SIAM J. SCI. STAT. COMPUT.
Vol. 5, No. 4, December 1984

010

UNDERFLOW AND THE RELIABILITY OF NUMERICAL SOFTWARE*

JAMES DEMMEL†

Abstract. We examine the effects of different underflow mechanisms on the reliability of numerical software. Software is considered reliable in the face of underflow if the effects of underflow are no worse than the uncertainty due to roundoff alone. The two primary underflow mechanisms discussed are store zero and gradual underflow, although we consider other mechanisms as well. By examining a variety of codes, including Gaussian elimination, polynomial evaluation, and eigenvalue calculation, we conclude that gradual underflow makes it significantly easier to write good numerical codes than store zero, and that this remains true even if extra range and precision are available for intermediate calculations.

Key words. software reliability, numerical software, roundoff, underflow, error analysis

1. Introduction and summary. In this paper we examine the effects of underflow on the reliability of codes for solving a wide variety of numerical problems. In particular we demonstrate the utility of gradual underflow for writing more robust codes than are usually written when the conventional "store zero" approach to underflow is used. This paper summarizes the work of several people over a period of several years during which they participated in the IEEE Floating Point Standard subcommittee's deliberations about the proper way to handle underflow. In addition to the author, these people are J. Coonen, D. Hough, W. Kahan and S. Linnainmaa. Some of the results presented here have been published (separately) before; others have not.

When we speak of reliable software, we mean software that ideally produces accurate results whenever they can be represented, and otherwise gives a warning. Needless to say, such software must cope with roundoff, and that may be difficult for many problems even in the absence of underflow. These unavoidable roundoff errors have led to diminished expectations and less stringent definitions of reliability for different kinds of codes. For example, a Gaussian elimination code to solve a system of linear equations is commonly called reliable if it delivers the exact solution of a problem close to the one it received as input (we will discuss this example in more detail below). Users have come to expect no more than these weaker forms of reliability from many of their codes because both experience and sometimes proofs have demonstrated that roundoff errors prevent better performance.

How much further must the notion of reliability be weakened in the face of underflow? For example, does Gaussian elimination still deliver the exact solution of a problem close to the input if underflows occur during the computation? If so, and in general, if we can show that the effects of underflow on a code are no worse than the uncertainty due to roundoff alone, then we consider that code no less reliable in the face of underflow than in the face of roundoff. Thus, our approach during our investigations has been to decide if underflow contributes nothing worse to a code than the uncertainty from the expected effects of roundoff errors which must be tolerated anyway.

To explain our approach and conclusions, we need some notation. A more complete discussion of the following terminology may be found in § 2 of this paper. We describe

* Received by the editors December 14, 1982, and in revised form May 5, 1983. This work was supported by the U.S. Department of Energy, contract DE-AM03-76SF00034, Project Agreement DE-AS03-79ER10358, the Office of Naval Research, contract N00014-76-C-0013, and IBM, contract 82007PLP0446. The author holds an IBM Fellowship. This paper contains an expanded version of "Effects of Underflow on Solving Linear Systems", by Dr. James Demmel, appearing in the Proceedings of the 5th Symposium on Computer Arithmetic, Ann Arbor, MI, May 18–19, 1981, pp. 113–120. Copyright © 1981 IEEE.

† Computer Science Division, University of California, Berkeley, California 94720.

floating point arithmetic with two parameters: ε and λ. ε denotes the difference between 1 and the next larger floating point number; thus ε bounds the rounding error in the operations +, −, * and /. λ denotes the underflow threshold, i.e. the smallest positive normalized floating point number. The two basic underflow mechanisms we have compared are store zero and gradual underflow. Store zero, the standard response to underflow, simply replaces any result that would be smaller than λ in magnitude by 0. Gradual underflow, on the other hand, returns an unnormalized floating point number less than λ in magnitude which approximates the tiny result. These unnormalized numbers form an arithmetic progression between 0 and λ with common separation $\lambda\varepsilon$, and are called *denormalized* to emphasize that they occur only at the bottom of the exponent range. Gradual underflow will henceforth be abbreviated by G.U. and store zero by S.Z.

There are actually many more mechanisms available to the system architect; all underflow mechanisms will be discussed further in § 2 below. For reasons also explained there we selected the following variations on G.U. and S.Z. for analysis in this paper:

We compared using the *same* precision and range for intermediate calculations as are used to represent the inputs and outputs with using *extra* precision and range for intermediate calculations.

We compared using gradual underflow with the underflow flag being set by a *threshold test* (which signals underflow whenever the result is denormalized) with using gradual underflow with the flag being set by an *accuracy test* (which signals underflow only if the denormalized result has a numerical value different from that of the correctly rounded result).

We compared using underflow flags which are *sticky* (which, once set, remain set until explicitly reset by the user) with underflow flags which are *nonsticky* (which are reset prior to each floating point operation).

We have compared the effects of these mechanisms on the robustness of codes written without attention to over/underflow problems, but we occasionally consider highly robust, expert codes as well.

Our main conclusions are given below:

(1) For many algorithms written without attention to over/underflow, only if G.U. is used instead of S.Z. is the algorithm as robust in the face of roundoff and underflow as it is with roundoff alone. More specifically, as long as the data is normalized ($>\lambda$ in magnitude) the results are as good as can be expected just with roundoff when using G.U., but when using S.Z. the data must be at least λ/ε to expect the same performance.

(2) For some computations, one can claim more than in (1). Suppose we measure backwards error in the following combined relative/absolute way:

$$\text{the change in } x \text{ is comparable to } \begin{cases} \varepsilon|x| & \text{if } |x| \geqq \lambda, \\ \varepsilon\lambda & \text{if } |x| < \lambda, \end{cases}$$

for G.U., and

$$\text{the change in } x \text{ is comparable to } \begin{cases} \varepsilon|x| & \text{if } |x| \geqq \lambda/\varepsilon, \\ \lambda & \text{if } |x| < \lambda/\varepsilon, \end{cases}$$

for S.Z. For G.U. this means the change in x is comparable to a few units in the last place stored of x, no matter if x is normalized or not. For S.Z., on the other hand, numbers near λ contain almost no significant digits. Then with respect to this new distance function, many algorithms always deliver the exact solution of a problem close to the input problem, no matter if underflow occurs or not. This statement is true of Gaussian elimination as long as the results themselves do not underflow and lose

accuracy, of polynomial evaluation, and of computing the eigenvalues of a symmetric tridiagonal matrix, for example. In other words, these algorithms *always* have a small backwards error with respect to this new definition (and subject to easily testable constraints), no matter what the inputs are. For G.U., this means nearly *every* bit stored in a number is significant, whereas in S.Z. almost no bits in any number of the problem may be significant, if all the numbers are too close to λ.

(3) In addition to extending the effective exponent range of the system by $-\log_2 \varepsilon$ as described in (1), G.U. preserves certain mathematical relationships (such as $x = y$ if and only if $\mathrm{fl}(x-y)=0$) over the *entire* range of floating point numbers. These relationships may occasionally fail with S.Z. Their failure can lead to strange and elusive bugs in codes (see § 4 below), whereas it is easier to write reliable code if these relationships can be depended on.

(4) Availability of extended precision *and* range does not always obviate the advantage of G.U. over S.Z. For some computations, such as polynomial evaluation, an extended format does eliminate almost all worry about intermediate over/underflow, but for others, such as Gaussian elimination and Cholesky decomposition, as long as the solution itself and the triangular factors of the matrix are stored in the basic format, the conclusions in (1) above remain valid even if all intermediate results are computed *exactly*. Thus, G.U. is of advantage to a system with an extended format as well as to a system with just one format.

(5) There are computations for which the accuracy test for G.U. is preferable to the threshold test and computations for which the threshold test is preferable, but the relative advantage is not very great for either type of test. The only advantage of the accuracy test over the threshold test we discovered was in the underflow flag being a false alarm less frequently. These potential false alarms arise from the assignment statement $a := b$ when b is denormalized, negation ($a := -b$ when b is denormalized), addition, subtraction, multiplication when one factor is an integer, and remainder ($a \bmod b$). The only potential advantage of the threshold test over the accuracy test was in helping to automatically verify the constraint that inputs be normalized ($> \lambda$ in magnitude) mentioned in (1) above. It was not clear that this could be used easily in practice (see the discussion of Gaussian elimination in § 8 below).

(6) The sticky underflow flag is much more useful than the nonsticky kind, although there are several applications of nonsticky flags in expert codes (see the discussion of Gaussian elimination below). The sticky flag can be used to simulate a nonsticky one at the cost of resetting it before each relevant operation, a cost which may be severe if resetting requires an expensive system call in a tight loop.

(7) Highly robust, expert codes for problems like polynomial root finding are easier to write using G.U. than S.Z. However, as soon as any scaling is done it is usually as easy to scale to avoid S.Z. underflows as G.U. (see the discussions of Gaussian elimination, Cholesky decomposition, and eigenvalue computations in [2]).

We believe that the evidence weighs clearly in favor of G.U. over S.Z. Presumably that is why gradual underflow is required by the proposed floating point standard.

The evidence shows neither the accuracy test for G.U. nor the threshold test to be uniformly superior to the other, but the choice depends on whether the floating point designer also has control over how the compilers implement assignment and negation statements (see § 5). If he does have control, he should insist on simple bit copying (nonfloating point) operations; if not, choosing the accuracy test over the threshold test eliminates the possibility of spurious underflow messages during assignment and negation. The proposed standard incorporates the accuracy test for lack of control over compiler design.

The sticky underflow flag is preferable to the nonsticky kind if there can be only one; a friendly system would make both available. The proposed standard requires sticky flags for all exception conditions, including underflow.

The rest of this paper is organized as follows. Section 2 presents underflow from a system architect's point of view. We discuss number formats and the options available for handling underflows, both when the underflow occurs and when the result is used later. Section 3 discusses underflow from a numerical analyst's point of view and shows how to extend conventional error analyses to include underflow. Sections 4 through 14 elaborate on the above results (without proofs) for the eleven computations listed below. Sections 4 through 14 may be read independently of one another:

tests and comparisons
the accuracy test versus the threshold test for G.U.
complex arithmetic
inner product calculations
Gaussian elimination
Cholesky decomposition
iterative refinement of linear systems
polynomial evaluation and root finding
eigenvalue computations for symmetric tridiagonal matrices
numerical quadrature
accelerating the convergence of sequences

2. A system architect's view of underflow. In this section we have two goals, first to describe the mechanisms available to the system architect for handling underflow, and second to describe the mechanisms we compare in this paper and why we have chosen them. We will introduce much notation in this section; when a new term is defined it will appear in *italics.*

The design questions facing the system architect are of two kinds: what value should be returned in the destination word when underflow occurs, and what side effects (if any) should underflow have? Options for the destination value are G.U., S.Z., and several other conventions such as exponent wraparound [10] and nonnumeric symbols like UN [4] and NAN [8]. Possible side effects are raising an underflow flag and continuing execution, invoking a trap handler that may execute any code of the system's or user's choice, waiting until an underflowed quantity is to be used to decide what to do, or most simply doing nothing. In case the architect decides to have flags or traps, the efficiency of his implementation will affect how the programmer writes codes to use the flags or traps (see the discussion of Gaussian elimination in § 5, for example). Other side effects arise from design decisions made in the compiler; these are discussed below and in §§ 4 and 5. We will first discuss the different values that can be returned from an underflowed operation, and then possible side effects.

To describe the values that can be returned we need to refer to a specific floating point format which we now describe (the conclusions of this paper apply to similar formats as well). It contains three fields: a *sign bit* σ, a *significand* f, and an *exponent* e, and represents the value $x = (-1)^{\sigma} \cdot f \cdot 2^{e}$. The exponent e satisfies $e_{\min} \leqq e \leqq e_{\max}$. The binary point follows the leading bit of f.

We call f (and the entire floating point number) *normalized* if its leading bit is 1 (or if $e = 0$ and $f = 0$, which represents 0). This means $1 \leqq f < 2$. Otherwise $0 \leqq f < 1$ and is called *unnormalized.*

The *rounding error* of the arithmetic is the largest possible value of

$$\frac{|\mathrm{fl}(a \blacksquare b) - a \blacksquare b|}{|a \blacksquare b|},$$

where ■ denotes one of the operations $\{+, -, *, /\}$, and a and b are such that $a \blacksquare b \neq 0$ and fl $(a \blacksquare b)$, which denotes the floating point result of the operation $a \blacksquare b$, is normalized and nonzero. As long as fl $(a \blacksquare b)$ is the first floating point number greater than or equal to $a \blacksquare b$ or the first number less than or equal to $a \blacksquare b$ (e.g. if the arithmetic truncates or rounds), then

$$\varepsilon \equiv 2^{1-n},$$

where n is the number of bits used to represent f, is a bound on the rounding error. In other words, ε is the difference between 1 and the next larger floating point number. Note that ε is twice as big as the rounding error if fl $(a \blacksquare b)$ is the nearest floating point number to the true result $a \blacksquare b$.

The largest normalized number has $e = e_{\max}$ and $f = 1.1 \cdots 1$ (n bits long); it is called the *overflow threshold* and denoted by

$$\Lambda \equiv 2^{e_{\max}}(2 - \varepsilon) \approx 2^{e_{\max}+1}.$$

The smallest normalized number, which has $e = e_{\min}$ and $f = 1$, is called the *underflow threshold*, and is denoted by

$$\lambda \equiv 2^{e_{\min}}.$$

Even though λ is called the underflow threshold, we will see that underflow might not always be signalled whenever a result is less than λ in magnitude.

When $e = e_{\min}$ and $f < 1$ we call the number *denormalized*. Denormalized numbers are also called *subnormal* [6], a name which is perhaps more descriptive than denormalized. The denormalized numbers, which are a subset of the unnormalized numbers, form an arithmetic progression between 0 and λ with common separation $\lambda\varepsilon$. Not all floating point systems allow denormalized numbers, or any unnormalized numbers at all. If denormalized numbers are not allowed, we typically handle underflow using *store zero* (S.Z.). This means that if the rounded value of a computation x would lie strictly between $\pm\lambda$ so that we could not represent it as a normalized nonzero number, we return zero. If denormalized numbers are allowed then we can use *gradual underflow* (G.U.), which means rounding such an x to the nearest denormalized number and returning that instead of zero. Gradual underflow is also called *graceful underflow* [6].

Exponent wraparound [10] is another possibility which only makes sense on a system which does not trap on over/underflow but which increments/decrements a counter designated in advance by the user (cf. Kahan's Counting Mode [10]). When a result would underflow, the value returned has the normalized significand of the result stored in f and the result's exponent biased upward by a constant (such as $-3 \cdot e_{\min}/2$) stored in e. (The analogous technique applies to overflow). By examining the counter the user can keep track of the powers of two contributed by wraparound.

Finally, the system may return a nonnumeric symbol such as UN [4] or NAN (*Not A Number*) [8]. A NAN is encoded in the IEEE proposal by an exponent $e = e_{\max}+1$ and a nonzero significand f that may contain or point to diagnostic information about where and when the underflow occurred. This technique allows the user to defer deciding what to do about an underflow until later when he has more information (this is discussed further below). For more detail on floating point formats and representing underflowed quantities see [1].

The architect also has many options for side effects. Side effects of underflow may be generated on two occasions: when an underflowed quantity is created, and later when it is used. First we describe creation time side effects and then use time side effects.

The creation time options are raising a flag/not raising one, trapping/not trapping, and doing nothing. Doing nothing is the most common response of systems today

because underflow is generally presumed to be harmless (were that true, this paper would not have been written).

One attribute a flag can have is "stickiness". An underflow flag is *sticky* if, once set, it remains set until explicitly reset by the user (as in the proposed standard); otherwise it is *nonsticky*, that is reset prior to each operation. A sticky flag is generally much more useful than a nonsticky one because it allows the user to ask if any underflows have occurred anywhere in a section of code (since the last time the flag was reset). This is the proper type of flag for debugging or when underflows are not anticipated. A nonsticky flag, which can always be simulated by a sticky one, is useful only when analysis has shown that underflow in only a certain few operations can matter. This is the case in certain expert codes (see § 8 below) but is rare.

Another attribute a flag can possess is available only with G.U.: it can be set either by a threshold test or an accuracy test:

Threshold test. Signal underflow if the exact result would have been less than λ in magnitude and not zero, and

Accuracy test. Signal underflow if, in addition to the computed result being no more than λ in magnitude, it is different from what would have been the result had exponent range been unbounded.

The reason for the option is as follows. Just because a result of an operation must be represented as a denormalized number does not mean accuracy has been lost. It may be that the error incurred by denormalization is no worse than what roundoff would have caused had exponent range been unlimited. For example, $\lambda/2$ is representable exactly as a denormalized number. In such cases, the architect may decide not to signal underflow, since the error is no worse than what roundoff alone would have caused. This more restrictive definition of underflow has the advantage of signalling underflow less frequently than the threshold test and therefore generates fewer false alarms. For example, the accuracy test will never signal underflow on copy (assigning $a := b$), negation ($a := -b$), addition, subtraction, multiplication where one factor is an integer, or remainder ($a \bmod b$) [16]. On the other hand, a threshold test may be better for an application where any nonzero result less than λ in magnitude causes problems later in the code. In the friendliest system, the user would be able to choose the definition depending on his application. For example, when debugging a new code in which underflow is not expected to occur, a threshold test with a smart trap handler/debugger would be useful, whereas a clever, robust code might exploit the more restrictive definition. We give examples of codes which use both types of flags below.

There are at least as many options available to the designer of a trap handler, because in principle a trap handler can contain any code of the system's or user's choice. For example, one may want a smart trap handler/debugger which lets the user examine his operands and code when underflow occurs, or one which keeps a record of where and when all underflows occur and lets the user examine them at the end of the program, or even one which attempts to perform the computation in a totally different way to avoid underflow. Actually, any given underflow mechanism can be implemented using a trap handler if a trap occurs on every underflow, although this may be slow. Obviously these possibilities involve compiler and operating system questions which would be difficult and interesting even without raising any numerical issues; we will not consider traps further in this paper.

Finally, the system (or user) can decide at the time of use what to do about underflow. This option is not available in an S.Z. system because there is nothing unusual about an underflowed S.Z. value (it is zero) that lets us detect when it is used:

with G.U., however, denormalized numbers mark themselves as underflowed quantities. By delaying a reaction until time of use, the user can defer judgement about the harmfulness or harmlessness of an underflow until he has more information available to help him decide. If a denormalized number is to be added to a much larger number, for example, little or nothing is lost. If it is to be multiplied by a large number, accuracy lost in denormalization might become significant later, especially if cancellation occurs. In general these decisions can better be made when the denormalized number is to be used rather than when it is created. Again, it is advantageous to give the user a choice in response. One approach considered by the IEEE committee was to have two modes: warning and normalizing. *Warning mode* caused a trap whenever

the uncertainty in a denormalized operand ($\pm\lambda\varepsilon/2$) would be magnified relative to the result by multiplication or division by a normalized operand, or
dividing a finite nonzero dividend by a denormalized divisor, or
taking the square root of a denormalized number.

Normalizing mode does not trap in these cases. As with the different definitions of underflow, warning mode may be useful for debugging new codes, and normalizing mode for writing clever, robust ones. We again give examples of such clever codes below. The committee chose not to include warning mode in the standard.

Given this bewildering array of options, how do we intend to compare G.U. and S.Z. systems? It is obviously possible to compute anything using S.Z. that can be computed with G.U. (and vice versa) by testing and scaling each pair of operands before use, but this is hardly a fair comparison since one code may be much harder to write or take much longer to run than the other. One fair comparison is to ask if for a given level of system support and given level of effort the code using G.U. has substantially different reliability than one using S.Z. For the comparisons in this paper, we chose the least effort possible, meaning that we want to compare codes written without regard to underflow at all if possible, or sight modifications of such codes. Furthermore, we chose the least possible system support short of doing nothing: providing a user testable underflow flag (and, of course, not trapping on underflow). We also consider the two ways to raise the G.U. flag described above: the threshold test and the accuracy test (in what follows we will often use the phrase "inaccurate underflows" to refer to both S.Z. underflows and G.U. underflows according to the accuracy test). Applications of nonsticky flags will be noted when they exist; unless a flag is explicitly called nonsticky it should be assumed sticky. In addition to these underflow options, we examine the utility of performing intermediate calculations with extra precision and range to avoid as many underflows as possible.

Finally, a writer of clever library routines may well be interested in how much reliability he can get for a fixed execution time, code size, etc., independent of development cost. We believe several of the codes discussed in this paper (and in more detail in [2]) will provide a basis for such a comparison.

3. A numerical analyst's view of underflow. In this section we show how to extend traditional floating point error analyses to take underflow into account. Let $\blacksquare$ be one of the operations $\{+, -, *, /\}$ and let $\mathrm{fl}(a \blacksquare b)$ denote the floating point result of the indicated operation. Traditional error analyses use the formula [23]

$$\mathrm{fl}(a \blacksquare b) = (a \blacksquare b)(1+e) \quad \text{unless } a \blacksquare b \text{ underflows or overflows,} \tag{1}$$

where $|e| \leqq \varepsilon$. To take underflow into ccount, we write [13]

$$\mathrm{fl}(a \blacksquare b) = (a \blacksquare b)(1+e) + \eta \quad \text{unless } a \blacksquare b \text{ overflows.} \tag{2}$$

In the case of G.U. there are the following constraints on e and η:

$$|e| \leqq \varepsilon \quad \text{and} \quad |\eta| \leqq \lambda\varepsilon, \tag{3}$$

$$\eta \cdot e = 0 \quad \text{(i.e. at most one of } \eta \text{ and } e \text{ is nonzero), and} \tag{4}$$

$$\eta = 0 \quad \text{if } \blacksquare \text{ is either addition or subtraction.} \tag{5}$$

In the case of S.Z. we have the following somewhat different constraints on e and η:

$$|e| \leqq \varepsilon \quad \text{and} \quad |\eta| \leqq \lambda, \quad \text{and} \tag{3'}$$

$$\eta \cdot e = 0. \tag{4'}$$

Let us examine the differences in constraints. The different bounds on $|\eta|$ in (3) and (3') mean that the error contributed by underflow for S.Z. can be $1/\varepsilon$ times as large as for G.U. (5) means that there can be no underflow error in addition or subtraction for G.U., whereas underflow can cause complete loss of relative accuracy for S.Z.

Formula (2) gives a combined relative/absolute error bound on the error in floating point. For G.U. we have a bound ε on the relative error as long as the true result is bigger than a threshold λ, and an absolute error bound $\lambda\varepsilon$ for smaller results. The bounds match, in that for results at the underflow threshold λ, the absolute magnitude of the largest relative error ($\varepsilon \cdot$ result) is equal to the largest absolute error ($\varepsilon \cdot \lambda$) (see Fig. 1). This property of (2) means that when doing a G.U. error analysis, we are really doing both a floating point and fixed point analysis simultaneously, because

$$\mathrm{fl}(a \blacksquare b) = (a \blacksquare b) + \eta$$

is the error formula used in fixed point analyses.

For S.Z. on the other hand, the error jumps at λ. For results just bigger than λ, the largest possible error is $\lambda\varepsilon$ as with G.U., but for smaller results the error leaps up to nearly λ (see Fig. 2). In order to analyze errors in S.Z. arithmetic as in G.U. (relative error above a threshold, absolute below, and at the threshold the errors match), we must raise the threshold to λ/ε (see Fig. 3). Said another way, G.U. reduces underflow errors to the size of roundoff for all normalized results, but S.Z. underflow errors are roundoff size only for results greater than λ/ε in magnitude. This explains why so many of the results to be presented later read as follows:

(6) When using G.U., as long as the data is normalized ($>\lambda$), the results are as good as can be expected just with roundoff, but when using S.Z. the data must be at least λ/ε to expect the same performance. Furthermore, as the data decreases below the threshold (λ or λ/ε) G.U.'s results degrade smoothly rather than abruptly, as do S.Z.'s.

λ is a much more natural threshold (and easier to test for, depending on the definition of underflow) than λ/ε for the range of application of a code.

For some codes one can make a backwards error bound independent of input values if one measure backwards error in the way suggested in conclusion (2) of § 1. For G.U., this measure means *every* number is viewed as uncertain in the last few places stored, whether denormalized or not. For S.Z., it means some numbers near λ are viewed as uncertain in nearly all their places. Gaussian elimination without inaccurate underflows in the solution components themselves, polynomial evaluation, and our algorithm for eigenvalues of symmetric tridiagonal matrices have backwards error bounds of this form. For these codes, G.U. more than extends the apparent exponent range by $-\log_2 \varepsilon$ over S.Z.: it asserts the significance of nearly all bits in every number in the machine.

Horizontal axis: True result of operation $a \blacksquare b$. (Tic marks represent floating point numbers.)
Vertical axis: —— Error in computed result $a \blacksquare b - \mathrm{fl}(a \blacksquare b)$. (Arithmetic is binary and chopped with $\varepsilon = \frac{1}{8}$ = maximum rounding error, λ = underflow threshold)
—— Error bound.

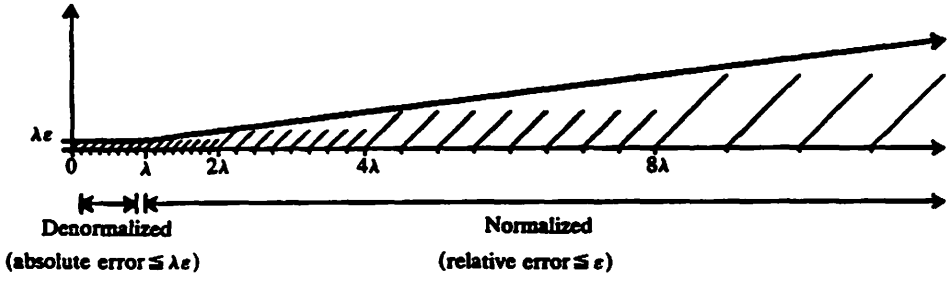

FIG. 1. *Error with gradual underflow (see (2), (3), (4) for error bound).*

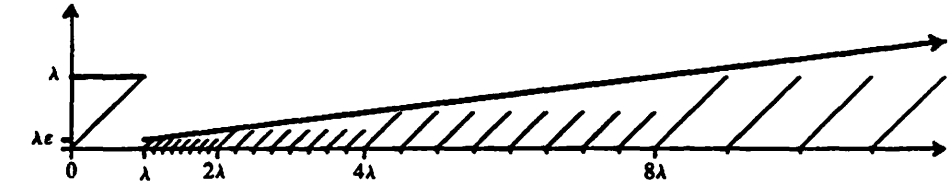

FIG. 2. *Error with store zero (see (2), (3'), (4') for error bound).*

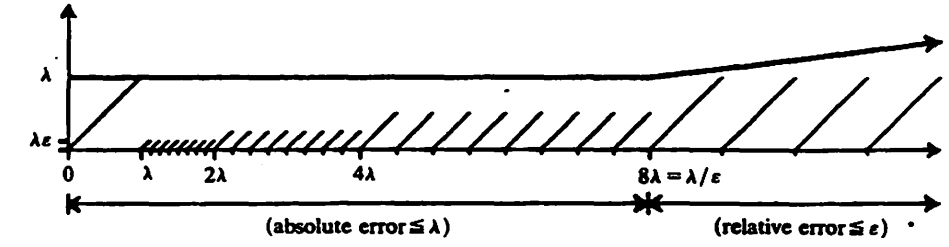

FIG. 3. *Store zero error bounded in same way as gradual underflow error.*

If all G.U. did were to extend the apparent exponent range of the system, then the argument for G.U. over S.Z. would become weaker as the actual exponent range grew larger. As we have just seen, however, there are certain mathematical relationships, preserved by G.U. but not S.Z. over the range of all floating point numbers, which make codes that are to work over the range of all inputs easier to write. Other useful relationships preserved by G.U. but occasionally violated by S.Z. include [1]:

(7) $\qquad x = y$ if and only if $\mathrm{fl}(x-y) = 0$,

(8) $\mathrm{fl}((x-y)+y) \approx x$ (to within a rounding error in the larger of x and y),

and assuming the exponent range $[e_{\min}, e_{\max}]$ is nearly symmetrical about 0 (as with the proposed IEEE standard), then if no overflow occurs

(9) $\qquad \mathrm{fl}(1/(1/x)) \approx x$ to within a few rounding errors in x.

Failure to satisfy statements like (7) to (9) can induce strange and elusive bugs in

codes (see § 4 and [10]). Their validity makes it much easier to write and maintain codes by eliminating the need for tests for the very rare cirumstances in which they fail.

The combined relative/absolute error measure given in (2) arises naturally in several ways. When solving linear equations with iterative refinement, we stop when the relative error in the solution vector is (hopefully) small. This means large components are known to high relative accuracy, and small components to an absolute accuracy of the same magnitude. In physical problems there is often a noise level which means that only measurements above it can be made relatively accurately, and below it only with absolute accuracy equal to the noise level.

4. Tests and comparisons. To analyze codes containing tests like

$$\text{(10)}\qquad \begin{array}{l}\text{if } x \neq y \quad \text{then } r := \dfrac{f(x)-f(y)}{x-y} \\ \qquad\qquad \text{else if } 100*x \neq 100*y \text{ then print } \textit{why}?\end{array}$$

or

$$\text{(11)}\qquad \text{if } x \neq 0 \text{ and } |x-y| \leqq .001|x| \text{ then } z = \mathrm{SQRT}(1.5 - y/x)$$

the first of which can produce a divide by zero error and the second of which a square root of negative number error, we must not only know how underflow is handled, but how the compiler implements tests like "$x \neq y$?". There are two possibilities for this: a fixed point (bitwise) comparison of x and y, and a comparison of $\mathrm{fl}(x-y)$ with 0.

Let us first analyze (10) and (11) using S.Z. With the first (fixed point) implementation of "$x \neq y$?", any choice of x and y such that $0 < |x-y| < \lambda$ (such as $x = 1.25\lambda$ and $y = 2\lambda$) will pass the test "$x \neq y$?" and cause a divide by zero error in the expression for r in (10). In (11), the same choice of x and y passes both tests but causes $1.5 - y/x$ to equal $-.1$ and gives a square root of negative number error. Using the second, floating point implementation of "$x \neq y$?" the same x and y causes *why*? to be printed by (10). Thus, both implementations and even the more robust looking test in (11) can cause strange results using S.Z.

With G.U., on the other hand, the two implementations of the test "$x \neq y$?" are equivalent (barring overflow of $\mathrm{fl}(x-y)$), and neither divide by zero nor *why*? nor square root of negative number messages are possible from (10) or (11). Any underflow flags raised by the threshold test should be ignored in these examples because if an addition or subtraction underflows in G.U. arithmetic, it must give the exact result (thus no underflow flag would be raised with the accuracy test).

The pitfalls of using extended range and precision in comparisons have been well documented in [15].

5. The accuracy test versus the threshold test for G.U. When an operation $a \blacksquare b$ underflows, the denormalized result need not have a different numerical value from the result that would have been returned had the exponent range been unbounded. For example, the results of $\lambda/2, \lambda/4, \cdots, \lambda/(1/\varepsilon)$ are all denormalized yet representable without error. The accuracy test for G.U. will not raise the underflow flag for these operations, or for any others where the denormalized result is identical to the result that would have been returned had the exponent range been unbounded. In contrast, the threshold test raises an underflow flag whenever a nonzero result is less than λ in magnitude (there are slight variations possible on this definition, but they do not effect the results of this analysis).

The accuracy test has the advantage over the threshold test, that if the *only* bad effect of underflow is its abnormally large loss of accuracy, then it avoids raising the

underflow flag unnecessarily, whereas the threshold test raises the flag whenever the result is small even if accurate. If, on the other hand, it is the *size* of an underflowed result that can cause difficulty later, the threshold test is more useful. We have found examples where both definitions of underflow are useful.

First we discuss examples where the threshold test appears advantageous. In conclusion (1) of the § 1, we stated that for many algorithms as long as the inputs were normalized ($>\lambda$ in magnitude), they would perform as well as expected with roundoff. This seems like an ideal use for the threshold test, but as described in the section on Gaussian elimination, for example, what we need to test is if *any* entry of the input matrix is normalized, a weaker condition on the matrix, but one requiring testing the underflow flag (and resetting it as well if it is a sticky flag) for each matrix entry. If testing, or more likely resetting, involves an expensive system call, we would not want to include it in such a tight loop. Similar input constraints apply to Cholesky, iterative refinement, inner product calculations and others: we would need to test and possibly reset the underflow flag in a tight loop. If these are expensive operations, the usefulness of the threshold test is undermined. Furthermore, some of these codes satisfy a combined relative/absolute error bound independent of the input values (see conclusion (2) of § 1).

Now we discuss the situations in which the accuracy test appears more useful. In Gaussian elimination, iterative refinement, and complex divide we may use the accuracy test to test intermediate and final results for underflows we know can be harmful only if they are inaccurate. There are also the simple assignment statement $a := b$ and negation $a := -b$. If b is denormalized, *and the compiler implements these statements as floating point operations*, then the accuracy test will raise no flag, but the threshold test will. If they are implemented as fixed point operations, then of course no flags will be raised, but in the unhappily common situation where one designer designs the floating point and another the compiler, the floating point designer may have no control over the compiler design decisions. One may counter that one could just test and reset the underflow flag after assignments and negations, but if this incurs the overhead of a system call, it may not be a good solution. These examples of assignment and negation may well be the major contributor of false alarms on threshold underflow.

6. Complex arithmetic. In order to make error analysis in complex arithmetic as similar as possible to the analysis in real arithmetic, we would like to have formulas describing the error in complex addition, subtraction, multiplication and division that are nearly identical to (1) to (5) and (3′) and (4′) which describe the error in real arithmetic.

6.1. Complex addition and subtraction. Here the situation is most satisfying: formulas (1) to (5) and (3′) and (4′) all remain true as long as "$a \blacksquare b$ overflows" is interpreted as "overflows in either component". We repeat these formulas for completeness. In the absence of overflow or underflow we have

$$\text{fl}(a \pm b) = (a \pm b)(1+e) \quad \text{unless } a \pm b \text{ underflows or overflows.} \tag{12}$$

To take underflow into account, we write

$$\text{fl}(a \pm b) = (a \pm b)(1+e) + \eta \quad \text{unless } a \pm b \text{ overflows.} \tag{13}$$

In the case of G.U. there are the following constraints on e and η:

$$|e| \leqq \epsilon \quad \text{and} \quad \eta = 0. \tag{14}$$

In the case of S.Z. we have the following somewhat different constraints on e and η:

$$|e| \leqq \varepsilon \quad \text{and} \quad |\eta| \leqq \lambda. \tag{15}$$

It is not true that at most one of e and η can be nonzero, as it was with real addition.

6.2. Complex multiplication. Multiplication is not quite so satisfactory as addition and subtraction because of the possibility of intermediate overflow in the obvious algorithm:

$$(a+i\cdot b)\cdot(c+i\cdot d)=(ac-bd)+i\cdot(ad+bc)\equiv p_r+i\cdot p_i, \tag{16}$$

even though the final product may be a representable number. Since this can only happen if one of p_r or p_i is within a factor of 2 of the overflow threshold Λ anyway, we accept this slight loss of robustness since formula (16) is otherwise so satisfactory, as we now discuss.

In the absence of overflow or underflow (in the intermediate or final results)

$$\mathrm{fl}(a*b)=(a*b)(1+e) \tag{17}$$

where a, b, and e are all complex quantities, and $|e|<2\sqrt{2}\varepsilon$. To take underflow into account we again write

$$\mathrm{fl}(a*b)=(a*b)(1+e)+\eta \quad \text{in the absence of overflow.} \tag{18}$$

For G.U. we have the following constraints on e and η (to first order in ε):

$$|e| \leqq 2\sqrt{2}\varepsilon \quad \text{and} \quad |\eta| \leqq 2\sqrt{2}\lambda\varepsilon. \tag{19}$$

For S.Z. we have the following slightly different constraints:

$$|e| \leqq 2\sqrt{2}\varepsilon \quad \text{and} \quad |\eta| \leqq 2\sqrt{2}\lambda. \tag{20}$$

Thus, complex multiplication can be analyzed in the identical way as real multiplication but with slightly larger bounds on e and η.

Hence, analyses of algorithms which use only $+$, $-$, and $*$ operations (such as inner product) and the error bounds in (2) extend immediately to the complex case.

It is no longer possible to test for underflow in multiplication with S.Z. by comparing the product to zero as in real multiplication. Indeed, it is possible for a nonzero product computed with S.Z. to be wrong in the second bit in both components due to underflow. For example, consider the product of $2\sqrt{\lambda}+i\cdot 0.5\sqrt{\lambda}$ and $\sqrt{\lambda}+i\cdot\sqrt{\lambda}$. The correct product, produced with G.U., is $1.5\lambda+i\cdot 2.5\lambda$, but S.Z. delivers $2\lambda+i\cdot 2\lambda$. The underflow flag, however, may also be raised spuriously, for S.Z. or G.U., accuracy test or threshold test, even though the product is exemplary.

6.3. Complex division. This case was originally analyzed by Hough [7]. The algorithm is due to Smith and can be found in Knuth [17, p. 195] and avoids almost all unnecessary intermediate overflows in the calculation. We want to compute the quotient $(a+i\cdot b)/(c+i\cdot d)=q_r+i\cdot q_i$:

$$\begin{aligned} &\text{if } |d|<|c| \text{ then compute } q_r+i\cdot q_i=\frac{a+b(d/c)}{c+d(d/c)}+i\cdot\frac{b-a(d/c)}{c+d(d/c)},\\ &\text{else compute } q_r+i\cdot q_i=\frac{b+a(c/d)}{d+c(c/d)}+i\cdot\frac{-a+b(c/d)}{d+c(c/d)}. \end{aligned} \tag{21}$$

As with complex multiplication, it is possible to have intermediate overflows even if q_r and q_i are exactly representable, but this can only happen if either the a and b or c and d are both within a factor of 2 of Λ anyway.

If no overflows or underflows occur, then the relative error in the quotient is bounded by $7\sqrt{2}\varepsilon$, where ε is the error in the underlying arithmetic. In contrast to addition, subtraction and multiplication, however, it is not possible to bound the error in the presence of underflow simply in terms of a few units in the last place of the correct result plus a few underflow errors. If either the dividend $a+i\cdot b$ or divisor $c+i\cdot d$ is entirely denormalized, it is possible to get a normalized quotient that may be wrong in most of its places. If both dividend and divisor are normalized in at least one component, however, then with G.U. the computed quotient does indeed agree with the correct quotient to all but a few units in the last place of $|q_r+i\cdot q_i|$. With S.Z. both divisor and divided have to be at least λ/ε to be assured of the same accuracy. We write these conclusions as follows:

(22) $\mathrm{fl}(a/b)=(a/b)^*(1+e)+\eta$ if both $|a|$ and $|b|$ are bigger than τ

where

(23) $|e|\leqq 7\sqrt{2}\varepsilon$ for both G.U. and S.Z.

and

(24) $\tau=\lambda$ and $\eta=\sqrt{2}\lambda\varepsilon$ for G.U.

and

(25) $\tau=\lambda/\varepsilon$ and $\eta=\sqrt{2}\lambda$ for S.Z.

Thus, when analyzing algorithms with complex division, more care must be taken than with real division to make sure the constraints given by τ above are satisfied.

Here are some examples to show what happens when the constraints given by τ are violated. We use 6 decimal arithmetic for ease of presentation. First, let $a+i\cdot b=2\lambda+i\cdot 1\lambda$ and $c+i\cdot d=4\lambda+i\cdot 2\lambda$. The correct quotient $(a+i\cdot b)/(c+i\cdot d)=.5$, but in S.Z. the term $\lambda(1/2)$ underflows to 0 and we get the quotient .4 instead of .5. With G.U. we get .5. If we now multiply both dividend and divisor by ε so they are denormalized, G.U. suffers the same fate as S.Z. and delivers .4 instead of .5.

Unfortunately, an underflow flag may be raised even though the product is very accurate. This is true for S.Z. or G.U. with either accuracy test or threshold test.

With extended precision *and* range both the multiplication and division routines can underflow (or overflow) only when storing the final results, thus avoiding all false alarms.

7. Inner product calculations. Consider the two vectors $a=(\Lambda, \lambda, 1/2, \lambda, 0)$ and $b=(0, 1/2, \lambda, 1, \Lambda)$. If we compute their inner product $\sum_{i=1,5} a_i b_i$ in the straightforward way

```
sum := 0
for i := 1 to 5 do sum := sum + a_i * b_i
```

we get very different answers if we use G.U. than if we use S.Z. With G.U. we get the exact answers 2λ whereas with S.Z. we get λ because both a_2b_2 and a_3b_3 are less than λ and so flush to zero in a S.Z. system. The difference is large in the forward sense (λ is relatively much different than 2λ) and the backward sense as well, because it cannot be explained by saying that the result obtained from S.Z. is the exact result of a different inner product whose vector components differ from the original ones by a few units in the last place. Note also that there are no scale factors α and β such

that the inner product can be calculated as

$$\frac{1}{\alpha\beta}\sum_{i=1}^{5}(\alpha a_i)(\beta b)_i \qquad (30)$$

without underflow or overflow.

We can state the following propositions about inner products which generalize the above example [2].

PROPOSITION 1. *Let g' be a bound on the partial sums and individual terms of the inner product $\sum_{i=1}^{n} a_i b_i$:*

$$g' = \max_{1\le i\le n}\left(\mathrm{fl}\left(\sum_{j=1}^{i} a_j b_j\right), a_i b_i\right). \qquad (26)$$

We can bound the error in computing $\sum_{i=1}^{n} a_i b_i$ as follows: In the absence of underflow we have

$$\left|\mathrm{fl}\left(\sum_{i=1}^{n} a_i b_i\right) - \sum_{i=1}^{n} a_i b_i\right| \le (2n-1)\varepsilon g \qquad (27)$$

where $g = g'/(1-\varepsilon)$.

In the case of G.U. we have

$$\left|\mathrm{fl}\left(\sum_{i=1}^{n} a_i b_i\right) - \sum_{i=1}^{n} a_i b_i\right| \le (n-1)\varepsilon g + n\varepsilon \max(\lambda, g) \le (2n-1)\varepsilon g \quad \text{if } g \ge \lambda \qquad (28)$$

where $g = g'/(1-\varepsilon)$.

In the case of S.Z. we have

$$\left|\mathrm{fl}\left(\sum_{i=1}^{n} a_i b_i\right) - \sum_{i=1}^{n} a_i b_i\right| \le (2n-1)\varepsilon \max\left(\frac{\lambda}{\varepsilon}, g\right) \le (2n-1)\varepsilon g \quad \text{if } g > \frac{\lambda}{\varepsilon} \qquad (29)$$

where $g = (g'+\lambda)/(1-\varepsilon)$. Note that the g used in equation (28) may differ from the g used in equation (29) because g depends on the kind of arithmetic used (G.U. or S.Z.). Also, g depends on the order of the terms $a_i b_i$.

The proof is a straightforward extension of the usual error analysis of inner products [23] using formula (2) of § 3.

The significance of this proposition is the following: (27) states the well-known result that the error in an inner product subject only to roundoff errors can be as large as about $2n$ rounding errors in the largest intermediate result g'. The second line of (28) says that the same is true for G.U. as long as the largest intermediate results g' is normalized. In particular, if the final result is normalized, then underflow is no worse than roundoff. (If g' is not normalized, then we have effectively computed the inner product in fixed point and we get only an absolute error bound from the first line of (28) as expected.) If we use the accuracy test with G.U. and the underflow flag is not raised, then (27) holds independent of the size of g. For S.Z. on the other hand (29) says that g' must exceed λ/ε for the same claim to hold. This is an example of statement (9) in § 3.

To analyze the backwards error in an inner product, we need another expression for the error.

PROPOSITION 2. *The floating point result of the inner product* $\sum_{i=1}^{n} a_i b_i$ *may be written*

$$\text{fl}\left(\sum_{i=1}^{n} a_i b_i\right) = \sum_{i=1}^{n} a_i b_i(1+E_i)+\eta. \tag{30}$$

In the absence of underflow we have

$$\begin{aligned} |E_1| &\leqq n\varepsilon, \\ |E_j| &\leqq (n+2-j)\varepsilon \quad \text{if } j>1. \end{aligned} \tag{31}$$

In the case of G.U. we have the same bounds on the $|E_i|$, and

$$|\eta| \leqq n\lambda\varepsilon. \tag{32}$$

In the case of S.Z. we have the same bounds on the $|E_i|$, and

$$|\eta| \leqq n\lambda. \tag{33}$$

The proof is again a straightforward extension of the usual error analysis [23] using formula (2).

(31) means that in the absence of underflow, an inner product can be computed with small backwards error; in other words the computed result is the exact inner product of two vectors whose components differ by at most n rounding errors from the components of the original vectors. (32) means that with G.U., as long as some intermediate result fl $(a_i b_i)$ is normalized ($>\lambda$), then the backwards error is also small, because η can be absorbed into the $a_i b_i(1+E_i)$ term, increasing E_i by at most $n\varepsilon$. In particular, if the final result is normalized, underflow is no worse than round off. (33) means that some intermediate term must be as large as λ/ε for a similar claim to hold for S.Z.

Of course, if we are using the accuracy test with G.U. and no flag is raised, then $\eta = 0$ and the roundoff only error bounds in (31) hold.

These two propositions may be used to extend the results of error analyses for many matrix computations to include underflow. The next three sections present the results of such analyses for Gaussian elimination, Cholesky decomposition, and iterative refinement.

8. Gaussian elimination.

8.1. Summary. The algorithm we analyze for solving the system of linear equations $Ax = b$ is a standard form of Gaussian elimination:

(1) Decompose $A = LU =$ (lower triangular) (upper triangular) using pivoting, so that the diagonal of L contains all 1's and no entries of L exceed 1 in absolute value;

(2) Solve $Ly = b$ for y (forward substitution);

(3) Solve $Ux = y$ for x (back substitution).

What kind of reliability do we expect from this algorithm in the absence of underflow? It is well-known that even though we can not expect an accurate solution if the input matrix is ill-conditioned, we can expect to get a residual $A\hat{x} - b$ ($\hat{x}$ is the computed solution) that is small in a sense made precise later. We also expect a small backwards error: $\hat{x}$ will be the exact solution of a problem slightly different from the original, again in a sense to be made precise later.

It turns out that as long as *one* component each of the matrix A and right-hand side b are normalized, then the only gradual underflows that can possibly contribute significantly to the residual or backwards error are inaccurate underflows in the final solution vector $\hat{x}$. Here we are using the accuracy test for underflow (see § 3), but our conclusions are also valid with the threshold test, though we get more false alarms.

This is a situation where the proper choice of underflow test depends on the application: if the output of the Gaussian elimination routine is input for another call to it, the user may choose the threshold test to see if he is passing normalized data to the second call as required for the conclusions above to apply. This may not be easy to do in practice, of course, but it shows that the accuracy test might not be best for all situations.

In contrast, unless one component each of the A and b is greater than λ/ε in magnitude, intermediate underflows with S.Z. during any stage of solution can introduce significant errors, possibly producing reasonable looking results whose error greatly exceeds the uncertainty attributable to roundoff alone (see the examples).

The measure of backwards error on which these conclusions depend is the following: in trying to solve $Ax = b$ we really compute $\hat{x}$ where $(A+\delta A)\hat{x} = b+\delta b$ and

(34a) $\|\delta A\|$ is comparable to $\varepsilon \cdot \|A\|$

and

(34b) $\|\delta b\|$ is comparable to $\varepsilon \cdot \|b\|$.

$\|A\|$ is a measure of the size of A (similarly for $\|b\|$), and "comparable to" means not larger by more than a factor $f(n)$ which is a low order polynomial in the dimension n of A (this will be made more explicit later). In other words, under the conditions stated above, Gaussian elimination has an error no larger than $f(n)$ rounding errors in the largest entry of A or b.

If we weaken our measure of backwards error in (34) and ask how much larger $\|\delta A\|$ ($\|\delta b\|$) can be than $\varepsilon\|A\|+\varepsilon\lambda$ ($\varepsilon\|b\|+\varepsilon\lambda$) instead of $\varepsilon\|A\|$ ($\varepsilon\|b\|$), then as long as the solution $\hat{x}$ itself does not underflow inaccurately, we can prove that Gaussian elimination using G.U. always has a small backwards error no matter how big $\|A\|$ or $\|b\|$ is. In other words, δA changes A (δb changes b) in the last few places of the largest entry, no matter if the largest entry is normalized or not. This robustness is not shared by S.Z.: almost all the bits in all the entries of A or b can be insignificant using S.Z. if the entries are too close to λ in size.

Gaussian elimination using G.U. with a warning of (inaccurate) underflows in the solution $\hat{x}$ appears to be a robust enough program to deserve inclusion in a library. If we insist on the traditional measure of backwards error in (34a) and (34b) above, and if we are willing to include an explicit scaling test ("are the largest entries of A and b at least λ in magnitude?") then G.U. offers no great advantage over S.Z. because changing the test threshold from λ to λ/ε makes an equally ironclad program in S.Z. with an only slightly smaller range of application, and eliminates the largest potential advantage of G.U.: making robust code faster to execute or easier to write. Note that we want to test whether *any* input component of A and b exceeds the threshold λ, and whether *any* output component of $\hat{x}$ underflows inaccurately. The most efficient test on the inputs would use a nonsticky flag based on the threshold test, since a sticky flag would have to be reset after each entry was tested. The most efficient test on the outputs would also use a nonsticky flag but be based on the accuracy test instead. Since each component $\hat{x}_i$ of $\hat{x}$ is computed in a loop with other computations, a sticky flag would have to be reset within the loop just before the last operation yielding $\hat{x}_i$.

It is very important to point out that use of extended range and precision for intermediate results does *not* invalidate the results just discussed. As long as the entries of L and $\hat{x}$ must be stored in the basic format the conclusions remain valid, because it is possible underflows in these entries that undermine the code's reliability. Thus, the conclusions of this section are as relevant to a system with just one precision and range available as to one with extended precision and range.

Section 8.2 contains examples and § 8.3 presents the theorems and offers conclusions. This material has been published before [3].

8.2. Examples of Gaussian elimination. $\|A\|_\infty(\|b\|_\infty)$ denotes the infinity norm of the matrix A (vector b):

$$\|A\|_\infty = \max_i \sum_j |A_{ij}| \quad \text{and} \quad \|b\|_\infty = \max_i |b_i|.$$

$|A|$ ($|b|$) denotes the matrix (vector) whose entries are the absolute values of the entries of $A(b)$. Inequalities like $|A| < |B|$ are meant componentwise.

We denote the usual condition number of the matrix A by

$$k(A) = \|A_\infty\| \cdot \|A^{-1}\|_\infty,$$

and a new set of condition numbers by

$$\text{Cond}\,(A, x) = \frac{\||A^{-1}||A||x|\|_\infty}{\|x\|_\infty}$$

$$\text{Cond}\,(A) = \||A^{-1}||A|\|_\infty.$$

These new condition numbers, due to Skeel [20], will be discussed more fully below. Note Cond $(A) \geqq$ Cond (A, x) for all x.

In this section we present four examples of the effects of underflow on performing Gaussian elimination. The first example shows how store zero can produce a reasonable looking but completely inaccurate decomposition of a well conditioned matrix, whereas gradual underflow either produces the correct decomposition or correctly decides the matrix is singular. (There are no rounding errors nor pivot growth in this example.) The second example shows that G.U. produces the correct decomposition of a well conditioned matrix which S.Z. incorrectly decides is singular. Third, we present an innocuous looking ordinary differential equation and show that the linear system arising from trying to solve it numerically leads to underflow which is handled correctly by G.U. and not by S.Z. Finally, we present an example which shows that regardless of whether we use G.U. or S.Z., Gaussian elimination can only guarantee small residuals, not an accurate answer, even when the matrix A is well conditioned in the sense that Cond (A) is small.

8.2.1. *Example* 1. Consider the family of matrices $A(x)$ where

$$A(x) = \lambda \cdot \begin{bmatrix} 2 & & & & 1 \\ & 2 & & & 1 \\ & & 2 & & 1 \\ & & & 2 & 1 \\ 1 & 1 & 1 & 1 & x \end{bmatrix} \tag{35}$$

(blanks denote zero entries). The LU decomposition obtained by G.U. is

$$L^{\text{G.U.}}(x) \cdot U^{\text{G.U.}}(x) = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ .5 & .5 & .5 & .5 & 1 \end{bmatrix} \cdot \lambda \cdot \begin{bmatrix} 2 & & & & 1 \\ & 2 & & & 1 \\ & & 2 & & 1 \\ & & & 2 & 1 \\ & & & & x-2 \end{bmatrix} = A(x) \tag{36}$$

exactly, and by S.Z. is

$$(37)\quad L_{S.Z.}(x)\cdot U^{S.Z.}(x)=\begin{bmatrix}1&&&&\\&1&&&\\&&1&&\\&&&1&\\.5&.5&.5&.5&1\end{bmatrix}\cdot\lambda\cdot\begin{bmatrix}2&&&&1\\&2&&&1\\&&2&&1\\&&&2&1\\&&&&x\end{bmatrix}=A(x)+E,$$

where the error matrix E equals

$$(38)\qquad E=\lambda\cdot\begin{bmatrix}&&&&\\&&&&\\&&&&\\&&&&\\&&&&-2\end{bmatrix}.$$

We see S.Z. causes a relatively large error in the $U(x)_{55}$ entry, whereas G.U. gives the correct decomposition. When $x=2$, using S.Z. leads us to conclude that the matrix is far from singular, when in fact it is exactly singular. Note that the matrix $A(x)$ is well conditioned when x is far from 2, and if x is a smaller integer no rounding errors occur in either decomposition.

8.2.2. *Example* 2. Let

$$(39)\qquad A=\begin{bmatrix}2\lambda&3\lambda\\\lambda&2\lambda\end{bmatrix},$$

a well conditioned matrix. Using G.U. we obtain

$$(40)\qquad L^{G.U.}\cdot U^{G.U.}=\begin{bmatrix}1&\\.5&1\end{bmatrix}\cdot\begin{bmatrix}2\lambda&3\lambda\\&\lambda/2\end{bmatrix}=A,$$

but by using S.Z. we obtain

$$(41)\qquad L^{S.Z.}\cdot U^{S.Z.}=\begin{bmatrix}1&\\.5&1\end{bmatrix}\cdot\begin{bmatrix}2\lambda&3\lambda\\&0\end{bmatrix}.$$

Thus, G.U. correctly decomposes the matrix A, whereas S.Z. incorrectly makes the matrix look singular.

8.2.3. *Example* 3. Consider the ordinary differential equation

$$(42)\qquad \dot{x}(t)=\frac{1-(t/T)^M}{T-t}x(t),\qquad x(T_0)=c.$$

We try to solve this equation numerically by replacing $x(t)$ by the truncated power series $\sum_{n=1}^{N}x_nt^n$, the function $(1-(t/T)^N)/(T-t)$ by its (finite) power series, and then equating coefficients of equal powers of t on both sides of equation (42). After we scale the last row (which represents the initial condition) down to have the largest entry equal to 1, we get the linear system $Ax=b$, where

$$(43)\qquad A=\begin{bmatrix}N&-1/T&-1/T^2&\cdots&&-1/T^N\\&N-1&-1/T&\cdots&&-1/T^{N-1}\\&&N-2&\cdots&&-1/T^{N-2}\\&&&\ddots&&\vdots\\&&&&1&-1/T\\1&1/T_0&1/T_0^2&\cdots&1/T_0^{N-1}&1/T_0^N\end{bmatrix},$$

Finally, if we swapped f with h at the start, we must remember to swap c_L with s_R and s_L with c_R.

Our Program

Considering how complicated it was to figure out, our program is surprisingly short. It is presented here in a syntax like that of Fortran 77, but with two innovations. One is the invocation of an intrinsic procedure SWAP(x,y) that swaps the values of its arguments. On a machine that contains a SWAP instruction in its hardware, this should be preferable to the three MOVES that would be needed instead. The second innovation is the use of three consecutive dots (...) to introduce a comment at the end of a line rather than have to add a line beginning with "C" for every short annotation.

```
      SUBROUTINE SVD2x2(f,g,h,cL,sL,w,v,cR,sR)
c     Accurate singular value decomposition of a given 2x2 real
c     matrix: ( cL  sL ) . ( f  g ) . ( cR  -sR ) = ( ±w  0 ) ,
c             ( -sL cL )   ( 0  h )   ( sR   cR )   ( 0  ±v )
c     with cL*cL+sL*sL = cR*cR+sR*sR = 1 and w .GE. v .GE 0 .
      REAL    f,g,h,                  cL,sL,w,v,cR,sR
c             -- Input --             -- Output --    Aliasing is OK
c     w and v are the singular values; the c's and s's define
c     the singular vectors of the given matrix.  In the special
c     case g = 0 , we get cL = cR = 1 and sL = sR = 0 .  In
c     the special case h = 0 , we get cL = 1 and sL = 0 .
      LOGICAL L
      REAL ft,gt,ht, cLt,sLt, cRt,sRt          ... Copied and scratch values
      REAL fa,ga,ha                            ... may be kept in registers
      REAL α,δ,λ,μ,μμ,ρ,σ,τ                    ... to improve speed & accuracy.
      REAL                                     Zero, Half, One, Two, Four
      DATA Zero,Half,One,Two,Four / 0.0, 0.5, 1.0, 2.0, 4.0 /
      ft = f
      fa = ABS(ft)
      ht = h
      ha = ABS(ht)
      L = ( ha .GT. fa )
      IF (L)  THEN
                  SWAP( ft, ht )
                  SWAP( fa, ha )
          ENDIF                                ... now fa ≥ ha .
c
      gt = g
      ga = ABS(gt)
      IF ( ga .EQ. Zero ) THEN
          v = ha                               ... the trivial case.
          w = fa
          cLt = One
          cRt = One
          sLt = Zero
```

```
            sRt = Zero
c           - - - - - - - - - - - - - - - -
      ELSE IF ( ga+fa .EQ. ga ) THEN
            w = ga                                  ... the case of gargantuan g .
            IF ( ha .GT. One ) THEN
                                          v = fa/(ga/ha)
                                    ELSE
                                          v = (fa/ga)*ha
                                    ENDIF
            cLt = One
            sLt = ht/gt
            cRt = ft/gt
            sRt = One
c           - - - - - - - - - - - - - - - -
      ELSE
            δ = fa - ha                             ... the normal cases.
            IF (δ .EQ. fa) THEN
                                          λ = One   ... copes with infinite f or h .
                                    ELSE
                                          λ = δ/fa
                                    ENDIF             ... 0 ≤ λ ≤ 1.
            μ = gt/ft                                 ... |μ| ≤ 1/ε.
            τ = Two - λ                               ... τ ≥ 1.
            μμ = μ*μ
            σ = SQRT(τ*τ + μμ)                        ... 1 ≤ σ < 1+1/ε.
            IF (λ .EQ. Zero) THEN
                                          ρ = ABS(μ)
                                    ELSE
                                          ρ = SQRT(λ*λ + μμ)
                                    ENDIF
            α = Half*(σ + ρ)                          ... 1 ≤ α < 1+|μ|.
            v = ha/α
            w = fa*α
            IF (μμ .EQ. Zero) THEN                    ... μ must be very tiny.
                  IF (λ .EQ. Zero) THEN               ... with IEEE 754/854
                                          τ = CopySign(Two,μ)
c                                   i. e., τ = SIGN(Two,ft)*SIGN(One,gt)
                                    ELSE
                                          τ = gt/SIGN(δ,f) + μ/τ
                                    ENDIF
                  ELSE
                                          τ = (μ/(σ+τ) + μ/(ρ+λ))*(One+α)
                                                      ... see previous footnote
                  ENDIF
            λ = SQRT(τ*τ + Four)                      ... about λ = 1 - |h/f|
            cRt = Two/λ
            sRt = τ/λ
            cLt = (cRt + sRt*μ)/α
            sLt = (ht/ft)*sRt/α
```

```
      ENDIF
c               - - - - - - - - - - - - - - - - - -
      IF (L) THEN
                     SWAP( cLt, sRt )
                     SWAP( sLt, cRt )
             ENDIF
      cL = cLt
      sL = sLt
      cR = cRt
      sR = sRt
      RETURN
c     Cost:  15 Add/Subtract/Compares, 9 Multiplies, 10 Divides, 3 SQRTs
      END.
c     = = = End of SVD2x2 = = = W. Kahan April 27, 1988
```

$b^T = (0, \cdots, 0, c/T_0^N)$, and $x^T = (x_N, \cdots, x_0)$.

We chose $M = 15$, $N = 14$, $T = 512.$, $T_0 = 500.$, and $c = 100.$ for this example. We used a single precision implementation of the IEEE Floating Point Standard [8] on a VAX 11/780[1] for which ε was $2^{-23} \approx 1.19_{10}-7$ and λ was $2^{-126} \approx 1.18_{10}-38$. There was a switch on the compiler to enable/disable G.U., so we were able to obtain numerical results using both G.U. and S.Z.

L and U have a simple structure. L will be zero below the diagonal, except for the last row, which is graded from $L_{15,1} \approx 7.14285_{10}-2$ down to $L_{15,14} \approx 5.34726_{10}-35$. U is identical to A in all but its last row.

$$L = \begin{bmatrix} 1 & & & & \\ 0 & 1 & & & \\ 0 & 0 & & & \\ \vdots & \vdots & & & \\ 0 & 0 & \cdots & 1 & \\ L_{15,1} & L_{15,2} & \cdots & L_{15,14} & 1 \end{bmatrix} \tag{44}$$

$$U = \begin{bmatrix} N & -1/T & -1/T^2 & \cdots & \cdot & -1/T^N \\ & N-1 & -1/T & \cdots & \cdot & -1/T^{N-1} \\ & & N-2 & \cdots & \cdot & -1/T^{N-2} \\ & & & \ddots & & \vdots \\ & & & & 1 & -1/T \\ & & & & & U_{15,15} \end{bmatrix} \tag{45}$$

A's columns are badly scaled, although this is not obvious because no row nor column is drastically smaller in norm than any other; nonetheless, bad scaling causes A to *appear* very ill conditioned, and this ill conditioning shows up in the last row of U, making $U_{15,15}$ very small, barely above the underflow threshold. S.Z. and G.U. compute all elements of L and U identically except for $U_{15,15}$. In fact, all additions in the computation of L add normalized numbers with like magnitudes and like signs, so no cancellation, loss of significance, nor underflows occur. If the exponent range were unbounded, so underflow never happened, the correct value $U_{15,15} \approx 2.09261_{10}-37$ would be computed. This is the value computed using G.U. But when S.Z. is used instead, the computed value is $U_{15,15}^{S.Z.} \approx 1.72763_{10}-37$, a relative difference of .174 from the correct value. All additions in the computation of $U_{15,15}$ involve numbers of like magnitude and sign, so cancellation cannot be blamed for the discrepancy. This relative difference in the last entry of U is very important, because one divides by $U_{15,15}$ in the course of solution. Thus, the computed solution $x^{G.U.}$ is very close to the true x, and the relative difference in solution vectors is

$$\frac{\|x^{G.U.} - x^{S.Z.}\|_\infty}{\|x^{G.U.}\|_\infty} \approx .211.$$

Thus, G.U. obtains markedly better results than S.Z. This example is very interesting because there is nothing obviously wrong with the matrix. All its entries are unexceptional normalized numbers, and every row and every column contains at least one number no tinier than $1/T \approx .00195$ and none larger than $N = 14$, yet 11 out of 14 products $L_{15,j} * U_{j,15}$ in the sum for $U_{15,15}$ underflow just slightly below the underflow

[1] VAX is trademark of the Digital Equipment Corporation.

threshold. Since the true value of $U_{15,15}$ is itself not much larger than the underflow threshold, this makes for a large relative error.

This example was chosen to be simple and realistic; even though it can be solved analytically, it could be changed easily into a two-dimensional problem without an explicit solution, but with the same sensitivity to underflow.

We repeat that even though A appears very ill conditioned, since $k(A) \approx 1/\lambda$ (i.e. near the overflow threshold in most arithmetics), it is also well conditioned in the sense that Cond $(A, x) \approx 5.5$. We will discuss the significance of this example later in § 8.4.

8.2.4. *Example* 4. Let

$$A = \begin{bmatrix} G & G \\ g & 2g \end{bmatrix}, \qquad A^{-1} = \begin{bmatrix} 2/G & -1/g \\ -1/G & 1/g \end{bmatrix},$$

where g/G underflows to 0 using either S.Z. or G.U. The L obtained is thus the identity matrix since $L_{2,1} = \mathrm{fl}(g/G) = 0$, and so the L and U obtained are the exact factors of the matrix

$$A + E = \begin{bmatrix} G & G \\ 0 & 2g \end{bmatrix},$$

which is a very different matrix than A. If $b^T = (G, 0)$, then $x = A^{-1}b = (2, -1)^T$, whereas $\hat{x} = (A+E)^{-1}b = (1, 0)^T$, so $\hat{x}$ does not resemble x at all. The residual r is however guaranteed to be small, in the sense that $\|r\|_\infty / \| |A||\hat{x}| + |b| \|_\infty$ is small:

$$\frac{\|r\|_\infty}{\| |A||\hat{x}| + |b| \|_\infty} = \frac{\|E\hat{x}\|_\infty}{\| |A||\hat{x}| + |b| \|_\infty}$$

$$\leqq \frac{g|\hat{x}_1|}{G|\hat{x}_1| + G|\hat{x}_2|} \leqq \frac{g}{G} \leqq \lambda\varepsilon/2.$$

Of course A is an exceedingly ill conditioned matrix in the sense that $k(A) \approx 2G/g$ is beyond the reciprocal of the underflow threshold, so we would be inclined not to trust our results anyway. However, Cond (A) is only 7. This is true because Cond (A) = Cond (DA) for any nonsingular diagonal matrix D, so A has the same condition number as the utterly tame matrix

$$\begin{bmatrix} G^{-1} & \\ & g^{-1} \end{bmatrix} A = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}.$$

Needless to say, in the absence of underflow we would compute a very accurate solution. We will return to this example later to explain why we can get inaccurate results from a matrix with a small condition number Cond (A).

8.3. Results of error analysis.

8.3.1. *Approach.* As stated in the introduction, we use backward error analysis. Thus, when Gaussian elimination is used to solve

$$Ax = b \tag{46}$$

for x it generates instead an approximation $\hat{x} = x + \delta x$ which satisfies some perturbed problem

$$(A + \delta A)\hat{x} = b + \delta b. \tag{47}$$

The task of backwards error analysis is to infer bounds on δA and δb from the details of the arithmetic used to implement the elimination process. These bounds can be

used in turn to bound the residual

$$r = A\hat{x} - b = -\delta A\hat{x} + \delta b = A\delta x \tag{48}$$

and then the error δx.

Wilkinson's approach [22] is to determine a bound ω_w on the errors

$$\|\delta A\|_\infty \leqq \omega_w \|A\|_\infty \quad \text{and} \quad \|\delta b\|_\infty \leqq \omega_w \|b\|_\infty \tag{49}$$

whence

$$\|r\|_\infty \leqq \omega_w [\|A\|_\infty \|\hat{x}\|_\infty + \|b\|_\infty] \tag{50}$$

and then it will follow that the error δx is bounded:

$$\frac{\|\delta x\|_\infty}{\|x\|_\infty + \|\hat{x}\|_\infty} \leqq \omega_w k(A). \tag{51}$$

The detailed derivation of ω_w from the details of the arithmetic is given elsewhere [2]. Theorem 1 below states simple requirements on A and b that ensure ω_w will be scarcely worse if underflow occurs than if it does not.

Skeel's approach [20], modified slightly here, is to determine a bound ω_s on the relative error in each entry of A and b:

$$|\delta A| \leqq \omega_s |A| \quad \text{and} \quad |\delta b| \leqq \omega_s |b|. \tag{52}$$

From these inequalities follows a bound upon the error δx:

$$\frac{\|\delta x\|_\infty}{\|x\|_\infty} \leqq \omega_s \cdot \frac{\| |A^{-1}||A||x| + |A^{-1}||b| \|_\infty}{(1 - \omega_s \| |A^{-1}||A| \|_\infty)\|x\|_\infty} \tag{53}$$

(provided the denominator is positive). This motivates defining the following condition numbers:

$$\text{Cond}\,(A, x) \equiv \frac{\| |A^{-1}||A||x| \|_\infty}{\|x\|_\infty}, \tag{54a}$$

$$\text{Cond}\,(A) \equiv \| |A^{-1}||A| \|_\infty. \tag{54b}$$

Cond (A) is an upper bound for Cond (A, x) for all x; the error bounds are useful only if ω_s Cond $(A) < 1$.

Following Oettli and Prager [19] and Skeel [20] we use an expression for ω_s obtainable from (48) in terms of the residual r:

$$\omega_s = \max_i \frac{|r_i|}{(|A||\hat{x}| + |b|)_i} \tag{55}$$

where the max is over those i for which the denominator is nonzero. Following Skeel, we overestimate ω_s by analyzing the elimination process to infer an inequality

$$\|r\|_\infty \leqq \omega_s' \| |A||\hat{x}| + |b| \|_\infty \tag{56}$$

from which we compute the overestimate $\bar{\omega}_s$ as

$$\bar{\omega}_s = \frac{\max_i (|A||\hat{x}| + |b|)_i}{\min_i (|A||\hat{x}| + |b|)_i} \omega_s' \tag{57}$$

(where the min in the denominator is over the nonzero values of $(|A||\hat{x}|)_i$ only). Unfortunately $\bar{\omega}_s$ can be a gross overestimate of ω_s, as we will see when we return to Example 3 later.

The detailed derivation of ω_s' is given in [2]. Theorem 2 below states requirements on A and b that ensure ω_s' will be scarcely worse if underflow occurs than if it does not. These requirements on A and b are nearly identical to the requirements in the Wilkinson style analysis.

8.3.2. *Results.*

THEOREM 1. *Wilkinson style error analysis of solving $Ax = b$ with Gaussian elimination in the presence of underflow: Let $a_{\max} = \max_{ij} |A_{ij}|$, and $g = [$largest intermediate result appearing in the decomposition$]/a_{\max}$. g is the "pivot growth factor" and is $\leqq 2^{n-1}$.*

Then a bound ω_w for which

$$\|r\|_\infty \leqq \omega_w[\|A\|_\infty \|\hat{x}\|_\infty + \|b\|_\infty] \tag{50}$$

is given as follows. In the absence of underflow, we have

$$\omega_w = n^3 \varepsilon g/2. \tag{58}$$

If underflow occurs then

$$\omega_w = 3n^3 \varepsilon g/2 \tag{59}$$

provided certain conditions are met. For G.U. these conditions are:

$$\begin{aligned} & g a_{\max} \geqq \lambda && \text{if there are any underflows during triangular decomposition,} \\ & \|b\|_\infty \geqq \frac{\lambda}{n} && \text{if there are any intermediate underflows during forward and back substitutions,} \\ & \frac{\|b\|_\infty}{a_{\max}} \geqq \frac{2\lambda}{n^2} && \text{if the solution } \hat{x} \text{ itself underflows in some component.} \end{aligned} \tag{60}$$

For S.Z. the above conditions still apply but λ must be increased to λ/ε.

Proof. See [2].

THEOREM 2. *Skeel style error analysis of solving $Ax = b$ with Gaussian elimination in the presence of underflow: Let $a_j = \max_i |A_{ij}|$, and $g_C = \max_j ([$largest intermediate result appearing in the decomposition in column $j]/a_j)$. g_C is the "columnwise pivot growth factor" and is $\leqq 2^{n-1}$.*

Then a bound ω_s' for which

$$\|r\|_\infty \leqq \omega_s' \| |A||\hat{x}| + |b| \|_\infty \tag{56}$$

is given as follows. In the absence of underflow we have

$$\omega_s' = n^3 \varepsilon g_C. \tag{61}$$

If underflow occurs, then

$$\omega_s' = 3n^3 \varepsilon g_C/2 \tag{62}$$

provided certain conditions are met. For G.U. these conditions are:

$$\begin{aligned} & g_C a_j \geqq \lambda && \text{for all } j, \text{ if there are any underflows during triangular decomposition,} \\ & \|b\|_\infty \geqq \frac{\lambda}{2n} && \text{if there are any intermediate underflows during forward and back substitutions,} \\ & \frac{\|b\|_\infty}{a_{\max}} \geqq \frac{\lambda}{n^2} && \text{if the solution } \hat{x} \text{ itself underflows in some component.} \end{aligned} \tag{63}$$

For S.Z. the above conditions apply with except λ must be increased to λ/ε.

Proof. See [2].

The theorems indicate how to write software that will solve $Ax = b$ reliably despite underflow, and how the requirements for G.U. differ from those for S.Z. To keep the residual small in the sense of a Wilkinson style error analysis, we appeal to Theorem 1. With G.U., as long as one normalized number appears during the decomposition ($ga_{\max} \geq \lambda$), residual with underflow has a bound not much worse than residual without underflow. If there are intermediate underflows while solving the triangular systems, as long as some component of b is normalized ($\|b\|_\infty \geq \lambda$), residual with underflow has a bound scarcely worse than without underflow. If the answer $\hat{x}$ itself underflows, we can either issue an error message (which would be very reasonable since the first goal of reliable software is only to compute an answer if it is representable) or test to see if $\|b\|_\infty / a_{\max}$ is not too small.

All these requirements are natural ones to make, since they say that when a problem's inputs and its computed solution are normalized numbers, we should expect the residual to be scarcely worse with underflow than without. Thus, the only gradual underflows which can cause concern in a problem with normalized inputs are underflows in the solution itself. The scaling condition $\|b\|_\infty / a_{\max} \geq \lambda / n^2$ arises naturally; consider solving the scalar equation $ax = b$ by the division $x = b/a$.

In contrast, the bounds for S.Z. are all higher by a factor of $1/\varepsilon$. Thus, using S.Z. we can neither solve as many problems as the G.U., nor decide so easily which underflows matter. Thus, from the point of view of a Wilkinson style error analysis, G.U. makes writing reliable software easier.

Theorem 2 shows that Skeel style bounds for the residual are scarcely worse with underflow than without provided conditions are satisfied that are almost the same as in Theorem 1. Therefore the previous paragraphs' comments remain valid provided, when underflow is gradual, at least one normalized number appears in each column of A, rather than just somewhere is A, before or during the decomposition process.

8.4. Examples 3 and 4 revisited. We wish to emphasize that we have only derived conditions under which with underflow are about the same as without underflow. There is no way using this analysis to say how closely this bound will be approached with and without underflow, or how accurate the computed solution will be.

In Example 4 above, the matrix A and vector b satisfy all the conditions of Theorems 1 and 2 for G.U. as well as S.Z., so the residual is small, but the answer $\hat{x}$ is totally inaccurate. This inaccuracy can be explained either by the huge condition number $k(A) \approx$ overflow threshold, or the large backwards error in equation (55): $\omega_s = 1$. In this case ω_s's upper bound $\bar{\omega}_s$ in (57) is also 1. Thus, having a small value of Cond (A) is not sufficient to guarantee accuracy given a small residual ω_s' ((56)), although a small value of $k(A)$ combined with a small residual ω_w is enough, as can be seen from (51).

Example 3 is another case where the conditions of Theorems 1 and 2 hold, but now G.U. successfully computes the last pivot $U_{15,15}$ and an accurate solution $\hat{x}$ while S.Z. does not. Again, we have a problem where $k(A)$ is huge and Cond (A, x) is small. Now the ω_s of equation (55) is $\approx 5.23_{10}-8$, verifying the high accuracy of solution. Unfortunately the bad scaling of the matrix causes the upper bound $\bar{\omega}_s$ of equation (57) to be $2.0_{10}20$. This example demonstrates the occasionally intense pessimism of Skeel's approach.

In summary, the significance of Examples 3 and 4 is to show that maintaining a small residual in the face of underflow does not guarantee an accurate solution $\hat{x}$.

although we conjecture that for not terribly ill conditioned matrices G.U. will provide answers at least as accurate as provided by S.Z.

We have proven something quite unremarkable: if underflows are gradual, then we continue to get what we have come to expect from Gaussian elimination. That is, we get a small residual as long as the inputs and outputs are all representable (normalized) numbers and there is no indication of singularity or excessive pivot growth. If, however, underflows are handled in the usual way and set to zero, then no such simple guarantee can be made, and some kind of testing on the scaling of the problem is necessary. These results demonstrate that gradual underflow makes it easier to write reliable linear equation solvers than "store zero."

9. Cholesky decomposition.

9.1. Summary. The algorithm we discuss is analogous to Gaussian elimination, but is applicable only to positive definite symmetric matrices A:

(1) Decompose $A = LL^T$ where L is lower triangular;

(2) Solve $Ly = b$ for y (forward substitution);

(3) Solve $L^T x = y$ for x (backward substitution).

We expect the same kind of reliability from this algorithm in the absence of underflow as we do from Gaussian elimination: a small residual $A\hat{x} - b$ where $\hat{x}$ is the computed solution, and that $\hat{x}$ is the exact solution of a slightly different problem than the original.

With G.U., as long as *one* component each of the matrix A and right-hand side b are normalized the only harmful underflows are underflows in components of x and y (recall that with Gaussian elimination the only harmful underflows were in the solution x). Intermediate gradual underflows contribute an error with a bound scarcely worse than the bound for the error contributed by roundoff alone. As with Gaussian elimination, the accuracy test for underflow (see § 3) leads to fewer false alarms than the threshold test, although the threshold test might make it easier to test the inputs to the Cholesky routine ("are the largest components of A and b at least λ in magnitude?") for the applicability of this analysis.

In contrast, with S.Z. intermediate underflows during any stage of solution can introduce significant errors, possibly producing reasonable looking results whose error greatly exceeds the uncertainty attributable to roundoff alone (see the examples). In fact, one can show that S.Z. can only produce a decomposition of a matrix when G.U. fails if the matrix is so ill conditioned that the computed solution cannot be trusted, or if it is not positive definite at all (see § 9.2.2).

As with Gaussian elimination, the results of this section remain true even if intermediate products are computed to extra range and precision, as long as the entries of L, y and $\hat{x}$ are stored in the range and precision of A and b.

Section 9.2 contains examples and § 9.3 contains theorems and conclusions. Proofs of these results can be found in [2].

9.2. Examples.

9.2.1. *Example* 1. Let m be the smallest floating point number $\geq \sqrt{\lambda}$, so that m^2 does not underflow. Consider the family of symmetric matrices:

$$A(x) = m^2 \cdot \begin{bmatrix} 4 & 2 & 1 \\ 2 & 2 & 1 \\ 1 & 1 & x \end{bmatrix}$$

which has the exact lower triangular factor

$$L(x) = m \cdot \begin{bmatrix} 2 & & \\ 1 & 1 & \\ .5 & .5 & \sqrt{x-.5} \end{bmatrix}.$$

$L^{G.U.}(x)$, the factor provided by Cholesky using G.U., is the same as $L(x)$ except for the rounding error incurred by having to represent $(x-.5)^{1/2}$. $L^{S.Z.}(x)$, the factor provided by S.Z., is

$$L^{S.Z.}(x) = m \cdot \begin{bmatrix} 2 & & \\ 1 & 1 & \\ .5 & 1 & L_{33}^{S.Z.}(x) \end{bmatrix}$$

where

$$L_{33}^{S.Z.}(x) = \begin{cases} \sqrt{x-1} & \text{if } x \geqq 2, \\ 0 & \text{if } 2 > x \geqq 1 \end{cases}$$

so S.Z. computes a totally wrong value for $L_{33}(x)$, incorrectly labelling the matrix singular for $2 > x \geqq 1$ when in fact it is well conditioned.

9.2.2. *Example* 2. Let m be as before. Consider the family of matrices

$$A(x) = m^2 \cdot \begin{bmatrix} 4 & & & & 1 \\ & 4 & & & 1 \\ & & 4 & & 1 \\ & & & 4 & 1 \\ 1 & 1 & 1 & 1 & x \end{bmatrix}.$$

Its correct factor $L(x)$, if it exists, is

$$L(x) = m \cdot \begin{bmatrix} 2 & & & & \\ & 2 & & & \\ & & 2 & & \\ & & & 2 & \\ .5 & .5 & .5 & .5 & \sqrt{x-2} \end{bmatrix}.$$

This matrix is positive definite if $x > 2$, positive semidefinite if $x = 2$, and has both positive and negative eigenvalues if $x < 2$. Both G.U. and S.Z. compute all entries of the factor $L(x)$ except the (5,5) entry correctly (using Cholesky decomposition). G.U. obtains the correct value $(x-2)m^2$ for its value of L_{55}^2, whereas S.Z. computes xm^2. Thus, as x decreases from 3 to 2 to 1, G.U. correctly decides the matrix is positive definite when $x = 3$, and becomes nonpositive definite when $x \leqq 2$. S.Z., on the other hand, produces an (incorrect) decomposition all the way down to $x = 1$. Thus, S.Z. cannot only produce an inaccurate decomposition, but produces it after G.U. has correctly decided no such decomposition exists.

S.Z. can produce a decomposition of a matrix when G.U. fails only if the matrix is either 1) so ill conditioned that the decomposition cannot be trusted, or 2) not positive definite at all. Here is the reason. Assume $a_{max} \geqq \lambda$, since otherwise the matrix is identically 0 in S.Z. arithmetic. G.U. fails when its computed value of L_{jj}^2 either rounds to 0 or is negative for some j. L_{jj}^2 rounds to 0 when $L_{jj}^2 < \lambda\varepsilon$. It is easy to see

that $a_{max} \leqq \lambda_{max}(A)$ and $L_{jj}^2 \geqq \lambda_{min}(A)$, because

$$\frac{1}{\min_j L_{jj}^2} = (\lambda_{max}(L^{-1}))^2 \leqq |L^{-1}|_2^2 = |A^{-1}|_2 = \frac{1}{\lambda_{min}(A)}.$$

Therefore

$$k_2(A) = \frac{\lambda_{max}}{\lambda_{min}} > \frac{a_{max}}{L_{jj}^2} > \frac{1}{\varepsilon},$$

which means that the matrix is so ill conditioned as to make it difficult to even recognize an accurate inverse, let alone compute one. If L_{jj}^2 is in fact negative, the matrix is not positive definite.

9.3. Results of error analysis.

9.3.1. *Approach.* Our approach is essentially identical to the one we used to analyze Gaussian elimination with the following additions. The Cholesky decomposition uses the square root operation which Gaussian elimination does not. We model the error in square root as follows:

$$\text{SQRT}(x) = \sqrt{x} \cdot (1+e) \text{ for all } x \tag{64}$$

where $|e| < \varepsilon$. (SQRT denotes the floating point square root.) (64) holds because SQRT compresses the exponent range, making overflow and underflow impossible. We make an extra assumption about λ and ε we did not need before; it also arises from the use of square roots in the Cholesky decomposition. This relationship is satisfied by all single precision arithmetics known to the author (but not by a number of double precision arithmetics, such as D format on the VAX, for example) and is only needed to analyze Cholesky decomposition using S.Z.: $\lambda < \varepsilon^3$.

9.4. Results.

THEOREM 3. *Wilkinson style error analysis of solving $Ax = b$ with Cholesky Decomposition in the presence of underflow. Let $a_{max} = \max_{ij} |A_{ij}|$. Then a bound ω_w for which*

$$\|r\|_\infty \leqq \omega_w [\|A\|_\infty \|\hat{x}\|_\infty + \|b\|_\infty] \tag{50}$$

is given as follows. In the absence of underflow, we have

$$\omega_w = n^3 \varepsilon / 2. \tag{65}$$

If underflow occurs then

$$\omega_w = 4n^3 \varepsilon / 2 \tag{66}$$

provided certain conditions are met. For G.U these conditions are:

$$\begin{aligned}
&a_{max} \geqq \lambda && \text{if there are any underflow during Cholesky decomposition,}\\
&\|b\|_\infty \geqq \frac{2\lambda}{n^2} && \text{if there are any intermediate underflow during forward substitution,}\\
&\frac{\|b\|_\infty}{\sqrt{a_{max}}} \geqq \frac{\lambda}{n} && \text{if some } y_i \text{ underflow or there are any intermediate underflows during back substitution,}\\
&\frac{\|b\|_\infty}{a_{max}} \geqq \frac{2\lambda}{n^2} && \text{if the solution } \hat{x} \text{ itself underflows in some component.}
\end{aligned} \tag{67}$$

For S.Z. the above conditions still apply but λ must be increased to λ/ε.

Proof. See [10].

The above theorem shows how to write software that will solve $Ax = b$ with Cholesky reliably despite underflow just as Theorems 1 and 2 in § 8.3.2 did for Gaussian elimination. With G.U., as long as there is one normalized component in A ($a_{\max} > \lambda$) residual with underflow has a bound scarcely worse than without underflow. If there are intermediate underflows during forward substitution, the residual bound is again scarcely worse than without underflow as long as some component of b is normalized ($\|b\|_\infty \geqq \lambda$). Intermediate underflows during back substitution or in y require a scaling condition ($\|b\|_\infty / a_{\max} \geqq \lambda/n$) to be satisfied, as do underflows in the final solution ($\|b\|_\infty / a_{\max} \geqq 2\lambda/n^2$). It is clear that some such scaling condition needs to be satisfied from considering the $n = 1$ case (i.e. solving the scalar equation $ax = b$ by two divisions $x = (b/\sqrt{a})/\sqrt{a}$). If there are underflows in the back substitution, y, or x, then we can either issue an error message or check the scaling.

For S.Z. all the bounds are higher than the ones for G.U. by a factor of $1/\varepsilon$.

The situation with Cholesky is not as satisfactory as for Gaussian elimination, where only underflows in the final solution x could matter for G.U.

10. Iterative refinement. We study the following algorithm for refining the solution of the linear system $Ax = b$. The phrase "in precision (ε, λ)" means that particular computation is to be done in arithmetic with rounding error ε and underflow threshold λ. x_0 is an arbitrary starting vector.

```
i := 0
repeat
  r_i := Ax_i − b in precision (ε_r, λ_r)
  solve Ad_i = r_i for d_i in precision (ε, λ)
  x_{i+1} := x_i − d_i in precision (ε, λ)
  i := i + 1
until convergence.
```

Double precision computation of the residual (the traditional algorithm) corresponds to $\varepsilon_r = \varepsilon^2$, and single precision to $\varepsilon_r = \varepsilon$. We also assume $\lambda_r \leqq \lambda$.

In order to understand the effects of underflow on this algorithm, we need a theorem due to Skeel [21] which shows, contrary to popular belief, that computing r_i in single precision ($\varepsilon_r = \varepsilon$) does improve the solution in a significant way.

THEOREM 4. *Analysis of iterative refinement in the absence of underflow for both single and double precision computation of the residual: As long as the condition number* $\mathrm{Cond}(A) = \||A^{-1}||A|\|_\infty$ *is sufficiently less then* $1/\varepsilon$, *then*

1) *If* $\varepsilon_r = \varepsilon^2$ (*double precision residual computation*) *then*

$$\limsup_{i\to\infty} \|x - x_i\|_\infty \leqq 2\varepsilon \|x\|_\infty \tag{68}$$

where x denotes the exact solution;

2) *If* $\varepsilon_r = \varepsilon$ (*single precision residual computation*) *then*

$$\limsup_{i\to\infty} |Ax_i - b| \leqq 4n\varepsilon |A||x_i|. \tag{69}$$

Furthermore, this inequality is almost always attained after just one application of iterative refinement.

Proof. See [21].

This last inequality means that for large enough i, x_i is the solution of a slightly perturbed problem

$$(A+\delta A)x_i = b$$

where $|\delta A_{ij}| < 4n\varepsilon|A_{ij}|$. In other words, the perturbed problem agrees with the original problem up to a few rounding errors *in each component* [19]. This is a very strong notion of backwards error, and so Skeel's theorem shows that single precision iterative refinement does lead to a significantly more reliable code than no refinement at all.

How does underflow effect this reliability? For G.U., we can say the following:

If the inputs A and b and the output x are normalized and if either double or single precision residuals are computed, then gradual underflows can degrade the algorithm's performance to the level of single precision residual computation but no worse. To guarantee double precision performance, both b and x need to exceed λ/ε. Specifically, it is underflow in $r_i = Ax_i - b$ that contributes to the lower bound on b and underflow in d_i that contributes to the lower bound in x. Using this information, the accuracy test for G.U. could be used to decide when underflow might degrade the performance more precisely than the threshold test. For S.Z., all thresholds are increased by $1/\varepsilon$.

The use of extended range and precision in intermediate computations does not change these conclusions. Assuming r_i and d_i are stored in the same format as A, b and x, underflows in r_i and d_i have the same potential effects on performance as they did when they were not computed in extended range.

We have not yet considered underflow's effect on the *rate* of convergence of the iteration. There are matrices for which the iteration converges only if underflows do not occur, but the matrices are so ill conditioned as to make the computed solution untrustworthy anyway. It follows from the analysis of § 8 that as long as some entry of A is large enough (λ for G.U. and λ/ε for S.Z.) then underflows will have an effect on the convergence rate comparable to round-off.

11. Polynomial evaluation and root finding.

11.1. Horner's rule for polynomial evaluation. We consider Horner's rule for evaluating the polynomial $\sum_{i=0}^{n} a_i x^i$ for real a_i and x:

(70)
```
sum := a_n
for i := n-1 to 0 do sum := sum*x + a_i
```

We have the following very satisfying theorem.

THEOREM 5 (Analysis of Horner's rule for polynomial evaluation). *Let P denote the result of applying Horner's rule to the polynomial $\sum a_i x^i$ above. Then in the absence of underflow and overflow we have*

$$P = \sum_{i=0}^{n} a_i(1+E_i)x^i \tag{71}$$

where

$$|E_n| \leqq 2n\varepsilon \quad \text{and} \quad |E_i| \leqq (2i+1)\varepsilon \quad \text{if } i < n. \tag{72}$$

In the presence of underflow we write

$$P=\sum_{i=0}^{n}(a_i+\eta_i)(1+E_i)x^i \tag{73}$$

where E_i *has the same bound as in* (72), $\eta_n=0$ *for both* G.U. *and* S.Z., *and*

$$|\eta_i|\leqq\lambda\varepsilon \text{ for G.U. and } |\eta_i|\leqq 2\lambda \text{ for S.Z.} \tag{74}$$

for $i<n$.

The proof is a straightforward extension of the usual error analysis of Horner's rule [23] using formula (2) of § 3.

Thus, in the absence of underflow and overflow, Horner's rule delivers the exact value of a new polynomial each coefficient a_i of which differs by a few rounding errors from the corresponding original a_i. This is a strong backwards error bound.

For G.U., we can make the same kind of statement providing we define backwards error as motivated by the last paragraph of § 3: a relative error no greater than ε for values $>\lambda$ and an absolute error no greater than $\lambda\varepsilon$ for smaller values. Thus, for example, we treat the value 0 as indistinguishable from any value in the interval $[-\lambda\varepsilon/2, \lambda\varepsilon/2]$. By this definition of backwards error, Horner's rule with G.U. delivers the exact value of a new polynomial each of whose coefficients differs by a small relative/absolute error from the corresponding original coefficient. We can further guarantee each new coefficient has a small *relative* error with respect to the original if each a_i is a *nonzero* normalized number.

For S.Z. all thresholds in the last paragraph increase by $1/\varepsilon$ to be able to make corresponding statements.

Here, extended range and precision is extremely beneficial, eliminating most concerns about over/underflow. Indeed, any overflows in extended range would have occurred with the original range, and any underflows in extended range would contribute an uncertainty far less than a unit in the last place of even the smallest denormalized number to any a_i.

11.2. Polynomial root finding. Linnainmaa [18] has analyzed Newton's method for root finding and shown that it is much easier to write an underflow/overflow proof code if G.U. is available than if it is not. An essential feature of his code is evaluating $\sum a_{n-i}z^i$ at $z=1/x$ instead of $\sum a_i x^i$ when $x>1$. This changes almost all potential overflow problems to underflow problems, which are handled by G.U. The advantage of evaluating polynomials at points $x<1$ is that any rounding or underflow errors made early in Horner's recurrence are multiplied down by factors of x. In particular, underflow errors, already at the level of roundoff in the smallest normalized number, only decrease in significance so that if the final value P is normalized we know that any gradual underflows must be completely harmless.

12. Computing eigenvalues of symmetric tridiagonal matrices. Given the symmetric tridiagonal matrix:

$$T=\begin{bmatrix} a_1 & b_2 & & & \\ b_2 & a_2 & b_3 & & \\ & & & & \\ & & & b_n & a_n \end{bmatrix}, \tag{75}$$

how do we compute its eigenvalues? One way is to use the following program which, given a real value z, computes (in exact arithmetic) $v(z)=$ the number of eigenvalues

of T that are $< z$:

$$
\begin{aligned}
&u := 1 \\
&v := 0 \\
&\text{for } j := 1 \text{ to } n \text{ do} \\
&\qquad u := a_j - z - (b_j/u)b_j \\
&\qquad \text{if } u < 0 \text{ then } v := v + 1,
\end{aligned} \tag{76}
$$

where we define $b_1 \equiv 0$. We assume $b_i \neq 0$ for $i > 1$, since otherwise T is block diagonal and its eigenvalues are those of its diagonal blocks. We also use the conventions $\pm 1/0 = \pm\infty$ and $1/\pm\infty = 0$ (which are part of the proposed IEEE floating point standard). A proof that this algorithm computes what we claim is based on Sylvester's inertia theorem and can be found in [5]. It can be used to obtain eigenvalues to any desired accuracy by bisecting an interval in which $v(z)$ increases (which means the interval contains an eigenvalue) until the interval is narrow enough.

What does this algorithm compute when implemented in floating point? There are two interesting questions:

Is $v(z)$ a monotone increasing function of z as it is in exact arithmetic?

Do we compute accurate eigenvalues either of our original matrix or a matrix very close to our original matrix?

In the absence of overflow and underflow, the answer to both questions is yes [11]:

The function $v(z)$ computed by algorithm (76) in the absence of overflow and underflow is an increasing function of z. Furthermore, the value of $v(z)$ computed is the exact value of $v(z)$ for a matrix T' whose diagonal entries a_i' are identical to the diagonal entries a_i of T, and whose off diagonal entries b_i' satisfy $b_i' = b_i(1 + e_i)$ where $|e_i| \leqq 2\varepsilon$. T' will in general depend on z.

This is a very strong backwards error bound. It says we can compute the exact number of eigenvalues less than z of a matrix differing from the original by a small relative error in the off diagonal entries, and with no difference on the diagonal.

What can be said in the presence of underflow? Barring overflow, $v(z)$ remains monotonic using either S.Z. or G.U. The only property of the arithmetic needed to prove $v(z)$ monotonic is monotonicity of the arithmetic: if $a \geqq b$ are the exact results of two different arithmetic operations, then $\mathrm{fl}(a)$ must be $\geqq \mathrm{fl}(b)$ as well.

The monotonicity of $v(z)$ is an appealing property but not necessary for the correct functioning of a bisection algorithm for determining one eigenvalue [22]. Lack of monotonicity could lead to lower bounds exceeding upper bounds in codes for determining such bounds for all eigenvalues at once, but since $v(z)$ is monotonic, we will not discuss this possibility further.

Kahan [11] discusses an ironclad version of (76) which scales the problem and inserts tests against carefully chosen thresholds into the inner loop to guarantee that overflow and underflow (G.U. or S.Z.) cannot degrade the results appreciable more than roundoff. Here, we discuss the robustness of the unadorned code in (76) which differs from the most obvious algorithm only in using $(b_i/u)b_i$ in the inner loop instead of b_i^2/u. At the end we will say why this change is important. We assume we have a balanced exponent range, i.e. $\lambda\Lambda$ cannot be larger than a small integer m ($m = 4$ in the proposed IEEE standard). The backwards error in (76) is given as follows:

The function $v(z)$ computed by algorithm (76) is the exact value of $v(z)$ for a matrix T' whose entries a_i' and b_i' satisfy:

$$
\begin{aligned}
&a_i' = a_i + \eta_i \quad \text{where } |\eta_i| \leqq (1+m)\lambda\varepsilon \\
&b_i' = b_i(1 + e_i) \quad \text{where } |e_i| \leqq 2\varepsilon
\end{aligned} \tag{77}
$$

when using G.U., and

$$(78)\qquad \begin{aligned} a_i' &= a_i + \eta_i \quad \text{where } |\eta_i| \leqq (3+m)\lambda \\ b_i' &= b_i(1+e_i) \quad \text{where } |e_i| \leqq 2\varepsilon \end{aligned}$$

when using S.Z.

Thus, in order to claim that we are computing the exact $v(z)$ for a matrix T' which differs from T by at most a few rounding errors in each component, which is the case in the absence of underflow, we need to make the following constraints on a_i:

$$|a_i| \geqq \begin{cases} \lambda & \text{for G.U.,} \\ \lambda/\varepsilon & \text{for S.Z.} \end{cases}$$

If we adopt the relative/absolute error measure suggested in the last paragraph of § 3 and discussed further in § 11.1 in connection with polynomial evaluation, then there is no constraint at all on the a_i if we ue G.U. in order to claim that a_i' differs from a_i by a small error.

These backwards error bounds are so strong that it does not seem the accuracy test for G.U. could be of much more use than the threshold test, if indeed it is of any use at all.

A weaker form of backwards error often used in analyses of matrix computations [22] is

$$(79)\qquad \frac{\max_{i,j} |T_{ij}' - T_{ij}|}{\max_{i,j} |T_{ij}|}.$$

With respect to this definition, underflow is insignificant if

$$\max_{i,j} |T_{ij}| \geqq \begin{cases} \lambda & \text{for G.U.,} \\ \lambda/\varepsilon & \text{for S.Z.} \end{cases}$$

What would happen if we used b_i^2/u instead of $(b_i/u)b_i$ in the inner loop? In that case, any $|b_i|$ smaller than $\sqrt{\varepsilon\lambda} \gg \lambda$ would underflow to zero when squared whether we used G.U. or S.Z., and the resulting perturbation could not always be explained as a small change in either b_i or a_i. Thus, a seemingly small change in the code effects the robustness a great deal.

If extended range and precision are available, then almost all concerns with over/underflow vanish, as with Horner's rule for polynomial evaluation.

13. Numerical quadrature. Quadrature, along with the matrix algorithms discussed earlier, benefits from the ability to compute inner products more robustly with G.U. than S.Z. This is because most quadrature codes, when asked to compute

$$(80)\qquad \int_a^{a+h} w(x)f(x)\,dx$$

evaluate an inner product

$$(81)\qquad h \cdot \sum_{i=1}^{n} w_n f(x_n).$$

From the analysis of inner products in § 6, we see that as long as the inner product in (81) is a normalized number, the effects of gradual underflows are no worse than roundoff, but that some intermediate result in the inner product must exceed λ/ε to make the same claim about S.Z. All the benefits of extended range and precision to

inner products also accrue to numerical quadrature. A more detailed analysis can be found in [14].

14. Accelerating the convergence of sequences. Methods to accelerate convergence of sequences often do so by extrapolating an estimated error to zero. This requires taking the ratio of differences of successive elements in the sequence. If the sequence is converging to a value near the underflow threshold, these differences can underflow to zero using S.Z. but not G.U. We illustrate with Aitken's δ^2 method.

Given a sequence $\{x_n\}$ which converges to a finite nonzero x, Aitken's δ^2 method produces a new sequence $\{x'_n\}$

$$x'_n = x_n - \left(\frac{x_{n+1}-x_n}{(x_{n+2}-x_{n+1})-(x_{n+1}-x_n)}\right)(x_{n+1}-x_n) \tag{82}$$

which will converge to x faster than $\{x_n\}$ under certain conditions [9]. We have written the term following x in (82) (the correction term) as it appears instead of as in

$$x'_n = x_n - \frac{(x_{n+1}-x_n)^2}{x_{n+2}-2x_{n+1}+x_n} \tag{83}$$

because of the latter's much greater susceptibility to over/underflow. (83) is likely to cause over/underflow if $|x|$ is much outside the range $[\sqrt{\lambda}, \sqrt{\Lambda}]$. (82) is much more robust. In fact, if N is large enough so that

$$\frac{1}{\sqrt{2}} < \frac{|x_n|}{|x|} < \sqrt{2}$$

for $n > N$ and we use G.U., then the correction term in (82) will be computed to within 2 rounding errors in x if $\lambda \leqq |x| \leqq \Lambda$ and to within $\pm\lambda\varepsilon$ if $|x| < \lambda$. In contrast, $|x|$ must exceed λ/ε to make the same claim for S.Z. The use of extended range and precision would not make S.Z.'s disadvantages disappear, since if $|x|$ is very close to λ, the correction term, even if calculated to extra precision, may make x'_n underflow.

A more detailed analysis can be found in [14].

15. Acknowledgments. The results in this paper are the culmination of several years of discussions among all the members of the P754 Floating Point Subcommittee, not just the author and the four others mentioned in the introduction, so thanks are due them and their various institutions which supported their participation in the committee. The author wishes to acknowledge Prof. W. Kahan in particular for his suggestions and comments.

REFERENCES

[1] J. T. COONEN, *Underflow and the denormalized numbers*, Computer, 14 (1981), pp. 75–87.
[2] J. DEMMEL, *Effects of underflow on solving linear systems*, Computer Science Division, Univ. California, Berkeley, 1980.
[3] ———, *Effects of underflow on solving linear systems*, Fifth Symposium on Computer Arithmetic, Ann Arbor, MI, May 18–19, 1981.
[4] R. A. FRALEY AND J. S. WALTHER, *A proposed standard for binary floating point arithmetic: Alternate 3*, IEEE Floating Point Subcommittee Working Document P754/80-1.24, 1980.
[5] F. R. GANTMACHER, *The Theory of Matrices*, trans. K. A. Hirsch, Chelsea, New York, 1959.
[6] J. B. GOSLING, J. H. P. ZURAWSKI AND D. B. G. EDWARDS, *A chip-set for a high-speed low-cost floating-point unit*, Fifth Symposium on Computer Arithmetic, Ann Arbor, MI, May 18–19, 1981.
[7] D. HOUGH, *Errors and error bounds*, IEEE Floating Point Subcommittee Working Document P754/80-3.2, 1980.

[8] *A proposed standard for binary floating point arithmetic*, Draft 10.0 of IEEE Task P754, December 2, 1982.
[9] E. ISAACSON AND H. B. KELLER, *Analysis of Numerical Methods*, John Wiley, New York, 1966.
[10] W. KAHAN, 7094-II *system support for numerical analysis*, SHARE Secretarial Distribution SSD-159, Item C4537, 1966.
[11] ———, *Accurate eigenvalues of a symmetric tridiagonal matrix*, Technical Report no. CS41, Computer Science Dept., Stanford University, Stanford, CA, 1966.
[12] ———, *A Survey of Error Analysis*, in Information Processing 71, North-Holland, Amsterdam, 1972, pp. 1214–1239.
[13] W. KAHAN AND J. PALMER, *On a proposed floating point standard*, SIGNUM Newsletter, Special Issue, October 1979.
[14] W. KAHAN, *Aitken's extrapolation and Gaussian quadrature*, IEEE Floating Point Subcommittee Working Document P754/80-2.23, 1980.
[15] ———, *Why do we need a floating point arithmetic standard?*, IEEE Floating Point Subcommittee Working Document P754/81-2.8, 1981.
[16] ———, *Three questions about underflow*, IEEE Floating Point Subcommitee Working Document P754/82-7.6, 1982.
[17] D. KNUTH, *The Art of Computer Programming, Vol.* 2, Addison-Wesley, Reading, MA, 1969.
[18] S. LINNAINMAA, *Combatting the effects of underflow and overflow in determining real roots of polynomials*, IEEE Floating Point Subcommittee Working Document P754/80-2.23, 1980.
[19] W. OETTLI AND W. PRAGER, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right-hand sides*, Numer. Math., 6 (1964), pp. 405–409.
[20] R. D. SKEEL, *Scaling for numerical stability in Gaussian elimination*, J. Assoc. Comput. Mach., 26, (1979), pp. 494–526.
[21] ———, *Iterative refinement implies numerical stability for Gaussian elimination*, Dept. Computer Science Report, Univ. Illinois, Urbana, 1979.
[22] J. H. WILKINSON, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.
[23] ———, *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, NJ, 1963.