trouble with this is that if d is small or zero, overflow occurs. The standard fix is to test to see if d is too small, and explicitly set it to a tiny safe value if it is:

```
for $i = 1$ to $n$
        $d = a(i) - x - b(i-1)^2/d$
        if ($|d| < too\_small$), $d = -too\_small$
if $d < 0$, $count = count + 1$
```

A faster way used by PDLAIECT if the user is on an machine implementing IEEE infinity arithmetic (i.e. most of the time) is to go ahead and compute an infinite d, since the next time through the loop $b(i-1)^2/d$ will be zero, and the recurrence will continue uneventfully. This also requires updating count slightly differently, to account for -0s. Speedups reported in [15] range from 1.18 to 1.47. This also does not require the full power of **Feature 5: Exception handling**, just the IEEE defaults [3, 4].

```
for $i = 1$ to $n$
        $d = a(i) - x - b(i-1)^2/d$
        $count = count + \text{signbit}(d)$
```

The above loop requires neither extra precision nor wider range, and is in fact part of ScaLAPACK [8].

One can further accelerate this recurrence by reorganizing it to remove the division, which can cost 6 times or more than the other floating point operations. This loop looks like

```
for $i = 1$ to $n$
        $p(i) = (a(i) - x) \cdot p(i-1) - b(i-1)^2 \cdot p(i-2)$
        if ($p(i)$ and $p(i-1)$ have opposite signs), $count = count + 1$
```

The relationship between these loops is that $d(i) = p(i)/p(i-1)$. Again we can remove the subscript on $p(i)$ if we use two temporaries:

```
for $i = 1$ to $n$ step 2
        $p = (a(i) - x) \cdot p1 - b(i-1)^2 \cdot p2$
        if ($p$ and $p1$ have opposite signs), $count = count + 1$
        $p1 = (a(i+1) - x) \cdot p - b(i)^2 \cdot p1$
        if ($p$ and $p1$ have opposite signs) $count = count + 1$
        $p2 = p$
```

This loop has no division, and so may be much faster than the first one. Not all machines have such slow division, so this will be machine dependent. However this loop is much more susceptible to over/underflow, because $p(i)$ is the determinant of the leading i-by-i submatrix of $T - x \cdot I$, and determinants easily over/underflow. For example $\det(x \cdot eye(n)) = x^n$ in Matlab notation, and neither x or n has to be very large for overflow to occur. We again have two approaches as in Section 2.7: First we can use **Feature 4: Extra-wide variables** which requires **Feature 3: Wider internal range** to implement the loop; for example, using 80-bit arithmetic will let us reduce the frequency of scaling tests in the inner loop to one every 128 iterations with IEEE single inputs or 16 with IEEE double. Second, we can use **Feature 5: Exception handling** to decide when overflow has occurred and (partial) recomputation with scaling is necessary (this may be used in conjunction with **Feature 4: Extra-wide variables** to reduce the incidence of exceptions).

There are various ways to accelerate bisection by using Newton's method or related ideas, but all involve inner loops similar to the one above.

Since PDLAIECT is not a BLAS routine, there is no way to hide the use of **Feature 4: Extra wide variables** or **Feature 5: Exception handling**.

2.9 Example 8: Cheap Error Bounds

A simple and attractive, if not always rigorously justified, way to get error bounds is to do an entire computation in one precision, say single, and then do it again entirely in a high precision, say double. For simplicity, suppose the true result of subroutine *solve* is the scalar x, and call the single precision scalar answer x_S and the double precision scalar answer x_D. Then the simple error bound for the error in x_S is simply $|x_S - x_D|$. The justification is that if the algorithm is *backward stable*, then x_S can be expressed as $x_S = x + \kappa \cdot \varepsilon_S$, where κ is the unknown condition number and ε_S is at most single-precision machine epsilon in magnitude. Similarly, $x_D = x + \kappa \cdot \varepsilon_D$. Then $|x_S - x_D| = |\kappa(\varepsilon_S - \varepsilon_D)| \approx |\kappa\varepsilon_S|$, the error in x_S. The error in x_D is roughly $\varepsilon_D/\varepsilon_S$ times smaller. There are a number of ways these estimates can be too large or too small, but they are better than nothing.

When x_D cannot be computed entirely in double precision, but perhaps only with some parts in double (like the internal precision of the BLAS), then the reliability of $|x_S - x_D|$ as an error bound decreases, but it still has some value.

This makes it attractive to write subroutine *solve* for computing x with the precision (S or D) as an input parameter. This means just one version of *solve* would have to be written, and the decision to use single or double precision would be made at run-time. Given the way variables are declared in languages like Fortran and C, we cannot change their types from single to double at run-time, but we can hope to change **Feature 2: Extra internal precision**, i.e. how much internal precision is used within the BLAS. In other words, we could pass in a variable PREC which might equal Single or Double, and in turn pass this as a run-time argument to the BLAS. The alternative, where the internal precision is determined at compile time, would mean that we would need (at least) two versions of *solve*, or code within *solve* that branches based on PREC and calls different BLAS routines. This would make *solve* unpleasant to write, read and maintain.

3 Implementation Techniques for Extended Precision

3.1 Introduction

Described in this section are most of the diverse schemes by which hardware, compilers and languages may support the five arithmetic features listed in Section 2.1. The schemes discussed are just the ones deemed most likely to support extended precision BLAS. After a general discussion and history of these techniques, we discuss the implementation used in our reference implementation in more detail.

Almost all computer architectures and the programming languages available for them support more than one floating-point format. The two formats supported practically universally are 32-bit = 4-byte single precision and 64-bit = 8-byte double. The CRAY X-MP, ..., J90 family constitutes the only exception at present and shall be discussed no further. Many architectures and some compilers support a third floating-point format wider than the first two. The basic kinds of extended (beyond double) precision formats that have been supported in hardware, firmware or software are named here in roughly descending order of speed. More details will be presented in Sections 3.2 and 3.3.

Double-extended. This is a conventional floating-point format with one field each of sign bit, exponent, and significand. IEEE Standard 754 requires this format to have at least 15 exponent bits and 64 bits of significand. These requirements are met (barely) by a 80-bit = 10-byte register on most Intel processors and their clones. These processors performs floating-point operations by default in these Double-extended registers even if they get rounded to narrower widths. This makes operations fast in so far as their execution times vary with operands' widths mainly in the times taken to move them through memory, not much in times taken by arithmetic. Except for 11 extra bits of precision, Double-extended arithmetic rounds in the same careful way as does double, so error-analysis changes solely because of a somewhat smaller roundoff threshold.

Double-double. Each 16-byte operand is the unevaluated sum of two 8-byte doubles of which the first consists of the "leading" digits, and the second the "trailing" digits, of the format's value. Its exponent range is almost the same as double's, differing only in some vagueness about underflow. Rounding is vague too; its precision is roughly twice double's, and its error-analysis rather rough (see section 3.3.2). Implementation is in software that may be hardware-dependent, though it can be made portable to all processors relevant to this document at the cost of some speed. This issue is important because Double-double is worth using only when it is noticeably faster than Quadruple, as can often occur if floating-point hardware capabilities are exploited fully. These implementation techniques can be extended further to tripled- and quadrupled-double, but with diminishing efficiency except in special cases [11, 45, 47]. These techniques have also been used to compute dot products ($\alpha \sum_i x_i y_i + \beta r$) with all correct leading digits, despite all roundoff [35, 9, 33]. We will not consider these very high precision implementations further here.

Quadruple precision. This is a conventional 128-bit = 16-byte wide floating-point format with one field each of sign bit, exponent, and significand, though perhaps stored in two 8-byte words with the second's exponent field unused. The precision is at least wide enough to hold exactly the product of two doubles. Nowadays Quadruple precision operations run at least several times slower than double but use the same careful algorithms for rounding, so error-analysis changes solely because of a much smaller roundoff threshold.

BigFloats. This is not really an extension of floating-point hardware; instead, arbitrarily high precision is accomplished by breaking every number into equal-width "big digits" each stored as an element of an array of integers or integer-valued doubles. This kind of software-simulated floating-point is built into automated algebra systems like Macsyma [28], Mathematica [51] and Maple [40], and provided to other programmers by packages like Richard Brent's MP [10] and David Bailey's MPFUN [5]. BigFloats are optimized for very high precision, and run so much slower than Double-double or Quadruple when delivering comparable precisions that BigFloats shall not be treated further in this document.

Unfortunately, our three names "Double-extended", "Quadruple precision" and "Double-double" need not correspond with the names programming languages use for precisions beyond double. For instance, "TEMPREAL", "REAL*10", "REAL*12" and "REAL*16" are the names used by the few Fortran compilers that support(ed) Double-extended. On the other hand, "REAL*16" in a Fortran compiler can mean any one of those three, and "long double" in a C or C++ compiler can mean any one of those three or just double precision. The names "EXTENDED" and "quadruple" have also been used by compilers written for hardware with built-in Quadruple precision even if it is vestigial, existing only as unimplemented op-codes that would trap to software if the compiler did not circumvent trap overheads by invoking the software directly. But by conforming to the minimum requirements of a standardized language like Fortran 77, many a compiler denies access to whatever extended precision does exist in the hardware or, worse, uses that extended precision whimsically to evaluate some floating-point subexpressions while disallowing mention of it, as used to happen on the Sun III.

Consequently, of the three kinds of extended precision named, only a sub-optimal implementation of Double-double can be made portable across all compiler and hardware boundaries.

To standardize extended precision BLAS we shall have to cope with diverse formats, implementations, speeds and linguistic impediments. Section 3.2 will present a little more history of the support (or lack of it) for extended precisions. Section 3.3 discusses design options for implementing double-double arithmetic, and finally section 3.3.2 discusses our implementation, which is based on the Fortran 95 double-double package [6].

3.2 A Little More History of Extended Formats

Among computers still (or once) commercially significant in the world of floating-point computation, many provide(d) more than nominal support for a third floating-point format wider than the nearly ubiquitous "REAL*4 (float)" and "REAL*8 (double)" formats.

Format Type	Computer Family	Bytes Width	Sig. Dec.	$\log_{10}$(Overflow)
Double-extended IEEE 754 binary 64 sig. bits	Intel x86/87 & Pentium PC & clones by AMD, Cyrix.	10	19 - 20	4932
	Motorola 680x0-based Apple Macintosh, NeXT, SunIII; Intel 80960KB.	12	19 - 20	4932
	HP/Intel IA-64 Itanium; Motorola 88110.	16	19 - 20	4932
Double-double 2 x 53 sig. bits	IBM RS/6000, Power PC, Apple Power Mac; HP 8000 & PA RISC 2.0; (HP/Intel IA64 Itanium) (IBM s/390 G5) (SGI MIPS)	16	grubby 32	308
Quadruple (Hex) 28 hex digits	IBM S/370, 3090, S/390.	16	31- 33	76
Quadruple (Binary) 113 sig. bits	DEC VAX "H", (Compaq Alpha)	16	33 - 34	4931
	IBM S/390 G5 ; (Sun SPARC, HP PA RISC, HP/Intel IA 64, SGI MIPS)	16	33 - 34	4932

Table 2: Higher-than-Double Precision Formats Available in Hardware.

The term "Extended" first came into widespread association with "Precision" in late 1967 when the IBM S/360 mod 85 came out with hardware to support addition, subtraction and multiplication (not division) of Quadruple precision hexadecimal floating-point operands. This hardware was IBM's response to NASA's expressed but misperceived need for precision beyond double during orbit calculations. IBM's hardware competed against Seymour Cray's CDC 6600 whose "single precision" word was 60 bits wide with 48 significant bits of precision; it also had three peculiar "DX" operations to speed up software-implemented "double precision" (actually "single-single") with almost 96 significant bits, much wider than IBM's S/360 double precision with 14 hex digits (about 53-56 significant bits). The mod 85's "Extended" format persists today, fully supported (including division) in IBM's S/390 hardware and compilers for it, stored as two 8-byte double precision hex floating-point words superficially resembling Double-double; however, the second word's sign and exponent are ignored during operations, and its significand may be unnormalized.

In the mid 1970s, to provide a functionality competitive with IBM's "Extended" for the DEC VAX a few years after its debut, DEC augmented its floating-point instruction set with the "H" format, a true Quadruple precision binary format with 113 significant bits of precision in a 16 byte word. DEC's compilers, plus an excellent mathematical library, supported this format fully though it ran "hot" (in hardware), "warm" (in microcode) or "cold" (in software) on various VAX models, and "cold" on the DEC (now Compaq) Alpha. For several years an almost identical de facto standard Quadruple precision format has existed in several microcomputer architectures that conform to IEEE Standard 754 for Binary Floating-Point [3]. This format has 15 exponent bits and 113 bits of significand in a 16 byte word, so it qualifies as an IEEE 754 Double-extended format with more than the minimum (64 bits) of required precision. At this time only the recent (1999) IBM S/390 G5 processor supports it fully in hardware; it is implemented in software on the HP PA RISC, Sun SPARC, SGI MIPS and HP/Intel IA 64 (Itanium) processors, and accessible through their respective manufacturers' compilers, though not always accompanied by a fully accurate mathematical library of elementary transcendental functions.

The history of extended precisions is littered with sad stories. One is the dearth of language support for extended precisions except in compilers produced under the aegis of hardware manufacturers, and the variability of that support where it existed; perhaps C99 [12] will remedy this deficiency. Another arises because so far, no matter whether "hot", "warm" or "cold", Quadruple has always run at least several times slower than double precision. This discourages programmers from exploiting Quadruple precision; and its consequently low duty cycle and absence from benchmarks have provided little incentive for hardware manufacturers to speed it up. In fact, sales of hardware that supports it have scarcely diminished whenever Quadruple precision has been made cheaper and slower. Meanwhile, it has been shunned for over three decades by expert developers of portable numerical software libraries like EISPACK [48], LINPACK [20], LAPACK [2] and ScaLAPACK [8].

A compromise, to offer programmers some benefits of extra precision and thus enhance the reliability of their numerical software, insinuated a few digits of extra precision into a few places in the hardware's architecture where performance would not be degraded too much if at all. This compromise appeared in the late 1960s in the accumulator register of GE 635 mainframes (subsequently taken over by Honeywell), then in the floating-point register file of Pr1me minis, then in the DEC VAX's EMOD instruction in the early 1970s, then in the internal registers (inaccessible to most users) of HP handheld calculators and decimal computers after 1975. These extra digits were at first intended for use solely in assembly language or microprogramming, so they did not appear in higher-level languages. This linguistic mindset contributed to more sad stories later.

In 1977 the 8087 floating-point coprocessor for Intel's 8086/88 processors was being designed to serve a market bigger by orders of magnitude than the computing industry had served before. This vast market's average level of numerical expertise was expected to be lower than before so, for the sake of enhanced reliability, the 8087 was designed to support three floating-point formats: 4-byte single precision, 8-byte double precision, and 10-byte Double-extended precision with three more exponent bits and eleven more bits of precision than double. Most important, all the 8087's registers had this extended width, and all operations were carried out to this width regardless of the operands' precisions, much as the original Kernighan-Ritchie C on DEC PDP-11s had evaluated all floating-point subexpressions in double regardless of declared variables' precisions. In short, applications programmers were intended to use the 8087's Double-extended precision (Intel called it "TEMPREAL") for all local variables and intermediate expressions except perhaps large arrays, since the use of narrower precisions could save appreciable time only while they moved through the memory system.

By 1981 a proposed IEEE Standard 754 had grown out of the 8087's design, and its three formats' rationale and intent had been adopted at three more companies: Motorola was working on the 68881 floating-point coprocessor for its 68020, Zilog on the Z8070 for its Z8000; and Apple had incorporated all three formats into its Standard Apple Numeric Environment (SANE) first for the Apple III's Pascal and then for the 680x0-based Macintosh and retrofitted for the Apple II. With their eyes on that vast mass market, these three companies designed the Double extended format to be the default for all floating-point operations. However, there was another market, the one then served by mainframes and minicomputers, to which upscale workstations could be sold with higher speeds and markups than were contemplated for the mass market of Personal Computers. Workstation users were expected to recompile many of their mainframe applications, which hardly ever mentioned Extended. Since they cared about speed to the near exclusion of everything else, the draft of IEEE 754 left the Double extended format to the implementor's option. By 1982-3 implementations of IEEE 754 without that option were under way at National Semiconductor, Weitek, Sun SPARC, MIPS (now SGI), and IBM, whose RS/6000 processor developed in Austin TX evolved into the Power PC now produced also by Motorola and used in Macs.

By 1985, when IEEE 754 was promulgated as an official standard, two versions of it had become de facto standards for most new computer hardware. All three of its floating-point formats were supported by a few numerically superb compilers for SANE on Apple Macintoshes; but most compiler makers continued to use front ends that lacked names for a third floating-point format, so they catered to only the minimal version of IEEE 754 with just single and double.

Short-term marketing priorities overruled long-term mathematical considerations. In 1980, for instance, Microsoft was implementing compilers for the forthcoming IBM PC, which had an Intel 8088 processor and a socket for the 8087. Microsoft wrongly prophesied emptyness for almost all of those sockets, so its compilers lacked support for Double extended arithmetic which, in software, would have run somewhat slower than double. Later, when Borland came out with Pascal and C compilers that supported all three floating-point types, Microsoft followed suit for a decade until, by the late 1990s, its dominance of the market let it drop support for Double extended from its compilers run on Windows NT, which was implemented originally on DEC Alphas whose IEEE 754 arithmetic had only single and double.

Apple's alliance with IBM in the late 1980s, entailed switching Macintoshes from the 680x0 to IBM's Power PC. Because it supported only single and double precisions, this choice of processor scuttled SANE. It need not have happened that way; other fast RISC processors like Intel's 80960KB and Motorola's 88110 could have sustained SANE. But floating-point did not figure in Apple's marketing plans.

Currently 10-byte Double-extended is practically ubiquitous in the hardware of Intel x86/87 and Pentium processors and their clones by AMD and Cyrix. The width of this format may be padded by extra bytes to avoid memory alignment delays and allow for future growth to a wider significand if and when market conditions demand it; the total padded width is 12 bytes for Motorola 680x0/68881/2 and Intel 80960KB, 16 bytes for Motorola 88110 and HP/Intel IA 64 (Itanium) processors. All these processors can evaluate floating-point operations by default in Double-extended registers, even if they get rounded to narrower widths; their execution times vary with operands' widths mainly in the times taken to move them through memory, not much in times taken by arithmetic. However, even if some Microsoft compilers use it surreptitiously, they stopped supporting Double-extended when Windows NT came out. Java does not support it at all, and no commercially significant Fortran compilers support it. Maybe things will change with the emergence of C99 and Itanium.

The first and easiest way for BLAS to exploit Double-extended on hardware that has it is to use it to accumulate scalar products and matrix products of double precision data. There is ample precedent for extra-precise accumulation; products of single precision matrices used to be accumulated in double precision routinely until the mid 1960s when doing so became slow on CDC mainframes and inconvenient on IBM mainframes because standard Fortran semantics had been changed. The superior quality of results obtained by LAPACK and Matlab for large dimensional and ill-conditioned problems would distinguish hardware capable of extra-precise accumulation advantageously from others [30].

Versions of Matlab that ran on old 680x0-based Apple Macintoshes, and old versions of Matlab running on Wintel PCs, delivered matrix products more accurate thereon than Matlab used to deliver on upscale Sun SPARCs and SGI MIPS, or delivers now on all computers. Mathematica's N[...] operator delivered three more decimal digits' accuracy on old Macs and NeXTs than on all other computers. Should ostensibly portable software be allowed to get better results on some hardware than on others? The answer "Yes" seems obvious to the owners of the better hardware, but not so clear to software developers who yearn for Java's promise of identical floating-point results for everybody everywhere even if it is a pipe-dream. Each variation multiplies the cost of software testing and validation. On the other hand, "no variation" implies "no change", which stifles evolutionary progress leaving no alternative but revolution. Voltaire put the conflict succinctly: "The Better is the enemy of the Good." It will not be resolved here.

The tension between knowledgeable purchasers of computing platforms (hardware and operating systems) on the one hand, and applications software vendors on the other hand, poses dilemmas for platform vendors whose different product lines offer different floating-point qualities. Here are some of the vendors who must cope with invidious comparisons and conflicting loyalties within their own houses:

Vendor	Line(s) with Double-extended	Line(s) without
Apple	Old 680x0-based Macintosh	Power Mac, iMac
Compaq	Wintel PC	DEC Alpha
HP	Wintel PC, IA-64	HPUX on PA RISC
IBM	Wintel PC	Power PC, RS/6000
SGI	Wintel PC	UNIX on MIPS R-x000
Sun	Solaris on Intel-based PC	Solaris on SPARC
Microsoft	Windows 9x on PC	Windows NT on DEC Alpha

Table 3: Conflicting product lines with and without Double-extended.

In Windows NT on a PC the conflict was resolved highhandedly by inhibiting the PC's Double-extended to match the DEC Alpha's arithmetic limitations. Java seems to be attempting the same for the sake of Sun's SPARC. Other vendors may have to exercise greater sensitivity to market conditions. The outcomes of these conflicts will not be predicted here except for the wry observation that, by itself, technical superiority cannot ensure market success despite what R.W. Emerson may have said about a better mousetrap.

Finally, we mention that G. Henry built a comprehensive suite of scalar extended precision operations for ASCI Red, which consists of Pentium Pro processors running Linux [26]. It includes object code in ELF format, and so is callable from C or Fortran on Linux-based Intel-inside workstations, and perhaps Solaris X86, though this remains to be determined. A typical routine is xadd_YYY_(a,b,c), which computes $a = b + c$. Each Y indicates the type of a (first Y), b (second Y) or c (third Y), which can independently be single ("s", 32 bit), double ("d", 64 bit), extended ("x", 80 bit), or even double-double ("w", 128 bit) and double-extended–double-extended ("q", 160 bits) (discussed in the next section). In other words, there are $5^3 = 125$ add functions. Other operations are subtraction, multiplication, mul-add, mul-subtract, sqrt, assignment (conversion), and reciprocation. There is also a dot product that takes double precision input vectors and accumulates the dot product in extended. Finally, there are routines to change the rounding mode to 64-bits or 80-bits, and to print a number. There are over 700 routines, which were produced with the help of an automatic generation tool. This shows that an implementation that completely supports all kinds of mixed precision operations will be very large, and require automation to build and support.

3.3 More about Double-Double

The 16-byte Double-double format can be implemented in portable software to run on all machines that conform to IEEE 754 and on quite a few that don't, and runs about an order of magnitude slower than double but usually considerably faster than Quadruple. Double-double and its variations have an extensive history, with implementation ideas going back to the 1960s [44, 13, 29, 39, 45, 47, 11, 26, 6]. See especially [45] for a history. Our implementation is based on Bailey's [6], the main routines of which are shown in the Figures 5 through 7 below. For the routines with other combinations of inputs and outputs, see http://www.nersc.gov/∼xiaoye/XBLAS/.

The addition, multiplication and division algorithms in figures 5, 6 and 7, require 20, 25 and 32 floating-point operations, respectively.

Double-double can be sped up roughly twofold if advantage is taken of a Fused Multiply-Accumulate (FMAC) operation on hardware that has it. This operation computes expressions like $X*Y+Z$, $X*Y-Z$ and $Z-X*Y$ with just one rounding error at the end instead of two rounding errors, one after multiplication and another after addition. (Do not confuse the FMAC with the unfused MAC operation that computes $X*Y+Z$ in one instruction but with two rounding errors, thus saving only time.)

The FMAC appeared first in 1984 on the IBM RS/6000, whose reduced instruction set designed two decades earlier by Dr. John Cocke had only three floating-point operations: conversions between single and double precision, and a MAC. This MAC was upgraded to an FMAC when it was found necessary for a software implementation of division correctly rounded at a tolerable speed.

Among many other uses, the FMAC accelerates Double-double by computing exactly a product $X*Y$ of two doubles in just two operations, thus drastically shortening algorithm ddmuld in figure 6:

$P := X*Y+0$ rounded to double;

```
void ddadd(double dda[2], double ddb[2], double ddc[2])
{
    /* Compute double-double = double-double + double-double
       (ddc = dda + ddb). */
    /* dda[0] represents high-order word, dda[1] represents low-order word. */
    double bv;
    double s1, s2, t1, t2;

    /* Add two high-order words. */
    s1 = dda[0] + ddb[0];
    bv = s1 - dda[0];
    s2 = ((ddb[0] - bv) + (dda[0] - (s1 - bv)));

    /* Add two low-order words. */
    t1 = dda[1] + ddb[1];
    bv = t1 - dda[1];
    t2 = ((ddb[1] - bv) + (dda[1] - (t1 - bv)));

    s2 += t1;

    /* Renormalize (s1, s2)  to  (t1, s2) */
    t1 = s1 + s2;
    s2 = s2 - (t1 - s1);

    t2 += s2;

    /* Renormalize (t1, t2)  */
    ddc[0] = t1 + t2;
    ddc[0] = t2 - (ddc[0] - t1);
}
```

Figure 5: Adding two double-doubles

```
void ddmuld(double dda[2], double db, double ddc[2])
{
    /* Compute double-double = double-double * double (ddc = dda * db). */
    double a11, a21, b1, b2, c11, c21, c2, con, e, t1, t2;
    double split = 2^27 + 1;

    /* Split dda[0] and db into two parts with at most 26 bits each,
       using the Dekker-Veltkamp method. */
    con = dda[0] * split;
    a11 = con - dda[0];
    a11 = con - a11;
    a21 = dda[0] - a11;
    con = db * split;
    b1 = con - db;
    b1 = con - b1;
    b2 = db - b1;

    /* Compute dda[0] * db using Dekker's method. */
    c11 = dda[0] * db;
    c21 = (((a11 * b1 - c11) + a11 * b2) + a21 * b1) + a21 * b2;

    c2 = dda[1] * db;

    /* Compute (c11, c21) + c2. */
    t1 = c11 + c2;
    t2 = (c2 - (t1 - c11)) + c21;

    /* Renormalize (t1, t2)  */
    ddc[0] = t1 + t2;
    ddc[1] = t2 - (ddc[0] - t1);
}
```

Figure 6: Multiplying a double and a double-double

```
void dddivd(double dda[2], double db, double ddc[2])
{
    /* Compute double-double = double-double / double (ddc = dda / db),
       using a long division scheme. */
    double b1, b2, con, e, t1, t2, t11, t21, t12, t22;
    double split = 2^27 + 1;

    /* Compute a DP approximation to the quotient. */
    t1 = dda[0] / db;

    /* Split t1 and b into two parts with at most 26 bits each,
       using the Dekker-Veltkamp method. */
    con = t1 * split;
    t11 = con - (con - t1);
    t21 = t1 - t11;
    con = db * split;
    b1 = con - (con - db);
    b2 = db - b1;

    /* Compute t1 * b using Dekker's method. */
    t12 = t1 * db;
    t22 = (((t11 * b1 - t12) + t11 * b2) + t21 * b1) + t21 * b2;

    /* Compute dda - (t12, t22) using Knuth trick. */
    t11 = dda[0] - t12;
    e = t11 - dda[0];
    t21 = ((-t12 - e) + (dda[0] - (t11 - e))) + dda[1] - t22;

    /* Compute high-order word of (t11, t21) and divide by b. */
    t2 = (t11 + t21) / db;

    /* Renormalize (t1, t2). */
    ddc[0] = t1 + t2;
    ddc[1] = t2 - (ddc[0] - t1);
}
```

Figure 7: Dividing a double-double by a double

$Q := X * Y - P$ exactly as a double thanks to the FMAC.

Now the product of X and Y equals the sum of doubles P and Q exactly although rounding that sum to double would yield just P . This kind of unevaluated sum of a leading and a trailing double is the way every Double-double is represented by a pair of doubles. To compute P and Q without an FMAC would cost six conventional multiplications plus several additions or subtractions.

The FMAC is built into the hardware of IBM's RS/6000 and, for compatibility with it, into the S/390 G5 processor as well as the IBM/Motorola Power PC used in Apple's Power Macs. The same goes for HP's 8000 and PA RISC 2.0, and for the SGI MIPS R-8000. At present C99 [12] is the only higher-level programming language that offers an explicit invocation $fma(x, y, z) = x * y + z$ of the FMAC. Language support for the use of FMACs by default on hardware that has it poses problems partly because of ambiguity (how should "$X * Y + A * B$" be evaluated?) and partly because indiscriminate use can, in very rare instances, break programs that would have worked well with an unfused MAC. Because of these rare instances, Matlab eschews the FMAC. Worse, the MIPS R-10000 uses the same opcode for its unfused MAC as the R-8000 used for its FMAC, thus dropping conundrums in the way of language support. The HP/Intel IA 64 Itanium has a Double-extended FMAC which can serve Double-double and Double-double-extended with the same algorithms, and perhaps serve software-implemented Quadruple too.

Here are some other double-double implementation efforts.

Keith Briggs released a C++ class library implementing a `doubledouble` type [11]. It lets users declare double-double variables and operate on them using standard infix operators via operator overloading. It supports the usual four arithmetic operations, sqrt, transcendentals, and I/O. In other words, it essentially makes double-double a first class floating point data type, so that one could use them internally to the BLAS or externally. It runs on the X86 architecture with Linux and gcc, X86 with Windows 95 and Microsoft C++, MIPS with IRIX and egcs, and SPARC with Solaris and egcs.

David Bailey released a Fortran 95 double-double package [6]. It does not introduce a new data type for double-double variable, but just represent it using a pair of `REAL*8` numbers. It supports the usual arithmetic operations and transcendental functions. It is portable on all the machines with IEEE floating-point arithmetic. In fact, our double-double implementation and basic algorithms are very similar to Bailey's, modified as described in the next section.

Finally, G. Henry's package mentioned above [26] includes not just double-extended operations, but double-double and double-extended–double-extended (i.e. 160-bits, with 128 fraction bits).

3.3.1 Why Double-double is an attractive nuisance except for the BLAS

Currently, machines with an FMAC run Double-double a few times faster than correctly rounded Quadruple implemented in software. This speed advantage turns Double-double into an "attractive nuisance", like an unfenced backyard swimming pool. The trouble grows out of the diverse ways Double-double can be rounded, none of them strictly correct, as will be explained further below. An outline of the trouble, which only occurs trying to use double-double more widely than inside the BLAS, goes like this:

Consider a Pascal, Fortran or C Function $F(X)$ written "polymorphically" in four versions, the first using exclusively small integer constants and REAL*4 (float) variables, the second the same except for replacement of every REAL*4 (float) by REAL*8 (double), the third the same but for REAL*10 (Double-extended or TEMPREAL), and the fourth the same but for REAL*16 (long double). This program is intended to approximate some ideal real function $f(x)$ that can be inferred from $F(X)$ by substituting "INFINITE PRECISION" for every "REAL*..." declaration and attending to a few technical details concerning stopping criteria for iterations. For instance, the rounding error threshold EPS/2 should be gauged from statements like

```
    $T := 3$ ;
    $EPS := ABS(1 + 3 * (1 - 4/T))$ ;
```

(only the quotient should be inexact); and convergent iterations that would take infinitely many repetitions to achieve infinite precision should be assumed to do so instantly. If each of the first three versions of $F(X)$ matches $f(x)$ within a few units in the last place of the precision declared for $F(X)$, then we may well expect the same for the fourth version, and this expectation is fully justified if REAL*16 (long double) compiles to correctly rounded Quadruple precision or even just Double-extended. But not if it compiles to Double-double; then program F(X) can malfunction arbitrarily badly.

These malfunctions are not mere theoretical possibilities; they do happen, rarely but not rarely enough to ignore, on IBM's and Apple's Power PCs, whose compilers support Double-double fully as well as IEEE 754 single and double precisions. Consequently there is some non-negligible risk of malfunction if a program recompiled for those machines was imported from machines, like IBM's S/390 G5, on which "REAL*16" means Quadruple precision, or from machines like old Macs on which "long double" meant Double-extended. The risk for BLAS is probably negligible, arising perhaps from none but contrived situations; still, the risk deserves some discussion.

The quantity EPS computed above is the gap between 1.0 and the next bigger floating-point number in the format to which 4/T was rounded unless this format is Double-double, in which case the gap is far tinier. When computed in Double-IEEE-754-double, EPS comes out to 0.5^{106} provided division and subtraction are rounded with care; but $1.0 + 0.5^{1022}$ is a Double-double number much closer to 1.0. More generally, most Double-double precision numbers X are separated from their neighbors by gaps of the order of X*EPS ; but around double precision numbers the Double-doubles are much denser, some separated by gaps no bigger than the smallest positive double, 0.5^{1074} for Double-IEEE-754-double. Consequently a rounding error in Double-double can vastly exceed the gaps between adjacent numbers. This disparity breaks the logic and validity of some error- analyses valid for conventional floating-point.

Here is an example. A common way to approximate $cosh(x)$ is to first compute $Z = EXP(ABS(X))$ (rounded to roughly working precision) and then $COSH(X) = (Z+1/Z)*0.5$. Let's ignore overflow. Then $COSH(X)$ may well be less accurate than correctly rounded, yet it is provably monotonic in this sense: $COSH(X) > COSH(Y)$ implies $ABS(X) > ABS(Y)$. This is provable for every binary floating-point format conforming to IEEE 754/854. But not for Double-double; no matter how carefully it is implemented, it almost certainly violates that monotonicity relationship for some smallish arguments X and Y .

Here is another example. Suppose the iteration $x \rightarrow f(x)$ converges at a tolerable rate in a monotonically decreasing way to a finite limit L. Then the loop

```
    $X :=$ initial guess, certainly too big ;
    $Y :=$ anything bigger than $X$ ;
    WHILE $X < Y$ DO
        $Y = X$ ;
        $X = F(Y)$ ;
    END WHILE
    $L := Y$ ;
```

is a reasonable way to compute L as fast as possible about as accurately as the arithmetic allows. But not in Double-double. If unluckily the limit L is representable exactly in double precision, then the sequence of iterates X can look like

$$L + 0.5^{100}, L + 0.5^{101}, L + 0.5^{102}, ..., L + 0.5^{1074}$$

before it stops. That takes far longer than may have been expected. Worse, a loop not so different from this one is often used to reveal at run-time a conventional arithmetic's rounding error threshold EPS/2 by seeking the smallest increment D that increases the computed value of 1+D . For Double-double such a loop sets D near the underflow threshold instead of the rounding error threshold. (To get a correct result, substitute "4.0/3.0 rounded to Double-double" for "1" .)

Hazards like these examples are unlikely to afflict the BLAS though eigensystem calculation may be affected elsewhere, and so may some computations with special structured matrices like Cauchy matrices.

On all machines that conform to IEEE 754, Double-double can be implemented in a portable way that would give exactly reproducible results, but this is unlikely to persist for long for two reasons: First, an implementation optimal for one architecture may well be very suboptimal for another, especially if the latter has an FMAC the former lacks. Further differences arise if some machines tolerate testing and branching much better than others that run faster if extra arithmetic operations are used in place of tests and branches. Different optimized implementations can give different but roughly equally valid results on different machines.

Secondly, implementing Double-double differently for the BLAS than elsewhere is worth considering. For example, Double-double's roundoff threshold $EPS/2$ is roughly 0.5^{107} for IEEE 754 double, so one might reasonably expects a difference $X - Y$ of Double-double arguments to be computed with an error no worse than something like $EPS * |X - Y|$. Doing so takes noticeably longer than if instead an error no worse than roughly $EPS * (|X| + |Y|)$ were tolerable, as is almost surely the case for matrix multiplication. Thus, Double-double can run sloppier and faster inside the BLAS than outside them. Of course, if it is sloppy outside too then bad things may well happen faster, starting with the failure of the short formula above for EPS.

In short, Double-double is an attractive nuisance outside the BLAS but very worthwhile inside them.

3.3.2 Performance of our double-double implementation

Although the double-double arithmetic operations require many more floating-point operations, these operations are mostly performed at the register level without incurring much more memory traffic. The overall speed of an extra precision BLAS routine is not much slower. To confirm this, we timed several versions of GEMV and GEMM routines on a Sun UltraSPARC, with 333 MHz clock rate and 2 Mbytes secondary data cache. The peak Megaflop rate is 666 Mflops. The performance results are presented in Figures 8 and 9. The routines we have timed are the following:

1. `c_dgemv (c_dgemm)`: our GEMV (GEMM) with double precision for everything,

2. `c_dgemv_x (c_dgemm_x)`: our GEMV (GEMM) with double precision input/output, and double-double internal precision,
3. `f_dgemv (f_dgemm)`: the Fortran 77 double precision GEMV (GEMM) from Netlib, and
4. `sun_dgemv (sun_dgemm)`: the double precision GEMV (GEMM) from the Sun Performance Library.

Without extra precision, our GEMV and GEMM run at about the same speed as the Fortran versions. The Sun Performance Library version of GEMM can be much faster than ours and the Fortran version for large matrices; this is because we do not perform optimizations such as blocking to improve data locality.

For the extra precision routines, when the matrices are small and all fit in cache, we observed about a factor of 9 to 10 slowdown. When the matrices are large and do not fit in cache, we observed only about a factor of 3.6 slowdown, see Figures 8(b) and 9(b).

We also plot Megaflop rates in Figures 8(c) and 9(c). The flop counts for the normal GEMV and GEMM are $n(2n+3)$ and $n^2(2n+3)$, and for the extra precision versions are $n(37n+62)$ and $n^2(37n+62)$. The extra precision routines achieve a sustained performance of 270 to 300 Mflops for both GEMV and GEMM, showing excellent data reuse even without any algorithmic blocking.

Indeed, the Megaflop rates for the extended precision codes are so high that it is doubtful whether further tuning would make a significant impact on their usefulness.

For a standard model of floating-point arithmetic

$$fl(a \odot b) = (a \odot b)(1+\delta) \quad \text{in the absence of over/underflow,} \tag{1}$$

where $\odot \in \{+,-,\times,/\}$, we can show that the algorithms in Figures 5 through 7 yield $|\delta|$ bounded by $2 \cdot 2^{-106}$, $4 \cdot 2^{-106}$ and $4 \cdot 2^{-106}$, respectively. This is more than adequate to satisfy the error bounds described in section 6.

We note that there is a more relaxed model of floating-point arithmetic for addition and subtraction

$$fl(a \pm b) = a(1+\delta_1) \pm b(1+\delta_2)$$

satisfied by machines without guard digits, like old Crays, which can be implemented in double-double more efficiently than the stricter model 1. In this relaxed model, the double-double addition (`ddadd`) algorithm can be considerably simplified. For example, Bailey's implementation of addition [6] satisfies this model, with $|\delta_1|, |\delta_2|$ bounded by 2^{-106}. His algorithm requires only 11 floating-point operations as opposed to 20 for ours. We also measured the speed of a few BLAS routines using Bailey's double-double algorithms. For example, GEMV runs about 35 – 40% faster. For the sake of speed, BLAS implementors may feel free to use this relaxed model, since it is still adequate to meet the error bounds described in section 6.

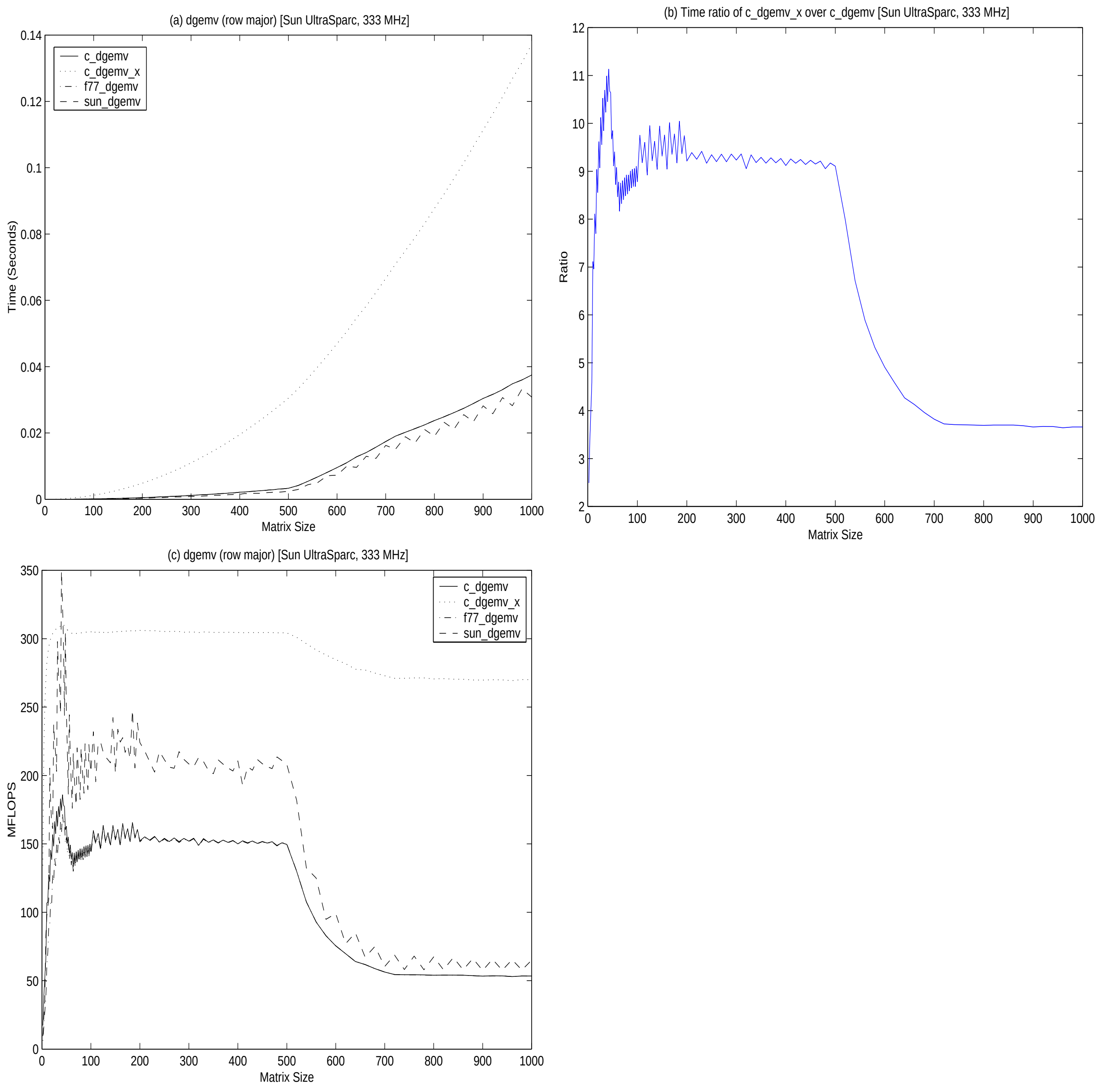

Figure 8: GEMV performance.

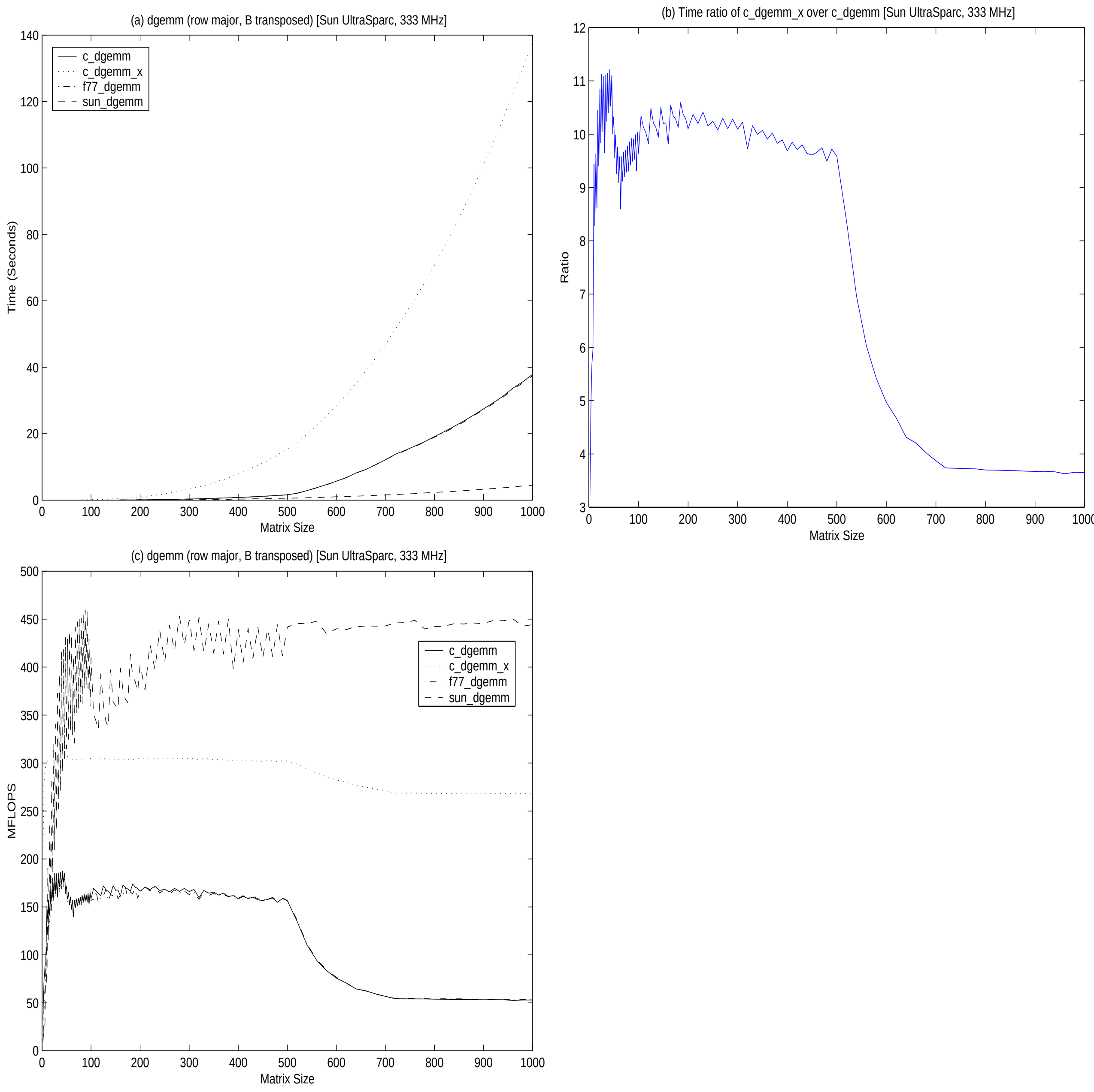

Figure 9: GEMM performance.

4 Summary of Design Decisions

The list of examples in Section 2 was meant to illustrate the opportunities of the five arithmetic features listed in Section 2.1, and the discussion in Section 3 was meant to illustrate their costs. In this section we weigh the costs and benefits of these features, and present our design decisions. In particular, our design should satisfy the following goals:

Goal 1: be reasonable to implement,

Goal 2: support some if not all important application examples,

Goal 3: be easy to use,

Goal 4: encourage the writing of portable code, and

Goal 5: accommodate growth as we learn about new algorithms exploiting our arithmetic features.

Here are our decisions.

1. We do not require that the user explicitly declare or use any new extended precision data types, i.e. we do not use **Feature 4: Extra-wide variables**. Since these are not supported in a standard way by every language and compiler, we cannot reasonably mandate them here. If we made up conventions (like double-double variables occupying a pair of consecutive double precision words in Fortran 77) then users would expect us to propose subroutines for all the other operations they might want using these variables (binary arithmetic operations, comparisons, binary-decimal conversion, etc.), which is definitely beyond the scope of this committee (violating Goal 1). Furthermore, in languages like Fortran 77 without "data hiding" mechanisms, users might have to modify their data structures significantly and replace all infix operators like $+$ and $\times$ with subroutine calls (violating Goal 3). Thus the only extended precision that we mandate is hidden inside the BLAS, and so can be implemented in any convenient machine dependent way.

2. The above decision mostly satisfies Goal 2, since many of the examples in Section 2 can be implemented without Feature 4. Nonetheless, those examples requiring Feature 4 could be supported when the working precision is single and the extended precision is double. This would violates the long-held goal in libraries like LAPACK to be able to run essentially the same code in all available precisions, creating all versions automatically from one single precision template. But it is worth it for the benefits in single.

3. Since we cannot predict all the future applications of extended or mixed precision, nor can we predict what new class libraries or modules will become available to support new data types (e.g. see Section 3.3), it is important to accommodate growth (Goal 5). We do this by making our design as orthogonal as possible to the rest of the BLAS Standard, showing how to take any BLAS routine (including future ones with, say, quadruple precision inputs and outputs), determine whether extra precision is worth using or not, and define the extended precision version if it is.

   In particular, we believe that extra precision is *not* worth using unless (1) there is the possibility of cancellation between non-input quantities in the algorithm, and (2) one is not applying orthogonal transformations. The first condition eliminates scaling a matrix by a constant, computing norms of matrices and vectors, and adding two arrays, since in all these cases the result is already good to high relative accuracy without extra internal precision. In fact, adding two numbers to higher precision and then rounding their sum to the lower output precision can yield a very slightly less accurate answer than doing everything in the output precision. The second condition reflects the fact that the orthogonal transformations have already been rounded to working precision, and this does about as much damage to the accuracy as roundoff in their subsequent application in working precision to another vector or matrix, so it is not worth making this second step more accurate.

4. Since the number of possible routines with all combinations of mixed precision inputs is very large, and includes combinations unlikely to be used in practice (like multiplying a single-precision-real matrix by a double-precision-complex matrix) we specify a small subset of routines that seem to cover most foreseeable needs. In particular, we allow only certain combinations of single real/double real, of single complex/double complex, of single real/single complex, and of double real/double complex. Furthermore, we limit all scalars and output variables to be the "higher" of the two precisions. For example, for matrix-matrix multiplication $C = \alpha \cdot A \cdot B + \beta \cdot C$, the following 12 mixed precisions and types are allowed in addition to the 4 unmixed precision versions (where all arguments are S = single precision real, D = double precision real, C = single precision complex, or Z = double precision complex):

α	A	B	β	C
D	S	S	D	D
D	S	D	D	D
D	D	S	D	D
Z	C	C	Z	Z
Z	C	Z	Z	Z
Z	Z	C	Z	Z
C	S	S	C	C
C	S	C	C	C
C	C	S	C	C
Z	D	D	Z	Z
Z	D	Z	Z	Z
Z	Z	D	Z	Z

In other words only 12 out of the possible $4^5 - 4 = 1020$ mixed precision combinations are supported. This satisfies Goals 1 and 2.

5. From Example 8 in Section 2.9, we see that we want to be able to specify the internal precision at runtime; we do this with the variable PREC. This satisfies Goal 3.

6. PREC could potentially take on a great many values, whose meaning varied from machine to machine (see Section 3.3). To achieve Goals 1 and 4, we need to choose a parsimonious subset that will be available on all machines: Single (S), Double (D), Indigenous (I) and Extra (E). If an implementor wants to implement other ones (like tripled-double) or indeed if PREC=X (Extended) is supported because it is identical to PREC=I on an Intel machine,

that is allowed. Furthermore, if it is easier for the implementor to use *more* precision than the user requested, that is also allowed.

Therefore, provided the inputs are all single precision (real or complex), the only values of PREC that must be supported are S, D, I and E. On any particular machine, this would require at most two or three actual implementations, since either D=I (non-Intel IEEE machines), S=I (some Crays), and/or wider than requested precision may be used (so that S, D and I can all be implemented using I on Intel machines). Our reference implementation provides 3 versions: a true single precision version when PREC = S, a true double precision version when PREC = D or I, and a double-double version when PREC = E.

When the inputs are all double precision (real or complex), the only values of PREC that must be supported are D, I and E. Again, on any particular machine, at most two implementations are needed. On some Crays, I=S is shorter than D (this could be determined safely at run-time via environmental enquiries), so the implementor would use D in place of I on Crays. On non-Intel IEEE machines D=I. On Intel machines, D can be implemented using I. Our reference implementation provides 2 versions: a true double precision version when PREC = S, D or I, and a double-double version when PREC = E.

So the only really new implementation is PREC=E, which could either be implemented using double-double (as in our reference implementation) or native quadruple (if it exists).

7. The possible interpretations of PREC just discussed make it imperative that there be environmental enquiries so that the user can find out the actual machine epsilon (or over/underflow thresholds) of any requested internal precision, so that he or she can do error analysis, pick stopping criteria for iterations (as in Section 2.2), determine what values of PREC are legal, etc. We mimic the calling sequence of the LAPACK routine xLAMCH, but supply only that information needed to evaluate desired error bounds for computed quantities. The details of the environmental enquiry routine and the meanings of its parameters are described in section 6. This satisfies Goal 4.

5 Code Generation with M4 Macro Processor

In the existing BLAS, there are usually 4 routines associated with each operation. All input, output, and internal variables are single or double precision and real or complex. But under the new extended and mixed precision rules (see [1, Chapter 4]), the input, output and internal variables may have different precisions and types. Therefore, the combination of all these types results in many more routines associated with each operation. For example, DOT will have 32 routines altogether:

- 4 "standard" versions without mixed or extended precision
- 12 versions with mixed but no extended precision
- 4 versions with extended precision but no mixed precision, and
- 12 versions with both extended and mixed precision.

The first 16 routines in the above list, those without extended precision, have no PREC argument, whereas the last 16 do. In addition, the 16 versions with PREC argument support up to three internal precisions that can be chosen at runtime. Since there are 32 very similar routines to produce and maintain, we have automated the code generation as much as possible. We use the M4 [38] macro processor to facilitate this task. We note that although the BLAS operations differ from routine to routine, they all use the same basic arithmetic operations. All these arithmetic operations need only be coded once, and reused by the high level routines. The following subsections describe the two strategies to design the macro code.

5.1 Basic operations

The idea is to define a macro for each fundamental arithmetic operation. The macro's argument list contains the variables, accompanied by their types and precisions. For example, for the operation $c \leftarrow a + b$, we define the following macro:

```
ADD(c, c_type, a, a_type, b, b_type)
```

where, `x_type` can be one of:

real-single
real-double
real-extra
complex-single
complex-double
complex-extra

Inside the macro body, we use an "if-test" on `c_type`, `a_type` and `b_type`, to generate the appropriate code segment for "+". This is similar to operator overloading in C++; but we do it manually. All these if-tests are evaluated at macro-evaluation time, and do not appear in the executable code. Indeed, our goal was to produce efficient C code, which means minimizing branches in inner loops.

Other macros include `SUB, MUL, DIV, DECLARE` (variable declaration), `ASSIGN`, etc. Since these macros are shared among all the BLAS routines, we put them in a common header file, named `cblas.m4.h`, which has about 1100 lines of code.

A natural alternative to this macro approach would appear to be operator overloading. This would appear to simplify our job by not requiring us to pass in the type of each variable, because the compiler can figure it out. However, we note that the conventional operator overloading where the implementation of each binary operation depends only on the argument types and not the result type is *not* enough. For example, it cannot correctly choose between $Extra = Double \times Double$ and $Double = Double \times Double$, because it depends on the result type. An alternative would be to promote one argument to the output type, but this could be unnecessarily slow if implemented poorly (e.g. $Extra = Extra \times Double$ is significantly more expensive than $Extra = Double \times Double$.) Therefore, all these cases would have to be coded individually, and the logic would as complex as what we do now.

5.2 BLAS Functions

Each BLAS routine also has its own macro file, such as `dot.m4`, to generate the specific functions. All the macro files are located in the `m4/` subdirectory. Each macro file is structured in three levels. We take `dot.m4` as an example:

- The top level generates the 32 subroutines with the correct names, type statements, and switch statements based on PREC.
- The middle level is the actual dot product algorithm. It is written exactly once in such a way that lets it support all combination of types and precisions. It is about as long and complicated as a straightforward C implementation, with the difference that statements like $prod = x[i] * y[i]$ are converted into the M4 macro calls.
- The bottom level consists of calls to the macros that perform the fundamental operations.

For example, the inner loop of the M4 macro for the dot product is simply as follows (the M4 parameters `$2`, `$3`, and `$4` are types):

```
for (i = 0; i < n; ++i) {
    GET_VECTOR_ELEMENT(x_ii, x_i, ix, $2)
          /* put ix-th element of vector x into x_ii */
    GET_VECTOR_ELEMENT(y_ii, y_i, iy, $3)
          /* put iy-th element of vector y into y_ii */
    MUL(prod, $4, x_ii, $2, y_ii, $3) /* prod = x[i]*y[i] */
    ADD(sum, $4, sum, $4, prod, $4)   /* sum = sum+prod */
    ix += incx;
    iy += incy;
} /* endfor */
```

The motivation for this macro-based approach is to simplify software engineering. For example, the file `dot.m4` of M4 macros for the dot product is 401 lines long (245 non-comment lines) but expands into 11145 lines in 32 C subroutines implementing different versions of DOT. Similarly the macros for TRSV expand from 732 lines (454 non-comment lines) to 37099 lines in 32 C subroutines. (This does not count the shared M4 macros in the file `cblas.m4.h`.)

6 Testing

The goal of the testing code is to validate the underlying implementation. The challenge is twofold: First, we must thoroughly test routines claiming to use extra precision internally, where the test code is not allowed to declare any extra precision variables or use any other extra precision facilities not available to the code being tested. This requires great care in generating test data. Second, we must use M4 to automatically generate the many versions of test code needed for the many versions of the code being tested. For each BLAS routine, we perform the following steps in the test code:

1. Generate input scalars, vectors and arrays, according to the routine's specification, so that the result exposes the internal precision actually used.

2. Call the BLAS routine.

3. For each output, compute a "test ratio" of the computed error to the theoretical error bound, that is, $\frac{|Computed\_value - \text{“}True\_value\text{”}|}{ErrorBound}$.

In Step 1, we use a nested loop structure to loop over all the input parameters and vary all possible values for each parameter, to generate a large number of test cases. For example, for the dot product over 11,000 test cases were generated as described below.

By design, the test ratios computed in Step 3 should be bounded by 1. A large ratio indicates that the computed result is either completely wrong, or not as accurate as claimed in the specification. Indeed, our challenge was to construct our test examples so that the test ratio closely estimated (and certainly bounded) the ratio $\varepsilon_{int\_actual}/\varepsilon_{int\_claimed}$ between the actual internal precision used in the BLAS ε_{int_actual} and the claimed internal precision $\varepsilon_{int\_claimed}$ that should have been used.

There are two different kinds of test cases that we have to generate. The first kind is for routines that are like dot products. In addition to dot products themselves, this includes matrix-vector products, matrix-matrix products, axpbys, and other routines where each solution component is mathematically a kind of dot product. For dot products, the classical error bound provides our test ratio, and is used in section 4.3.3 of the Standard [1] to specify the required accuracy. Section 6.2 describes how we generate dot products of real data, section 6.3 describes the complications that arise when the matrix is symmetric or banded, and section 6.4 describes how complex and mixed precision cases are tested.

The second kind of test case is for the solution of triangular systems of equations. Here the classical error bound for the solution of triangular systems, which is part of the specification in section 4.3.3 of the Standard [1], does *not* lend itself to straightforwardly computing a test ratio as described above. Instead, we must test indirectly by testing each solution component of a triangular system as a dot product. This is described in more detail in section 6.5.

6.1 Environmental Enquiries

The machine dependent interpretations of PREC require us to have an environmental enquiry function to describe the error bounds we expect the BLAS to satisfy. The BLAS Standard [1] provides one such a function FPINFO in Sections 1.6 and 2.7, which is entirely analogous to the LAPACK routine xLAMCH [2]. However, this function only provided information about single precision (SLAMCH) and double precision arithmetic (DLAMCH), not indigenous or extra precision. Indeed, it may very well not be possible to return the overflow and underflow thresholds for indigenous arithmetic in any floating point variable that the user is permitted to declare! To avoid this problem, section 4.3.3 of the BLAS Standard describes a function FPINFO_X(CMACH,PREC) that returns only integer information (like maximum and minimum exponents) describing arithmetic of precision PREC. Here is a table of values that input argument CMACH can have, and the result returned by FPINFO_X.

CMACH	Integer value returned by FPINFO_X
BASE	base of the machine
T	number of "correct" (BASE) digits in the floating point mantissa
RND	1 when "proper rounding" occurs in addition, 0 otherwise
IEEE	1 when rounding in addition occurs in "IEEE style" 0 otherwise
EMIN	minimum exponent before (gradual) underflow
EMAX	maximum exponent before overflow

Based on these integers, we may derive the following real values, whose interpretation we explain below. Note that T is chosen to make the formula for EPS below give the right value; if rounding is not perfect, as in any efficient implementation of double-double, then T will be somewhat less than the number of digits used to represent double-double numbers.

Real parameter	Description
EPS	machine epsilon $= .5 \cdot \mathrm{BASE}^{1-\mathrm{T}}$ if RND = 1 $= \mathrm{BASE}^{1-\mathrm{T}}$ if RND = 0
UN	underflow threshold $= \mathrm{BASE}^{\mathrm{EMIN}}$
OV	overflow threshold $= \mathrm{BASE}^{\mathrm{EMAX}+1} \cdot (1 - \mathrm{EPS})$

For PREC = S or D=I on an IEEE machine, there is no difference between these parameters and the like-named ones returned by xLAMCH; for completeness we include their values in a table below. And for PREC = I meaning double-extended on an Intel machine, there is also no ambiguity.

But for PREC = E, we must be more careful in their interpretation, because double-double implementations, including our reference implementation, are not implemented as accurately as IEEE arithmetic in hardware. For example, EPS may well be larger than BASE^{-D}, where D is the actual number of (BASE) digits used to store numbers. Also, part of a double-double number may underflow without the other part underflowing. Finally, OV may be very slightly smaller than the actual overflow threshold because 1-EPS is slightly smaller than the largest floating point number less than 1.

Because of these reasonable variations in the way double-double can be implemented, the Standard itself makes no requirements about how accurate an individual floating point operation must be. Instead, it only requires that certain error bounds described below for overall BLAS operations should be proportional to EPS in the absence of over/underflow. These error bounds of course depend both on the error bounds of individual floating point operations and on the algorithms used for the BLAS, but not specifying bounds for individual floating point operations gives implementors more freedom to optimize. In this section, we will be quite specific about the accuracy of our reference implementation.

So for real floating arithmetic, the following formulas are what we need to know about the arithmetic done internally to our reference implementation of the BLAS in the absence of overflow (i.e. assuming neither the inputs a and b nor the output exceed OV). For real addition and subtraction,

$$fl(a \pm b) = [a(1+\delta_a)] \pm [b(1+\delta_b)] + \eta \tag{2}$$

where $|\delta_a| \leq$ EPS, $|\delta_b| \leq$ EPS, and $|\eta| \leq$UN. For real multiplication and division (represented by $\odot$)

$$fl(a \odot b) = (a \odot b)(1+\delta) + \eta \tag{3}$$

where $\delta_a \leq$ EPS and $|\eta| \leq$UN. In other words, δ_a, δ_b and δ are the roundoff errors, and η is the underflow error, if it occurs. (In fact addition and subtraction for our reference implementation satisfy bound (3), but (2) which is cheaper to satisfy, would have been enough).

Here is a table of values returned by FPINFO_X for our reference implementation, and of the values of EPS, OV, and UN.

CMACH	PREC = S	PREC = D or I	PREC = E
BASE	2	2	2
T	24	53	105
RND	1	1	0
IEEE	1	1	0
EMIN	-126	-1022	-1022
EMAX	127	1023	1023
EPS	$2^{-24} \approx 5 \cdot 10^{-8}$	$2^{-53} \approx 10^{-16}$	$2^{-104} \approx 5 \cdot 10^{-32}$
UN	$2^{-126} \approx 10^{-38}$	$2^{-1022} \approx 10^{-308}$	$2^{-1022} \approx 10^{-308}$
OV	$2^{128}(1-2^{-24}) \approx 10^{38}$	$2^{1024}(1-2^{-53}) \approx 10^{308}$	$2^{1024}(1-2^{-104}) \approx 10^{308}$

Now consider complex arithmetic. The algorithms for complex addition and multiplication are straightfoward (for multiplication we do 4 real multiplications and 2 real additions/subtractions) (see also [14]). Assuming a and b are complex numbers, with real operations satisfying bounds (2) and (3) above, we get

$$fl(a \pm b) = [a(1+\delta_a)] \pm [b(1+\delta_b)] + \eta \tag{4}$$

with δ_a, δ_b and η complex numbers satisfying $|\delta_a| \leq \sqrt{2}\cdot$EPS, $|\delta_b| \leq \sqrt{2}\cdot$EPS and $|\eta| \leq \sqrt{2}\cdot$UN, and

$$fl(a \times b) = (a \times b)(1+\delta) + \eta \tag{5}$$

with δ and η complex numbers satisfying $|\delta| \leq 2\sqrt{2}\cdot$EPS and $|\eta| \leq 3\sqrt{2}\cdot$UN.

Complex division is more complicated. The simplest possible algorithm

$$q = q_r + iq_i = \frac{a_r + ia_i}{b_r + ib_i} = \frac{a_r \cdot b_r + a_i \cdot b_i}{b_r^2 + b_i^2} + i \cdot \frac{-a_r \cdot b_i + a_i \cdot b_r}{b_r^2 + b_i^2} \tag{6}$$

overflows and underflows unnecessarily often, for example when $a = b$ (so $q = 1$) and both have magnitudes greater than the square root of the overflow threshold. A commonly used and reasonable algorithm is due to Smith [34, pp. 195]:

$$\begin{aligned} \text{if } |b_r| > |b_i| \quad \text{then} \quad & q_r + iq_i = \frac{a_r + a_i \cdot \frac{b_i}{b_r}}{b_r + b_i \cdot \frac{b_i}{b_r}} + i \cdot \frac{a_i - a_r \cdot \frac{b_i}{b_r}}{b_r + b_i \cdot \frac{b_i}{b_r}} \\ \text{else} \quad & q_r + iq_i = \frac{a_i + a_r \cdot \frac{b_r}{b_i}}{b_i + b_r \cdot \frac{b_r}{b_i}} + i \cdot \frac{-a_r + a_i \cdot \frac{b_r}{b_i}}{b_i + b_r \cdot \frac{b_r}{b_i}} \quad . \end{aligned} \tag{7}$$

But it can suffer from intermediate underflow, so only when the numerator and denominator are large enough in magnitude does Smith's algorithm satisfy an error bound of the form (3) [14].

Since complex division occurs rarely in the BLAS (n times, versus $O(n^2)$ for multiplication and division in TRSV, although the ratio could be closer to one for TBSV) we have chosen to have a yet more careful implementation in our reference implementation, that scales the numerator and denominator if they are two small or too large, in order to satisfy a bound like (3):

$$fl(a/b) = (a/b)(1+\delta) + \eta \tag{8}$$

where $|\delta| \le (6+4\sqrt{2})\cdot$EPS and $|\eta| \le \sqrt{2}\cdot$UN. Our scaled version of Smith's algorithm (7) appears in Appendix B.

We repeat that since the Standard states no requirements about error bounds in the presence of over/underflow, such careful implementations of arithmetic, especially complex division, are not required. However, since they do not slow down most BLAS and do make the error bounds much simpler to state, we strongly recommend them.

6.2 Testing DOT

The DOT function computes:

$$r \leftarrow \beta \cdot r_{in} + \alpha \cdot \sum_{i=1}^{n} x_i \cdot y_i \tag{9}$$

By careful error analysis (using bounds (2) and (3) for real data and bounds (4) and (5) for complex data) we can derive the following error bound in the absence of overflow (see Appendix A):

$$|r_{comp} - r_{acc}| \le (n+2)\cdot(\varepsilon_{int} + \varepsilon_{acc})\cdot S + U + \varepsilon_{out}\cdot|r_{acc}| \tag{10}$$

where

$r_{comp} = r$ computed by the routine being tested, with the accumulation done in internal precision ε_{int}, and then rounding to the output precision ε_{out}
$r_{acc} =$ the computed result using precision ε_{acc}
$\varepsilon_{int} =$ internal precision claimed by the routine being tested
$\upsilon_{int} =$ underflow threshold in the claimed internal precision
$\varepsilon_{acc} =$ our most accurate precision (106 bits) to compute test value r
$\upsilon_{acc} =$ underflow threshold in our most accurate precision
$\varepsilon_{out} =$ output precision
$\upsilon_{out} =$ underflow threshold in output precision
$S = |\alpha|\cdot\sum_{i=1}^{n}|x_i\cdot y_i| + |\beta\cdot r_{in}|$
$U = \max(2|\alpha|\cdot n+3,\ \sum_{i=1}^{n}|y_i|+2n+1,\ \sum_{i=1}^{n}|x_i|+2n+1)\cdot(\upsilon_{int}+\upsilon_{acc})+\upsilon_{out}$

The ε_{int}, ε_{acc} and ε_{out} terms may be derived from FPINFO_X as described in the last section. With real arithmetic they correspond to EPS for appropriate values of PREC. With complex arithmetic ε_{int} and ε_{acc} should be multiplied by $2\sqrt{2}$ and U should be multiplied by $3\sqrt{2}$ as required by bounds (4) and (5).

The U term is to accommodate all possible underflows at various places; in the absence of underflow it is zero. It takes into account the three possible ways of inserting parentheses to multiply α with x_i and y_i. The υ_{int}, υ_{acc} and υ_{out} terms may be derived from FPINFO_X as described in the last section. With real arithmetic they correspond to UN for appropriate values of PREC. With complex arithmetic υ_{int} and υ_{acc} should be multiplied by $3\sqrt{2}$ as required by bounds (4) and (5).

The returned "test ratio" is the ratio of the left side over the right side of inequality (10). That is,

$$ratio = \frac{|r_{comp} - r_{acc}|}{(n+2)\cdot(\varepsilon_{int} + \varepsilon_{acc})\cdot S + U + \varepsilon_{out}\cdot|r_{acc}|} \tag{11}$$

This ratio should be at most 1, and often is rather less than 1, because in the error bound n is pessimistic and $\sqrt{n}$ is more typical if rounding is equally likely to be up or down.

In order to estimate whether the implemented internal precision is at least as high as claimed by the routine, it is not sufficient to generate random inputs. Because for random inputs, the last term in the error bound (10) will very likely dominate the error bound, and wipe out the first term. Therefore our strategy is to make $|r|$ (hence $|r_{acc}|$) much smaller than S, so that ε_{int} term dominates the error bound. In our test generator, we choose α, β, r, x and y judiciously so as to cancel as many bits as possible when evaluating r.

If the routine being tested claims to use internal precision $\varepsilon_{int\_claimed}$ with underflow threshold $\upsilon_{int\_claimed}$, then the denominator of our test ratio will be the right-hand side of error bound (10) with $\varepsilon_{int} = \varepsilon_{int\_claimed}$ and $\upsilon_{int} = \upsilon_{int\_claimed}$. The actual error is bounded by the same formula with $\varepsilon_{int} = \varepsilon_{int\_actual}$ and $\upsilon_{int} = \upsilon_{int\_actual}$. Presuming that over many tests, the actual error occasionally approaches its bound, then our test ratio will be as large as

$$\frac{(n+2)\cdot(\varepsilon_{int\_actual} + \varepsilon_{acc})\cdot S + U_{actual} + \varepsilon_{out}\cdot|r_{acc}|}{(n+2)\cdot(\varepsilon_{int\_claimed} + \varepsilon_{acc})\cdot S + U_{claimed} + \varepsilon_{out}\cdot|r_{acc}|} \tag{12}$$

Since $\varepsilon_{acc} \leq \min(\varepsilon_{int\_actual}, \varepsilon_{int\_claimed})$, the U terms are typically much smaller than the other terms, and since r_{acc} is also very small by our choice of data (Section 6.2.1), the above ratio is roughly

$$\frac{\varepsilon_{int\_actual}}{\varepsilon_{int\_claimed}} \tag{13}$$

which we compare to 1. Therefore, a large ratio gives us a sense of the actual precision, assuming underflow does not dominate.

The error bound in section 4.3.3 of the Standard is derived from bound (10) by assuming underflow does not occur (letting us set $U = 0$), taking $\varepsilon_{int} = 2\sqrt{2}$EPS (where EPS corresponds to internal precision PREC), which corresponds to the bounds for complex arithmetic, and taking $\varepsilon_{acc} = 0$, i.e. ignoring the error in our test generation software. Our testing software can only confirm the slightly weaker bound (10). Thus, an implementation conforming to the Standard will pass our tests, and an implementation not conforming to the Standard will almost certainly be detected.

6.2.1 Generating input scalars and vectors

In the test code, we must generate three scalars α, β, r, (r is overwritten by the inner product on output) and two vectors x and y. To make the inner product small in magnitude, we designed the following algorithm to generate these quantities. (Here, we assume that the inputs and output are in single precision, and internal precision is double-double. The other situations are simpler than this.)

1. Choose n, random α, β and $x_i, i = 1 \dots n$.

2. Choose the leading $y_i, i = 1 \dots n-k+1$, to add bits into the prefix sum, $\alpha \cdot \sum_{i=1}^{n-k+1} x_i \cdot y_i$, such that the prefix sum contains at least 106 bits.

```
$y_1 = random()$
$s = \alpha \cdot x_1 \cdot y_1$
do $j = 2, n-k+1$
    $s = s \cdot 2^{-30}$                /* shift right 30 bits */
    $y_j = \frac{s}{\alpha \cdot x_j}$
enddo
```

3. Choose the remaining $y_i, i = n-k+2 \dots n$ and r, to cancel as many bits as possible.

```
do $j = n-k+2, n$
    $s = \alpha \cdot \sum_{i=1}^{j-1} x_i \cdot y_i$ /* very accurately */                    (*)
    $y_j = -\frac{s}{\alpha \cdot x_j}$          /* $s + \alpha \cdot x_j \cdot y_j$ cancels the leading 24 bits of $s$ */
enddo
$s = \alpha \cdot \sum_{i=1}^{n} x_i \cdot y_i$          /* very accurately */
$r = -\frac{s}{\beta}$                       /* $s + \beta r$ cancels the leading 24 bits of $s$ */
```

The pseudorandom numbers generated are uniformly distributed in $(0, 1)$.

In summary, the first $(n - k + 1)$ y_i's are chosen to add bits into the prefix sum, and the last $(k - 1)$ y_i's, together with r, are chosen to cancel bits in the final sum.

How big should k be, i.e., how many cancellations do we need? It depends on the relative sizes of ε_{out} and ε_{acc}. Basically,

$$k = \left\lceil \frac{\log \varepsilon_{acc}}{\log \varepsilon_{out}} \right\rceil \tag{14}$$

which is as large as $\lceil 106/24 \rceil = 5$ for single precision output and double-double accurate precision. This means that we cannot test dot products so simply with $n < 6$.

For smaller n, we do not have sufficient freedom to add and cancel all 106 bits. Instead, we use some algebraic identities to make the inner product small. For example,

1. $n = 1$
   Choose $\alpha = \beta$ at random, $a \in [1/2, 1)$ at random with only 12 leading bits nonzero, $x_1 = a + \varepsilon_{out}$ exactly, $y_1 = a - \varepsilon_{out}$ exactly, $r = -a^2$ exactly, so that the final result equals $-\alpha \cdot \varepsilon_{out}^2$.

2. $n = 2$
   Choose $\alpha = \beta$ at random, $r = x_1$, and $y_1 = -1$, to make $\alpha \cdot x_1 \cdot y_1 + \beta \cdot r = 0$ exactly. Choose y_2 such that the final result $\alpha \cdot x_2 \cdot y_2$ is much smaller than $\beta \cdot r$ in magnitude.

3. $n = 3$
   Choose $r = 0$. Choose $y_1 = -x_3, y_3 = x_1$ to make $x_1 \cdot y_1 + x_3 \cdot y_3 = 0$ exactly. Choose y_2 such that the final result $\alpha \cdot x_2 \cdot y_2$ is much smaller than $\alpha \cdot x_1 \cdot y_1$ in magnitude.

4. $n = 4$
   Choose $r = 0$. Choose $y_1 = -x_4, y_3 = 0, y_4 = x_1$ to make $x_1 \cdot y_1 + x_3 \cdot y_3 + x_4 \cdot y_4 = 0$ exactly. Choose y_2 such that the final result $\alpha \cdot x_2 \cdot y_2$ is much smaller than $\alpha \cdot x_1 \cdot y_1$ in magnitude.

5. $n = 5$
   Choose $r = 0$. Choose $y_1 = -x_5, y_2 = -x_4, y_4 = x_2, y_5 = x_1$ to make $x_1 \cdot y_1 + x_2 \cdot y_2 + x_4 \cdot y_4 + x_5 \cdot y_5 = 0$ exactly. Choose y_3 such that $|x_3 \cdot y_3|$ is much smaller than $\max(|x_1 \cdot y_1|, |x_2 \cdot y_2|)$.

In all the cases above, we have at least 2 summands available in the inner product (9). According to our design strategy, we can cancel from 24 to 106 leading bits in the final sum. That is, $10^{-32} \le |r_{acc}| \le 10^{-8}$.

For some special inputs, however, where there is at most 1 term in the inner product, $|r_{acc}|$ cannot be made very small, and is $O(1)$. These include:

- $n = 0$
- $\alpha = 0$
- $\alpha \neq 0, \beta = 0$, and $n \le 1$

For these cases, any internal precision higher than the output precision could not be revealed by our tests, and indeed any higher internal precision cannot change the answer by more than one ulp (making the answer slightly less accurate!).

6.2.2 Obtaining an accurate r_{acc}

One possibility is to use an arbitrary precision package, such as MPFUN, but our goal is to make this test code self contained, and use no extra precision facilities not available to the code being tested. Since most of our routines can be reduced to a series of dot products, testing can be based on DOT. We only need a *trusted* dot routine. Currently, we are using our own dot routine with double-double internal precision to compute r_{acc}, which is accurate to 106 bits. That is, $\varepsilon_{acc} = 2^{-106}$, and $\upsilon_{acc} = 2^{-968}$. This implies that any internal precision higher than double-double cannot be detected, and may result in a tiny test ratio. A very tiny test ratio (such as zero) may also occur if the result happens to be computed exactly.

The careful reader may wonder how we can trust test code that uses our own accurate dot product to test itself, as well as all other BLAS. Some test cases are generated where r_{acc} is known exactly by a mathematical formula, because there is exact term-by-term cancellation, and that we do not depend on our trusted dot product being correct. Indeed, our own trusted accurate dot product had to pass these other tests. These are precisely the cases when $1 \le n \le 5$ in see Section 6.2.1.

6.3 Testing SPMV and GBMV

The SPMV and GBMV (matrix vector product, and many other Level 2 BLAS routines) compute:

$$y \leftarrow \beta \cdot y + \alpha \cdot A \cdot x \qquad (15)$$

where, A is a symmetric or band matrix of dimension $n \times n$.

Testing it is no more difficult than testing DOT, because each component of the output vector y is a dot product of the corresponding row of A with vector x, and satisfies the error bound (10). So we can iterate over the n rows, generate each row of A using *almost* the same test generator as DOT, and compute a test ratio for each component of the output y.

However, some modifications are needed in the test generator. Consider the case of symmetric A. For the first row of A, we can use exactly the same algorithm discussed in Section 6.2.1. For the second row, by symmetry, the (2,1) entry should equal the (1,2) entry, so we only have $n-1$ free entries to choose. Furthermore, the x vector is fixed. For the third row of A, we only have $n-2$ free entries to choose, and so on. To accommodate this need, our test generator is modified as follows. The inputs are those free parameters chosen from a prior call to the generator. The outputs are the free parameters chosen from this call. The generator first evaluates the partial sum using the fixed input parameters, and computes the number of bits B available in the partial sum. If $B < 106$, we will add a few more terms into the partial sum such that $B \geq 106$. Afterwards, we will generate the remaining free parameters to cancel bits in the running sum. The way to add and cancel bits is the same as what we described in Section 6.2.1.

This approach can be generalized to most other Level 2 and Level 3 BLAS, except for triangular solve (see Section 6.5) and Hermitian matrix matrix product (see Section 6.4.3).

6.4 Testing the Complex and Mixed Precision Routines

In principle, the testing of the complex routines is no harder than that of the real routines, and all of our testing algorithms described in the preceding sections can be used in the complex cases. However, it is worth noting some subtle differences from the real cases, and the techniques we have used to reuse the existing test generators of the real cases. Using these techniques greatly reduces the amount of test software, particularly for those routines with mixed real and complex types and precisions.

6.4.1 Testing DOT

Our error bound (10) derived in Appendix A applies to real or complex arithmetic DOT function. The difference between the two is that for complex data, the part of the error bound proportional to $\varepsilon_{int} + \varepsilon_{acc}$ is larger by a factor $2\sqrt{2}$, and the part of the error bound U coming from underflow is larger by a factor $3\sqrt{2}$. Using the same error analysis, we arrive at the following error bound for the complex DOT function. The detailed derivation is omitted.

$$|r_{comp} - r_{acc}| \leq 2\sqrt{2} \cdot (n+2) \cdot (\varepsilon_{int} + \varepsilon_{acc}) \cdot S + 3\sqrt{2} \cdot U + \varepsilon_{out} \cdot |r_{acc}| \qquad (16)$$

See Section 6.2 for the notation. The most noticeable differences are the constant factors associated with each term: $2\sqrt{2}$ and $3\sqrt{2}$. This new bound is used in computing the test ratio (see Equation (11)).

Second, since the algorithms for the test generator contain complex logic to make decisions based on precisions and types (see Section 6.2.1), we can not easily macroize them using M4. Instead, we have to code them manually. On the other hand, we do not want to write 16 versions of the generator for each combination of the input types and precisions, because that will be too much work and more importantly, much too error-prone. Our compromise is as follows. We manually write only 4 generators for the un-mixed routines: single (C), double (D), single complex (C) and double complex (Z). For all the other routines, we only build wrappers around these 4 generators. There are two cases.

1. For the mixed precision routines with the same data type (real or complex), we use the generator corresponding to the lowest input precision. For example, to test `c_zdot_c_c()`,

we use single complex (C) test generator. For the higher precision variables, we pad zeros in the trailing part. As long as the vector length is long enough ($n \geq 6$), we will get significant cancellation, see Section 6.2.1.

2. For the mixed real and complex routines, we first use the real test generators (single (S) or double (D)) to generate the real inputs. Then, we apply simple complex scalings to the complex inputs and outputs. For example,

type				scaling				
α, β, r	x	y	generator	α	x	y	β	r
C	S	S	S	$1+i$	1	1	$1-i$	i
C	S	C	S	$1+i$	1	$1+i$	$1+i$	$1+i$
Z	D	D	D	$1+i$	1	1	$1-i$	i
Z	D	Z	D	$1+i$	1	$1+i$	$1+i$	$1+i$

As long as the complex scalars are of the form 2^j, $i \cdot 2^k$, or $2^j + i \cdot 2^k$ with integers j and k, we do not introduce any roundoff errors in this scaling.

6.4.2 Testing WAXPBY

WAXPBY (scaled vector accumulation) computes:

$$w \leftarrow \alpha \cdot x + \beta \cdot y \tag{17}$$

where, x, y and w are vectors of length n. In principle, we can treat each component as a DOT function of length 2, and use the DOT testing code. However, since the length $n = 2$ is small, DOT generators can only produce good cancellations for the routines with un-mixed types: S, D, C, and Z. For the routines with mixed precisions, such as `c_dwaxpby_s_s()` (α, β, w double, x, y single), if we were to use the DOT test generator, the result only cancels the leading 24 bits. We can achieve more cancellations using another algebraic identity. Here is how we do it.

Consider the identity $a^6 - 1 = (a^2 - 1)\ (a^4 + a^2 + 1)$, where a is a random integer in $(0, 2^{12})$. Let $\alpha = a^4 \cdot 2^{-48}$, $\beta = (a^4 + a^2 + 1) \cdot 2^{-48}$, $x_i = a^2 \cdot 2^{-24}$, and $y_i = -(a^2 - 1) \cdot 2^{-24}$. By design, $a \cdot 2^{-12}$ is a real random number in $(0, 1)$ and has only 12 leading nonzero bits. Therefore, all of α, β (both 53 bits), and x_i, y_i (both 24 bits) can be represented exactly. The final result is

$$w_i = a^4 \cdot 2^{-48} \cdot a^2 \cdot 2^{-24} + (a^4 + a^2 + 1) \cdot 2^{-48} \cdot (-(a^2 - 1) \cdot 2^{-24}) = 2^{-72} \ ,$$

which cancels down to the 72-nd bit exactly.

This scheme also applies to the other real routines with mixed precisions: `c_dwaxpby_s_d()` and `c_dwaxpby_d_s()`. For the routines involving complex input/output, we first generate the real counterparts, then apply the complex scalings similarly to the DOT testing as described in Section 6.4.1.

6.4.3 Testing HEMM

HEMM (Hermitian matrix matrix product) computes:

$$C \leftarrow \alpha \cdot A \cdot B + \beta \cdot C \tag{18}$$

where A is a Hermitian matrix, B is a real or complex matrix, and C is a complex matrix. Each entry of A satisfies $A_{ij} = \bar{A_{ji}}$, the complex conjugate.

For the pure complex cases, i.e., B is complex, we can generate the test matrices similarly to SPMV, i.e., treating each row as a dot product. When B is real, our DOT test generator generates real data first and applies a simple complex scaling, see Case 2 of Section 6.4.1. However, this complex scaling does not work for a Hermitian A, because in each row of A we need different complex scalars due to conjugation. Therefore, we have designed the following algorithms to generate the test inputs.

1. B is real.

   We generate real and imaginary parts separately and then apply scaling (to make α and β complex) as follows.

   (a) Generate real matrices R_1, A_1, B, C_1 and real scalars α and β using the symmetric test matrix generator, such that,

   $$R_1 = \alpha \cdot A_1 \cdot B + \beta \cdot C_1$$

   here, A_1 is real symmetric. R_1 is computed in extra precision.

   (b) Generate real matrices R_2, A_2, and C_2 with α, β, and B being fixed, using the skew symmetric test matrix generator, such that,

   $$R_2 = \alpha \cdot A_2 \cdot B + \beta \cdot C_2$$

   here, A_2 is real skew symmetric ($A_2^T = -A_2$). R_2 is computed in extra precision.

   (c) Now let $R = R_1 + i\ R_2$, $A = A_1 + i\ A_2$, $C = C_1 + i\ C_2$. Note that A is now Hermitian.

   (d) Finally, apply the following scalings to make α and β complex if necessary. There are four cases to consider, depending on the values of α and β.

values		scaling					
α	β	α	A	B	β	C	R (truth)
1	1	1	1	1	1	1	1
1	any	1	1	1	$-i$	i	1
any	1	i	1	1	1	i	i
any	any	$1+i$	1	1	$1+i$	1	$1+i$

   Note that we can scale R by $1 + i$, since no rounding error is introduced by this scaling.

2. B is complex.

   We first generate the case with B real in the same way as case 1, except for the scalings in step (d) for which we apply the following scalings to make matrix B complex.

values		scaling					
α	β	α	A	B	β	C	R (truth)
1	1	1	1	i	1	i	i
1	any	1	1	$1+i$	$1+i$	1	$1+i$
any	1	$1+i$	1	$1+i$	$2i$	1	$2i$
any	any	$1+i$	1	$1+i$	$2i$	1	$2i$

After the test matrices are generated, we then call HEMM routines, and for each entry C_{ij}, we compute a test ratio using the DOT test ratio computation routine, with R_{ij} as the truth r_{acc}. In the end, we report the maximum ratio among all the n^2 ratios.

6.5 Testing TRSV

The TRSV routine solves the system of equations:

$$x \leftarrow \alpha \cdot T^{-1} \cdot b \tag{19}$$

where T is a unit, non-unit, upper or lower triangular matrix of dimension $n \times n$, and typically x overwrites b (for mathematical simplicity, we will use different names x and b, even if they occupy the same storage at different times).

The correct way to solve with extra internal precision PREC is to perform all computations and keep all intermediate quantities (including components of x) in precision PREC, and then round x from precision PREC to the desired output precision at the end. If PREC is more precise than the output precision, this will require an extra working array of n numbers of precision PREC to store x, before rounding x at the end.

To derive the error bounds below, we use formulas (2) and (3) for real arithmetic, and formulas (4), (5), and (8) for complex arithmetic. We can either first multiply $\alpha \cdot b$ and then solve for x, or else solve $T\bar{x} = b$ and then multiply $x = \alpha\bar{x}$. In the first case, when we first compute $\alpha \cdot b$, the error bound in the absence of overflow is

$$(T + E)(x_{comp} + e) = \alpha(b + f) + U \tag{20}$$

where

$$\begin{aligned}
|E_{ij}| &\leq n\varepsilon_{int} \cdot |T_{ij}| \ , \\
|e_i| &\leq \upsilon_{out} + \varepsilon_{out} \cdot |x_i| \ , \\
|f_i| &\leq n\varepsilon_{int} \cdot |b_i| \ , \\
|U| &\leq (2n - 1 + |T_{ii}|) \cdot \upsilon_{int} \ .
\end{aligned}$$

On the other hand, if we first perform the substitution and then multiply $x = \alpha\bar{x}$, the error terms in Equation (20) are bounded by

$$\begin{aligned}
|E_{ij}| &\leq n\varepsilon_{int} \cdot |T_{ij}| \ , \\
|e_i| &\leq \upsilon_{out} + \varepsilon_{out} \cdot |x_i| \ , \\
|f_i| &\leq n\varepsilon_{int} \cdot |b_i| \ , \\
|U| &\leq (2n - 1 + |T_{ii}|) \cdot |\alpha| \cdot \upsilon_{int} \ .
\end{aligned}$$

In the case of complex arithmetic, ε_{int} should be multiplied by $6+4\sqrt{2}$, υ_{out} should be multiplied by $\sqrt{2}$, and υ_{int} should be multiplied by $3\sqrt{2}$, as required by bounds (4), (5), and (8). In the absence of underflow (except possibly on the final rounding of x from PREC to output precision), the term

$U = 0$. In the complete absence of underflow (including the final rounding of x), then $U = 0$ and $|e_i| \le \varepsilon_{out}|x_i|$.

To derive the bounds with underflow for complex arithmetic we needed to use error bound (8), which in turn required a quite careful implementation of complex division with scaling to avoid intermediate underflows. If a less careful implementation of complex division is used, then a conforming implementation is still possible because the Standard only specifies the error bound to be satisfied in the absence of over/underflow. Still, we strongly recommend using the careful implementation.

Unfortunately, it appears to be difficult to test TRSV systematically using the above error bounds in the same way that error bound (10) led to a test ratio for the dot product. The reason is that there is no simple formulas for the smallest E, e, f and U satisfying (20), even in the absence of underflow. If we did not round x on output, which would make $e = 0$, then Oettli-Prager's Theorem [41] would give a closed form formula for the smallest E and f satisfying (20), using the residual $Tx - \alpha b$. This formula exists because E and f appear linearly in (20), and furthermore each row can be chosen independently. But if e is present, the nonlinear term Ee means no such formula is available. For all but the most ill-conditioned T (the interesting cases!) the rounding proportional to ε_{out} in e has a larger effect than the much smaller rounding in E and f proportional to $\varepsilon_{int} \ll \varepsilon_{out}$, so we need a different testing scheme.

Instead of testing the bound (20) directly, we will use the same DOT-based testing scheme as before, by examining a single component of x. Consider a lower triangular matrix T, where the components of the solution vector x are obtained one by one from the following substitution algorithm:

$$x_i = \frac{\alpha \cdot x_i - \sum_{j=1}^{i-1} T_{ij} \cdot x_j}{T_{ii}} \ , \quad i = 1, ..., n \tag{21}$$

Therefore, we must test whether 1) the DOT product in the numerator is computed accurately, and 2) the division is performed accurately. If extra precision is specified, all the components of x should be computed and stored internally in extra precision. Since the i-th component x_i depends not only on the i-th row of T but also on all the preceding $x_j,\ j = 1, ..., i-1$ computed before, it is difficult to test all the x components simultaneously using a single test matrix. We designed a scheme that tests the x components one at a time. In this scheme, we generate n sets of input data, each of which tests only one component of x. The i-th test system is generated as:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \leftarrow \alpha \cdot \begin{bmatrix} T_{11} & & & & & \\ 0 & T_{22} & & & & \\ 0 & 0 & \ddots & & & \\ T_{i1} & T_{i2} & \cdots & 1 & & \\ 0 & 0 & 0 & 0 & \ddots & \\ 0 & 0 & 0 & 0 & 0 & T_{nn} \end{bmatrix}^{-1} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \tag{22}$$

In this triangular matrix, only the i-th row (we call this *principal row*) and the main diagonal are nonzero, all the other entries are zero. The solution vector is computed by:

$$x_j \quad = \quad \frac{\alpha \cdot x_j}{T_{jj}} \ , \ j \ne i, \ \ \text{compute and store in precision PREC if needed} \tag{23}$$

$$x_i \;=\; \alpha \cdot x_i - \sum_{j=1}^{i-1} T_{ij} \cdot x_j \tag{24}$$

In the test system (22), α, the main diagonal of T ($T_{jj}, j \neq i$), and the right-hand side vector (initial x) can be be generated at random. The $x_j, j \neq i$ are computed by (23). But we have to be very careful in generating the principal row of T, because we will use it to validate that x_i is accurately computed by (24). Since (24) is essentially a DOT function, we can use the same testing technique as used for DOT, which is described in Section 6.2. The only modification is that the test generator sometimes shall take the fixed input $x_j, j = 1, ..., i-1$ in extra precision.

In summary, we compute n test ratios for this system. For each non-principal component $x_j, j \neq i$, we compute a test ratio

$$\frac{|x_{j\,comp} - x_{j\,acc}|}{|x_{j\,comp}| \cdot 2 \cdot \varepsilon_{out}} \tag{25}$$

For the principal component x_i, we compute a test ratio using Equation (11).

System (22) is designed to test x_i thoroughly. We need to generate n such systems with i varying from 1 to n in order to test all the components thoroughly.

6.5.1 Testing Complex TRSV

Since TRSV is tested by treating each solution component as a dot product, the same approach to testing complex dot products that was used before can be used here.

7 Conclusions and Future Work

In this paper, we motivated the introduction of extended and mixed precision BLAS. By considering the costs and benefits of various styles of extended and mixed precision computation, we justified the design decisions made in the new BLAS Standard. We also described our macro-based approach for automatically generating C implementations and testing algorithms for most of the routines. This demonstrates that the proposed Standard is implementable with a reasonable amount of effort. For our reference implementation, we assumed that the hardware floating-point arithmetic conformed to IEEE Standard. Furthermore, our extended precision implementation is efficient, because of the high degree of data reuse in double-double arithmetic. We believe similarly structured M4 macros could generate Fortran 77 and Fortran 95 code as well.

There are several avenues we would like to explore in the future. First, we will finish the code generation and testing for all the remaining functions in Chapter 4 of the BLAS Standard.

Second, we will evaluate the overall performance of the library. Our code in the present form is meant to be a reference implementation, serving much the same purpose as the existing Fortran BLAS distribution in Netlib. Our generated code consists only of straightforward loops, without any architectural optimizations, such as blocking for locality or loop unrolling. Because of the high degree of data reuse in double-double arithmetic, this yielded good performance for extended precision, but further improvements are possible, especially for mixed precision. We plan to expand our macros to automatically generate optimized code in a similar way as ATLAS [50] and PHiPAC [7].

And third, we will investigate the benefit of the new BLAS for more linear algebra algorithms and applications.

Acknowledgments

Yulin Li, Weihua Shen, Berkat Tung and Ben Wanzo contributed codes in the early stage of the development. Adam Delisse and Hsin-Ying Lin helped with testing on the HP machines.

References

[1] Document for the Basic Linear Algebra Subprograms (BLAS) Standard, May 2000. `http://www.netlib.org/utk/papers/blast-forum.html`.

[2] 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 Users' Guide, Release 3.0.* SIAM, Philadelphia, 1999. 407 pages.

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

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

[5] D. H. Bailey. A Fortran-90 based multiprecision system. *ACM Trans. Math. Soft.*, 21(4):379–387, 1995.

[6] David Bailey. A Fortran-90 double-double precision library. `http://www.nersc.gov/∼dhbailey/mpdist/mpdist.html`.

[7] J. Bilmes, K. Asanović, J. Demmel, D. Lam, and C.W. Chin. The PHiPAC WWW home page. `http://www.icsi.berkeley.edu/~bilmes/phipac`.

[8] L. S. Blackford, J. Choi, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, Philadelphia, 1997. 325 pages.

[9] J. Bleher, A. Roeder, and S. Rump. ACRITH: High-Accuracy Arithmetic – An advanced tool for numerical computation. In *Proceedings of the 7th Symposium on Computer Arithmetic*, Urbana, IL, June 4-6 1985. IEEE Computer Society Press.

[10] R. Brent. A Fortran multiple precision arithmetic package. *ACM Trans. Math. Soft.*, 4:57–70, 1978.

[11] K. Briggs. Doubledouble floating point arithmetic. `http://www-epidem.plantsci.cam.ac.uk/∼kbriggs/doubledouble.html`, 1998.

[12] ISO/IEC 9899:1999 Standard for the C programming language (C99). Jan 99 draft available at `http://anubis.dkuug.dk/JTC1/SC22/WG14/www/docs/n869`. Final version to be available from `http://www.iso.ch`, 1999.

[13] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.

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

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

[16] James W. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, 1997.

[17] I. Dhillon, G. Fann, and B. Parlett. Application of a new algorithm for the symmetric eigenproblem to computational quantum chemistry. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, March 1997.

[18] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* PhD thesis, University of California, Berkeley, California, May 1997.

[19] I. S. Dhillon. Current inverse iteration software can fail. *BIT*, 38(4):685–704, December 1998.

[20] J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK User's Guide.* SIAM, Philadelphia, PA, 1979.

[21] J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[22] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[23] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 20:345–357, 1983.

[24] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, Third edition, 1996.

[25] A. Gupta, M. Joshi, G. Karypis, and V. Kumar. PSPASES: A scalable parallel direct solver for sparse symmetric positive definite systems. `http://www.cs.umn.edu/~mjoshi/pspases`.

[26] G. Henry. Unix extended precision library for the pentium. `http://www.cs.utk.edu/∼ghenry/distrib/archive.htm`, 1998.

[27] N. J. Higham. *Accuracy and stability of numerical algorithms.* SIAM, Philadelphia, 1996.

[28] Macsyma Inc. *Macsyma Mathematics and System Reference Manual, 16th edition.* 1996. 589 pages.

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

[30] W. Kahan. Matlab's Loss is Nobody's Gain. `http://www.cs.berkeley.edu/∼wkahan/MxMulEps.pdf`, 1998.

[31] W. Kahan and J. D. Darcy. How Java's Floating-Point Hurts Everyone Everywhere. `http://www.cs.berkeley.edu/∼wkahan/JAVAhurt.pdf`, 1998.

[32] W. Kahan and M. Ivory. private communication, 1997.

[33] W. Kahan and E. LeBlanc. Anomalies in the IBM ACRITH package. In *Proceedings of the 7th Symposium on Computer Arithmetic*, Urbana, IL, June 4-6 1985. IEEE Computer Society Press.

[34] D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, Reading, MA, 1969.

[35] U. Kulisch and W. Miranker, editors. *A new approach to scientific computing.* Academic Press, New York, 1983.

[36] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[37] Xiaoye S. Li and James W. Demmel. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of SC98*, Orlando, Florida, November 1998.

[38] M4 macro processor. `http://www.cs.utah.edu/csinfo/texinfo/m4/m4.html`.

[39] O. Møller. Quasi double precision in floating-point arithmetic. *BIT*, 5:37–50, 1965.

[40] M. Monagan, K. Geddes, K. Heal, G. Labahn, and S. Vorkoetter. *Maple V Programming Guide for Release 5.* Springer-Verlag, 1997.

[41] W. Oettli and W. Prager. Compatibility of approximate solution of linear equations with given error bounds for coefficients and right hand sides. *Num. Math.*, 6:405–409, 1964.

[42] B. Parlett and I. Dhillon. Fernando's solution to Wilkinson's problem: An application of double factorization. *Lin. Alg. Appl.*, 267:247–279, 1997.

[43] B. Parlett and O. Marques. An implementation of the dqds algorithm (Positive case). *Lin. Alg. Appl.*, 309:217–259, 2000.

[44] M. Pichat. Correction d'une somme en arithmetique à virgule flottante. *Numer. Math.*, 19:400–406, 1972.

[45] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145, Grenoble, France, June 26-28 1991. IEEE Computer Society Press.

[46] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[47] Jonathan Richard Shewchuk. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[48] 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.

[49] K. Turner and H. F. Walker. Efficient high accuracy solutions with GMRES(m). *SIAM J. Sci. Stat. Comput.*, 13:815–825, 1992.

[50] R. C. Whaley and J. Dongarra. The ATLAS WWW home page. `http://www.netlib.org/atlas/`.

[51] S. Wolfram. *Mathematica: A system for doing mathematics by computer*. Addison-Wesley, 1988.

A Error bound for inner product

To derive the error bound for the inner product (9), we will use the standard model of floating-point arithmetic, $fl(a \odot b) = (a \odot b)(1+\delta) + \eta$, where $|\delta|$ is bounded by the machine precision ε and $|\eta|$ is bounded by the underflow threshold υ. This applies to real or complex arithmetic, with slightly different values of ε and υ as described in section 6.1. for our reference implementation on an IEEE machine. In addition, we will use the following notation.

ε_{out} = machine precision of the output precision
υ_{out} = underflow threshold in the output precision
ε_{int} = machine precision error of the internal precision claimed by the routine to be tested
υ_{int} = underflow threshold of the internal precision claimed by the routine to be tested
ε_{acc} = machine precision of our most accurate precision for computing r (Section 6.2.2)
υ_{acc} = underflow threshold of our most accurate precision
$S = |\alpha| \cdot \sum_{i=1,n} |x_i \cdot y_i| + |\beta \cdot r_{in}|$
r_{truth} = the correct answer in exact arithmetic
r_{comp} = r computed by the routine being tested, with the accumulation done in internal precision ε_{int}, and then rounding to the output precision ε_{out}
r_{acc} = the computed result using precision ε_{acc}

The usual error analysis says that the correctly computed and rounded result of

$$r_{truth} = \alpha \cdot (\sum_{i=1}^{n} x_i \cdot y_i) + \beta \cdot r_{in} \tag{26}$$

should satisfy

$$|r_{comp} - r_{truth}| \le (n+2) \cdot \varepsilon_{int} \cdot S + (2|\alpha| \cdot n + 3) \cdot \upsilon_{int} + \varepsilon_{out} \cdot |r_{truth}| + \upsilon_{out} \tag{27}$$

assuming that the parentheses are respected, i.e. that $\sum_{i=1}^{n} x_i \cdot y_i$ is computed before being multiplied by α. This takes into account all possible underflows in the accumulation and final rounding, including underflows on addition/subtraction, which would not occur with gradual underflow, only flush-to-zero. We cannot directly compare with this error bound, because we do not know r_{truth}. We only know r_{acc}, as described in Section 6.2.2, which satisfies

$$|r_{acc} - r_{truth}| \le (n+2) \cdot \varepsilon_{acc} \cdot S + (2|\alpha| \cdot n + 3) \cdot \upsilon_{acc} \tag{28}$$

Applying triangle inequality to (27) and (28), we get

$$|r_{comp} - r_{acc}| \le (n+2) \cdot (\varepsilon_{int} + \varepsilon_{acc}) \cdot S + (2|\alpha| \cdot n + 3) \cdot (\upsilon_{int} + \upsilon_{acc}) + \upsilon_{out} + \varepsilon_{out} \cdot |r_{truth}| \tag{29}$$

From error bound (28) we know

$$|r_{truth}| \le |r_{acc}| + (n+2) \cdot \varepsilon_{acc} \cdot S + (2|\alpha| \cdot n + 3) \cdot \upsilon_{acc} \tag{30}$$

Substituting (30) into (29) and omitting the second order terms ($\varepsilon_{out}\varepsilon_{acc}$ and $\varepsilon_{out}\upsilon_{acc}$), we obtain the final error bound that the test code can compute

$$|r_{comp} - r_{acc}| \le (n+2) \cdot (\varepsilon_{int} + \varepsilon_{acc}) \cdot S + (2|\alpha| \cdot n + 3) \cdot (\upsilon_{int} + \upsilon_{acc}) + \upsilon_{out} + \varepsilon_{out} \cdot |r_{acc}| \tag{31}$$

If instead of associating $\alpha \cdot (\sum_{i=1}^{n} x_i y_i)$ in the computation we use $\sum_{i=1}^{n} (\alpha \cdot x_i) \cdot y_i$, then the term $(2|\alpha| \cdot n + 3) \cdot (\upsilon_{int} + \upsilon_{acc})$ bounding the underflow error changes to

$$(\sum_{i=1}^{n} |y_i| + 2n + 1) \cdot (\upsilon_{int} + \upsilon_{acc}) \ \ . \tag{32}$$

Finally, if we intead associate as $\sum_{i=1}^{n} x_i \cdot (\alpha \cdot y_i)$, then the bound on error from underflow changes to

$$(\sum_{i=1}^{n} |x_i| + 2n + 1) \cdot (\upsilon_{int} + \upsilon_{acc}) \ \ . \tag{33}$$

Thus, the worst case underflow error for the three ways of inserting parentheses is bounded by

$$\max(2|\alpha| \cdot n + 3, \ \sum_{i=1}^{n} |y_i| + 2n + 1, \ \sum_{i=1}^{n} |x_i| + 2n + 1) \cdot (\upsilon_{int} + \upsilon_{acc}) \ \ . \tag{34}$$

B Smith's Complex Division Algorithm with Scaling

Let OV denote the overflow threshold and UN denote the underflow threshold, and ε denote the machine precision. Our scaled version of Smith's complex division algorithm is as follows, which computes $q = q_r + iq_i = \frac{a+ib}{c+id}$.

```
Let $ab = \max(|a|, |b|)$, $cd = \max(|c|, |d|)$.
$S = 1$
/* Scaling */
if $ab > OV/16$ then /* scale down $a, b$ */
    $a = a/16;\ b = b/16$
    $S = S \cdot 16$
endif
if $cd > OV/16$ then /* scale down $c, d$ */
    $c = c/16;\ d = d/16$
    $S = S/16$
endif
if $ab < \frac{UN}{\varepsilon}B$ then /* scale up $a, b$ */
    $a = a \cdot \frac{B}{\varepsilon^2};\ b = b \cdot \frac{B}{\varepsilon^2}$
    $S = S/(B/\varepsilon^2)$
endif
if $cd < \frac{UN}{\varepsilon}B$ then /* scale up $c, d$; $B$ is a small integer */
    $c = c \cdot \frac{B}{\varepsilon^2};\ d = d \cdot \frac{B}{\varepsilon^2}$
    $S = S/(B/\varepsilon^2)$
endif

/* Now $a, b, c, d \in (\frac{UN}{\varepsilon}B, OV/16)$ */
if ($|c| > |d|$) then
    $r = d/c$
    $t = \frac{1}{c+d \cdot r}$
    $q_r + iq_i = (a + b \cdot r) \cdot t + i(b - a \cdot r) \cdot t$
else
    $r = c/d$
    $t = \frac{1}{d+c \cdot r}$
    $q_r + iq_i = (b + a \cdot r) \cdot t + i(-a + b \cdot r) \cdot t$
endif

/* Scale back */
$q_r = q_r \cdot S;\ q_i = q_i \cdot S$
```

By careful error analysis, we can show that the error of this algorithm satisfies the bound:

$$fl(a/b) = (a/b)(1 + \delta) + \eta$$

where $|\delta| \leq (5 + 3\sqrt{2} + (2 + \sqrt{2})/B) \cdot \varepsilon$ and $|\eta| \leq \sqrt{2} \cdot \text{UN}$. In our implementation, we choose $B = 2$, so $|\delta| \leq (6 + 4\sqrt{2}) \cdot \varepsilon$.