Handling Floating-Point Exceptions in Numeric Programs

JOHN R. HAUSER
University of California, Berkeley

There are a number of schemes for handling arithmetic exceptions that can be used to improve the speed (or alternatively the reliability) of numeric code. Overflow and underflow are the most troublesome exceptions, and depending on the context in which the exception can occur, they may be addressed either: (1) through a "brute force" reevaluation with extended range, (2) by reevaluating using a technique known as *scaling*, (3) by substituting an infinity or zero, or (4) in the case of underflow, with gradual underflow. In the first two of these cases, the offending computation is simply reevaluated using a safer but slower method. The latter two cases are cheaper, more automated schemes that ideally are built in as options within the computer system. Other arithmetic exceptions can be handled with similar methods. These and some other techniques are examined with an eye toward determining the support programming languages and computer systems ought to provide for floating-point exception handling. It is argued that the cheapest short-term solution would be to give full support to most of the *required* (as opposed to recommended) special features of the IEC/IEEE Standard for Binary Floating-Point Arithmetic. An essential part of this support would include standardized access from high-level languages to the exception flags defined by the standard. Some possibilities outside the IEEE Standard are also considered, and a few thoughts on possible better-structured support within programming languages are discussed.

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General—*standards*; D.3.2 [**Programming Languages**]: Language Constructs and Features—*control structures*; G.1.0 [**Numerical Analysis**]: General—*computer arithmetic*; *numerical algorithms*

General Terms: Algorithms, Design, Languages, Performance, Standardization

Additional Key Words and Phrases: Arithmetic, exception handling, floating-point

1. INTRODUCTION

Designers and implementors of all levels of the computer hierarchy (programming languages, systems, and hardware) are regularly asked to incorporate exception-handling features into their products; yet many have little familiarity with how these features might actually be used. More than all others, arithmetic exceptions seem to be especially mysterious. People often ask whether there can be any reasonable

This work was supported in small part by the Defense Advanced Research Projects Agency (DOD) under grant MDA972-92-J-1028 and by National Science Infrastructure grants CDA-8722788 and CDA-9401156.
Author's address: Computer Science Division, Soda Hall, University of California, Berkeley, CA 94720-1776.

response to an exception as seemingly conclusive as, say, floating-point overflow. The truth is, though, that a number of different techniques exist for handling arithmetic exceptions, depending on the context in which the exception occurs. In fact, it is often both easier and cheaper to respond to an exception after-the-fact than to prevent the exception from occurring in the first place [Demmel and Li 1994; Hull et al. 1994]. Conversely, when exception handling is not available, it is sometimes necessary to artfully evade exceptions, resulting in programs that exhibit no exceptional behavior but waste time doing so [Brown 1981; Parlett 1979].

In recent years, processor manufacturers have become increasingly suspicious that arithmetic exception handling is an unneeded nicety, of little value to their customers. Without a doubt, the main concerns of heavy users of computer arithmetic are accuracy and speed. If a poll were taken, probably few such users would express much interest in exception handling. The handling of arithmetic exceptions is an issue primarily for the implementors of functions such as complex division, or numeric libraries like LAPACK (*l*inear *a*lgebra *pack*age) [Anderson et al. 1995]. These routines are expected to be widely applicable and so must avoid being tripped up by exceptions that are simply an artifact of the way the calculation is performed [Demmel and Li 1994; Hull et al. 1994]. The authors of such routines naturally constitute only a small minority of the people writing numeric code, but the results of their work are incorporated into the work of many others. Programmers who use numeric libraries today are often paying for the lack of standardized, effective exception-handling features *without even knowing it.*

The IEC standard for Binary Floating-Point Arithmetic (commonly known as the IEEE Standard, and an ANSI standard) was carefully drafted to include a number of features for dealing with floating-point exceptions [Goldberg 1991; IEC 1989; IEEE 1985]. (IEC is the International Electrotechnical Commission.) Computer designs of the past decade have been nearly unanimous in adopting this standard for their floating-point arithmetic, so a significant level of support for exception handling exists in hardware today. And yet, there is little evidence so far that much use has been made of many of these special features, even by the people supposed here to have good reason to. However, this disordered state is to be blamed mostly on the lack of standardized access to these features from high-level languages such as Fortran or C. When high-level access is available at all, it generally varies from one manufacturer's machine to another. Programmers of numeric libraries usually have an interest in having their code be portable to as wide a range of platforms as possible, and so must avoid features that would make their code machine specific. Today there is a danger that some of the features of the IEEE Standard will disappear from future hardware before they ever had a chance to be useful.

A new international standard [ISO 1994] tries to redress the situation by prescribing (among other things) minimal facilities for floating-point exception handling within any high-level language. The ISO Language Independent Arithmetic Standard, Part 1, (LIA-1) does not assume conformance to the IEEE Standard; but it *is* designed to mesh harmoniously with the older standard nonetheless. An appendix to the standard makes specific recommendations for standardizing access to IEEE exception-handling features within Fortran, C, Ada, Common Lisp, BASIC, PL/I, Pascal, and Modula-2. Although laudably conceived, the new language standard has yet to prove its mettle in the marketplace. Unfortunately, neither the IEEE Standard nor the LIA-1 document says much to motivate the gamut of exception-handling features they require.

This article attempts to get at the heart of the matter by examining the ways in which different arithmetic exceptions may be handled in numeric programs. The emphasis will be on making numeric code run faster by eschewing convolutions whose only purpose is to avoid exceptions. Reasoning is occasionally given for some of the more inscrutable features of the IEEE Standard (and by extension, LIA-1), along with ideas for improvement. Overall, the article summarizes the system support needed for numeric exception handling. It is hoped that the information provided here will help inspire a more concerted and possibly creative effort at providing such support in the future.

2. FLOATING-POINT REVIEW

Before delving into the ways in which floating-point exceptions may arise, it will be worthwhile first to review the character of floating-point arithmetic. The LIA-1 standard specifies a general floating-point model in more detail than is attempted here. A more thorough review, also touching briefly on exception-handling issues, is provided by Goldberg [1991].

In addition to zero, a floating-point format contains numbers of the form

$$\pm d_0.d_1d_2d_3 \ldots d_{n-1} \times b^e,$$

where b is the floating-point **base** (or **radix**); each d_i is a *digit* ($0 \leq d_i < b$); and n is the number of digits. A floating-point number is **normalized** if $d_0 > 0$. The base and number of digits are generally constant for a particular format, and together these determine the **precision** of the format. Ordinarily, the size of the exponent e is also limited, giving the floating-point format a finite **range**. The double-precision format of the IEEE Standard, for example, has a base of 2 with 53 binary digits and requires that numbers be normalized (usually) and that $-1022 \leq e \leq 1023$. The range of this format covers numbers as large and small as $10^{\pm 307}$, and the precision is nearly 1 part in 10^{16}.

Because of the finite precision of floating-point numbers, floating-point arithmetic can only approximate real arithmetic. Every floating-point number is a real number, but few real numbers have floating-point equivalents. Consequently, floating-point operations (addition, etc.) are generally thought of as being composed of the corresponding real operation followed by a *rounding* step which chooses a floating-point number to approximate the real result. The symbols $\oplus$, $\ominus$, $\otimes$, and $\oslash$ are used to distinguish the basic floating-point operations from the real operations $+$, $-$, $\times$, and $\div$. The difference between the ideal real result r of an operation and its floating-point approximation f is measured as the relative error $\delta = |f - r| \big/ |r|$. (This relative error is considered to be 0 when $f = r = 0$.) Put another way, the rounded result can be said to be perturbed from the ideal result by a factor of $f/r = 1 + (f - r)/r = 1 \pm \delta$, where δ is again the relative error. For example, for a 3-digit decimal (base 10) format, if 5.166666 is the ideal result and 5.17 the rounded result, the relative error of the rounded result is $|5.17 - 5.166666| \big/ |5.166666| = 0.00064516$. The returned result 5.17 is equal to the ideal result 5.166666 times a perturbation factor of 1.00064516.

Of course, it is possible for the real result of an operation to be so large or so small that it falls outside the available floating-point range. This situation is known as a **range violation**. When the magnitude is too large, the result is said to **overflow** the available range. When it is too small, an **underflow** occurs.

In the absence of overflow or underflow, each operation has, as a practical matter, a maximum relative error which bounds how bad a floating-point result can be in relation to the ideal result for that operation. The maximum relative error is the *worst* relative error the operation exhibits for any set of operands. For instance, if the maximum relative error for the addition operation is known to be $\epsilon_\oplus$, then for any x and y one can guarantee that $x \oplus y = (x + y) \times \rho$ for some perturbation factor ρ satisfying $1 - \epsilon_\oplus \leq \rho \leq 1 + \epsilon_\oplus$.

These days it is common for the results of basic floating-point operations to be **rounded to nearest**, which means that in the absence of a range violation the result returned is always the floating-point number closest to the ideal real result. Clearly, this is the best that a floating-point arithmetic could be expected to do. With rounding to nearest, the maximum relative error is the same for all of the basic operations, namely $\epsilon = 1/(2b^{n-1})$, where b is the base, and n is the number of digits in the floating-point format.

In this article floating-point arithmetic will ordinarily be assumed to be normalized binary (base 2) with rounding to nearest for the basic operations—the same as required by the IEEE Standard. Given n bits of precision, the maximum relative error ϵ is thus $1/2^n$. The conclusions that follow should apply generally to other formats as well, but the details might differ. In addition, the maximum and minimum positive floating-point numbers within range (normalized) will be written as Ω and ω, respectively.

3. RANGE VIOLATIONS

The radius of a proton is 1.2×10^{-15} m. The distance from Earth to the farthest quasar is estimated as 1.5×10^{26} m. The mass of an electron is 9.1×10^{-31} kg. The mass of the Milky Way Galaxy is 2.2×10^{41} kg. The time it takes for a neutron to pass through the nucleus of an atom is 2×10^{-23} s. The age of the known universe is hypothesized to be less than 4×10^{17} s.

The ratio between the largest and the smallest of these numbers is 2.4×10^{71}. Given these realities, the IEEE Standard double-precision range of $10^{\pm 307}$ might seem ample enough to eliminate any threat of overflow or underflow. The same intuition, though, might also have trouble divining a use for imaginary or complex numbers. After all, imaginary numbers have no manifestation in the material world. No tangible quantity is measured in complex units. Yet complex numbers appear as intermediates in many scientific and engineering computations, as a consequence of the mathematics applied in those computations. In much the same way, numbers of extreme size can appear as intermediates in the calculation of practical, more terrestrial quantities.

Probably the single most common example given to demonstrate spurious range violations is the calculation of the 2-norm of a vector. For a vector with elements x_i, $1 \leq i \leq N$, its 2-norm (or simply "norm") is

$$\sqrt{\textstyle\sum_{i=1}^{N} x_i^2}.$$

A straightforward coding of this expression is

```
sum := 0;
for i := 1 to N do
  sum := sum + x[i]*x[i];
end for;
norm := sqrt(sum);
```

However, for many slightly large or small vectors, this code will fail due to overflow or underflow in evaluating one or more of the x_i^2 or in taking their sum, even though the norm itself would be representable as an unexceptional floating-point number. In fact, efficiently evaluating a vector norm without danger from overflow or underflow and with *only a single pass* over the vector elements has been shown by Blue [1978] to be a nontrivial problem.

As another example, Smith et al. [1981] have addressed the problem of computing specific values of "normalized Legendre polynomials," used in the calculation of angular momentum in quantum mechanics and elsewhere. The details of this problem are of little concern here, but a brief summary can be given. A particular normalized Legendre polynomial depends on two nonnegative integer parameters μ and ν and is written $\bar{P}_\nu^\mu$. To accurately calculate $\bar{P}_\nu^\mu(x)$ given specific values of μ, ν, and x, with $\mu < \nu$, Smith et al. advocate starting with the values

$$\bar{P}_\nu^{\nu+1}(x) = 0$$

and

$$\bar{P}_\nu^{\nu}(x) = \sqrt{\frac{1}{2}\frac{3\cdot 5\cdots(2\nu+1)}{2\cdot 4\cdots(2\nu)}}\,(1-x^2)^{\nu/2}$$

and then recursing using the formula

$$\bar{P}_\nu^{\mu-1}(x) = \frac{2\mu x}{\sqrt{(1-x^2)(\nu+\mu)(\nu-\mu+1)}}\,\bar{P}_\nu^\mu(x) - \sqrt{\frac{(\nu-\mu)(\nu+\mu+1)}{(\nu+\mu)(\nu-\mu+1)}}\,\bar{P}_\nu^{\mu+1}(x)$$

until the desired μ is reached.

According to Smith et al., this strategy works well for small ν and most x. However, if ν is more than a few decimal digits, and x is close to ± 1, the technique is hampered by the fact that the starting value $\bar{P}_\nu^\nu$ is extremely small, even though the desired value $\bar{P}_\nu^\mu$ may be well within range. For instance, $\bar{P}_{10000}^{10000}(-0.707)$ is approximately 0.5318×10^{-1504}, which is not representable in most double-precision formats. For comparison, $\bar{P}_{10000}^{0}(-0.707) \approx 0.8766 \times 10^{334}$, and the values for $0 < \mu < 10000$ fall between these extremes [Smith et al. 1981].

Because certain combinations of problems and computational methods give rise to exceptionally large or small numbers, the implementors of widely used library routines have at times invested considerable effort toward making their code resilient against possible overflows and underflows [Demmel and Li 1994; Hull et al. 1994]. One important library for scientific computation is LAPACK, a package of expertly crafted subroutines for performing linear algebra operations [Anderson et al. 1995]. (LAPACK subsumes the older LINPACK and EISPACK libraries [Dongarra et al. 1979; Smith et al. 1976].) The possibility of range violations in intermediate computations has had significant impact on the coding of subroutines in LAPACK for finding eigenvalues of symmetric tridiagonal matrices and singular values of bidiagonal matrices [Assadullah et al. 1992; Demmel et al. 1994]. Avoiding range violations in these routines requires extra work at execution time, which hurts the performance of these routines; yet without such precautions, disastrous overflow and underflow would occur for many perfectly reasonable matrices.

3.1 Extended Range

The obvious remedy to possible overflow and underflow is to evaluate potentially large or small intermediate quantities with greater range. When no format with enough range is supported by the machine hardware, a wider range must be simulated in some way. In principle, this is not difficult: an extended-range floating-point format can be constructed by pairing a machine integer i with an ordinary floating-point number f and treating the pair as representing the number

$$f \times B^i,$$

where B is a predetermined constant that is a power of the floating-point base. For instance, if f is a standard IEEE-format double-precision number, i is a 32-bit twos-complement integer, and $B = 2^{256} \approx 1.1579 \times 10^{77}$, the range of representable numbers is greater than $10^{\pm 165492990270}$. Subroutines like those in Figure 1 can be used to perform the basic arithmetic operations on this wide-range format. (In the subroutines, B is `EXPBASE`, for *exp*onent *base*.) This is essentially the technique employed by Smith et al. [1981] for calculating the normalized Legendre polynomials discussed above.

The problem with software-implemented arithmetic is of course its slow speed. Ideally, processors would provide hardware assistance for performing extended-range operations. For instance, given a conveniently fixed `EXPBASE`, a processor could support the `adjust` function directly in one or at most two machine instructions.[1] Extended-range addition would also be aided by the ability to perform floating-point multiplications and divisions by 2^a for integer values of a. Probably the most common calculations requiring extended range, though, are large products—that is, calculations of the form

$$\textstyle\prod_{i=1}^{N} x_i$$

for large N. Even if the product itself is known to be unexceptional, the likelihood that an overflow or underflow will occur while the product is being accumulated increases with the number of factors. Since extra range is needed only for the running product, the overhead of extended range can be kept relatively small, as follows:

```
product.exp := 0;
product.sig := 1.0;
for i := 1 to N do
  product.sig := product.sig*x[i];
  product := adjust(product);
end for;
```

[1]For many processors, this feature would be facilitated by keeping the `exp` values in floating-point registers.

```
subroutine adjust(x : bigFloat) : bigFloat =
  declare
    z : bigFloat;
  begin
    z := x;
    if abs(z.sig) > EXPBASE then
      z.exp := z.exp + 1;
      z.sig := z.sig/EXPBASE;
    elseif abs(z.sig) < 1/EXPBASE then
      z.exp := z.exp - 1;
      z.sig := z.sig*EXPBASE;
    end if;
    return z;
  end;

subroutine add(x,y : bigFloat) : bigFloat =
  declare
    sum : bigFloat;
  begin
    if x.exp > y.exp then
      sum.exp := x.exp;
      sum.sig := y.sig;
      Divide sum.sig by $2^{x.exp-y.exp}$;
      sum.sig := sum.sig + x.sig;
    else
      ... (the same with x and y reversed)
    end if;
    return adjust(sum);
  end;

subroutine mul(x,y : bigFloat) : bigFloat =
  declare
    product : bigFloat;
  begin
    product.exp := x.exp + y.exp;
    product.sig := x.sig*y.sig;
    return adjust(product);
  end;
```

Fig. 1. Subroutines for addition and multiplication of a software-based extended-range format. Here a `bigFloat` variable is comprised of two fields, a floating-point `sig` (significand) and an integer `exp` (extended exponent), representing the number $\mathtt{sig} \times \mathtt{EXPBASE}^{\mathtt{exp}}$. Subtraction and division subroutines would be very similar to their addition and multiplication counterparts shown here.

```
subroutine mul(x,y : bigFloat) : bigFloat =
  declare
    product : bigFloat;
  begin
    Attempt the following:
      product.exp := x.exp + y.exp;
      product.sig := x.sig*y.sig;
    If overflow or underflow occurs:
      product := mul(adjust(x),adjust(y));  (i.e., adjust arguments and try again)
    return product;
  end;
```

Fig. 2. The extended-range subroutine `mul`, rewritten using exception handling to defer calling `adjust` as much as possible.

Because the loop accumulating the product is so short, it is easy to see that a hardware implementation of `adjust` could have significant impact in this case.

Although hardware assistance could help speed up emulated extended range, such assistance rarely exists today. Without support from hardware, extended range is best avoided unless truly needed. Hence, rather than perform all calculations in extended range from the outset, it is usually better to attempt a calculation using the hardware-supported floating-point format first, and then to switch to the extended-range format only in the event that the hardware format is inadequate. Algorithmically, this optimization appears as

```
Attempt the following:
  Perform the calculation using the hardware arithmetic;
If overflow or underflow occurs:
  Perform the same calculation using emulated, extended-range arithmetic;
```

Typically, the hardware-supported format will suffice for most cases, and only the unusual (but still *valid*) cases will require the slower execution.

A similar trick can also be used to speed up the extended-range subroutines themselves by postponing calls to `adjust` until absolutely necessary. The `mul` subroutine, for example, could be rewritten as in Figure 2 to take advantage of overflow and underflow exceptions to indicate when exponent adjustments are needed. Whether this change results in a time savings will depend on the frequency with which adjustments must be made. In calculating a large product of ordinary floating-point numbers, one can generally expect adjustments to be rare. In fact, the code shown in Figure 3 for accumulating a large product with deferred adjustments should ordinarily be fast enough that a separate, first execution attempt using only the hardware format is unnecessary.

Unfortunately, these strategies of putting off work until proved necessary by a range violation are prohibited on any system that forces termination of a program on any overflow. In order for these schemes to work, a program must be able to detect when range violations have occurred and be allowed to perform an alternate computation when they have. Moreover, monitoring for the occurrence of overflow or underflow must not slow down the primary case significantly, or any advantage may be lost. Usually, range violations are conveniently detected by the arithmetic hardware, but on some systems this information can be accessed only at some cost.

```
product.exp := 0;
product.sig := 1.0;
for i := 1 to N do
  Attempt the following:
    temp := product.sig*x[i];
  If overflow occurs:
    while abs(product.sig) > 1 do
      product.exp := product.exp + 1;
      product.sig := product.sig/EXPBASE;
    end while;
    temp := product.sig*x[i];
  Else if underflow occurs:
    while abs(product.sig) < 1 do
      product.exp := product.exp - 1;
      product.sig := product.sig*EXPBASE;
    end while;
    temp := product.sig*x[i];
  product.sig := temp;
end for;
```

Fig. 3. Code to calculate $\prod_{i=1}^{N} x_i$ using a running product with extended range. Exception handling is used to defer adjustments to `product.exp` as much as possible. Equivalents of the `adjust` operations have been expanded inline for improved efficiency.

In principle, several programming languages provide a means for realizing these optimizations at a high level with a "termination" style of exception handling. Ada is the best-known such language [ANSI 1983; 1995; ISO 1995a]; its exception mechanism is based on the construct

```
begin
  statements
exception
  when exception '|' ...   '=>' statements
  ...
end;
```

The first set of statements is terminated as soon as an exception occurs. This construct neatly matches the optimizations above as follows:

```
begin
  Perform the calculation using the hardware arithmetic;
exception
  when OVERFLOW_ERROR '|' UNDERFLOW_ERROR  '=>'
    Perform the same calculation using emulated, extended-range arithmetic;
end;
```

Actually, it would be fairer to say that the construct above *would* neatly match the desired optimizations, if the `OVERFLOW_ERROR` and `UNDERFLOW_ERROR` symbols were really part of Ada. Instead of these, Ada defines a broader `NUMERIC_ERROR` class of exceptions, covering overflow, division by zero, and arithmetic domain violations in general, but *not* including underflow [ANSI 1983]. (The new Ada 95 standard renames `NUMERIC_ERROR` to `CONSTRAINT_ERROR` [ANSI 1995; ISO 1995a].) Underflows cannot be caught as exceptions within Ada at all. Of course, Ada is not alone in this regard: on practically all systems, underflows are substituted with zero (or an IEEE Standard subnormal number) without causing any exception trap. Although this response to underflows is often acceptable, it does not do for, say, a large product, since the running product can underflow when the true result would be quite reasonable or even extremely large. In order for a termination-style exception mechanism to be useful for extended-range emulation, the programmer must be able to specify when underflows should cause a trap rather than be substituted with zero or some other small value. Portable means for distinguishing these cases do not exist today.

For the applications above, it is not critical that the system be able to stop the first attempted computation very precisely when a range violation occurs. In the extreme, the computation being attempted could well be allowed to run to completion, if that makes sense, before the occurrence of any range violations is even considered. (Of course, if a range violation does occur, any results calculated in the first attempt would necessarily be suspect.) So long as care is taken to avoid wild or deadlocked behavior, the occurrence of overflow or underflow can simply be noted for the program to observe upon completion of the first attempt.

With this approach in mind, the IEEE Standard was defined so that, by default, overflows and underflows do nothing more than cause a flag to be set to indicate that the given range violation has occurred. Computation continues with a default value prescribed by the standard. The default result for underflow is a tiny value, often zero, that roughly approximates the desired result. The default result for overflow is an infinity value. These default values are unimportant to the current discussion, but they will be examined more closely in subsequent sections.

To give programs access to the exception flags, a system will typically define a `getflag` subroutine that takes an indicator of which flag to access and returns that flag's value. For instance, `getflag(EXC_OVERFLOW)` might be used to obtain the setting of the overflow exception flag. (The symbol `EXC_OVERFLOW` is assumed to be a system-defined constant.) A corresponding `setflag` subroutine takes a flag indicator and a new value for the specified flag. Preferably, `setflag` also returns the previous flag value, making an exchange of values possible in a single operation. In practice, `getflag` and `setflag` are often defined to act on multiple flags in parallel, with each flag corresponding to a different bit of a machine integer. However, this feature will not be considered here.

With IEEE Standard exception flags, the optimizations above appear as

```
oldOverflowFlag := setflag(EXC_OVERFLOW,FALSE);
oldUnderflowFlag := setflag(EXC_UNDERFLOW,FALSE);
Perform the calculation using the hardware arithmetic;
overflow := setflag(EXC_OVERFLOW,oldOverflowFlag);
underflow := setflag(EXC_UNDERFLOW,oldUnderflowFlag);
if ( overflow or underflow ) then
  Perform the same calculation using emulated, extended-range arithmetic;
end if;
```

The previous settings of the flags are preserved during the first attempt at the calculation so that the flags will ordinarily show what exceptions occurred that were never specially handled by the program. The programmer is responsible for

```
xMax := 0;
for i := 1 to N do
  if xMax < abs(x[i]) then
    xMax := abs(x[i]);
  end if;
end for;

sum := 0;
if xMax > LARGE then
  for i := 1 to N do
    xi := x[i]*SMALLSCALE;
    sum := sum + xi*xi;
  end for;
  norm := sqrt(sum)/SMALLSCALE;
elseif xMax < SMALL then
  for i := 1 to N do
    xi := x[i]*LARGESCALE;
    sum := sum + xi*xi;
  end for;
  norm := sqrt(sum)/LARGESCALE;
else
  for i := 1 to N do
    sum := sum + x[i]*x[i];
  end for;
  norm := sqrt(sum);
end if;
```

Fig. 4. Code to calculate the norm of a vector using scaling, and without exception handling. Underflows in the accumulation of `sum` are harmless assuming a small number like zero is substituted for the underflowed value. Both `LARGE` and `LARGESCALE` must be close to $\sqrt{\Omega}$, with `LARGE` $<\sqrt{\Omega}<$ `LARGESCALE`. (Recall that Ω and ω are the largest and smallest positive numbers within range, respectively.) Similarly, `SMALLSCALE` must be slightly smaller than $\sqrt{\omega}$, and `SMALL` must be slightly larger than $b^n\sqrt{\omega}$, where b is the floating-point base, and n is the number of digits. To protect accuracy, `SMALLSCALE` and `LARGESCALE` need to be powers of the floating-point radix b. Note that the loop at the top to find the maximum element must be executed every time, even when scaling is not needed.

ensuring that the first attempt will eventually terminate without any damaging effects should overflow or underflow actually occur. ¿From a high-level perspective, manipulating flags is a primitive albeit serviceable solution to the problem. Nevertheless, programs written to use the IEEE Standard exception flags tend not to be portable because the `getflag` and `setflag` subroutines have yet to be standardized across different systems.

3.2 Scaling

Another technique used to get the effect of wider range is scaling. With scaling, the input to a computation is first examined and then preadjusted so that all intermediate calculations will be within range. Afterward, the final result is adjusted back so that it corresponds with the original input. Scaling can be used with problems that are *homogeneous* or have a similar relationship between inputs and outputs. A function is homogeneous if $f(kx_1, kx_2, \ldots) = k^n \cdot f(x_1, x_2, \ldots)$ for some n. Another relationship that submits to scaling is $f(kx_1, kx_2, \ldots) = f(x_1, x_2, \ldots) + k$.

The vector norm problem is homogeneous because $\text{norm}(k\mathbf{x}) = k \cdot \text{norm}(\mathbf{x})$. With a proper choice for k, $\text{norm}(\mathbf{x})$ can be calculated as $k^{-1}\text{norm}(k\mathbf{x})$ without danger of overflow and without underflow seriously affecting the result. Figure 4 gives code to evaluate the norm of a vector using scaling. The code first determines what scaling is appropriate and then executes one of three loops corresponding to three different scaling factors. (One of the scaling factors is 1, which is equivalent to not scaling at all.) Compared with the straightforward implementation given earlier, this code spends a significant fraction of its time computing `xMax` which is rarely needed.

Scaling is the principle method by which linear algebra packages like LAPACK

```
Attempt the following:
  sum := 0;
  for i := 1 to N do
    sum := sum + x[i]*x[i];
  end for;
  norm := sqrt(sum);
  if norm < N*SMALL then
    sum := 0;
    for i := 1 to N do
      xi := x[i]*LARGESCALE;
      sum := sum + xi*xi;
    end for;
    norm := sqrt(sum)/LARGESCALE;
  end if;
If overflow occurs:
  sum := 0;
  for i := 1 to N do
    xi := x[i]*SMALLSCALE;
    sum := sum + xi*xi;
  end for;
  norm := sqrt(sum)/SMALLSCALE;
```

Fig. 5. Code to do the same thing as Figure 4 but starting with the straightforward calculation and then responding to any overflows that occur. Underflows are again assumed to be substituted with some small number, possibly zero. If the result of the first attempt is too small, underflow might have been a problem, and the norm is recalculated.

avoid range violations [Assadullah et al. 1992; Demmel and Li 1994; Demmel et al. 1994]. It is also used, for example, by Hull et al. [1994] to circumvent range violations in the evaluation of complex elementary functions such as complex sine and complex square root.

In the vector norm example in Figure 4, time spent choosing an appropriate scaling factor is usually wasted. For many of the situations involving scaling in LAPACK—as well as in the older LINPACK and EISPACK—significant time is wasted determining whether scaling is actually necessary [Demmel and Li 1994]. As with emulated extended range, it is often better to attempt a calculation first without scaling and then to resort to scaling only when it is proved necessary by overflow or underflow. Figure 5 shows how the vector norm calculation can be rewritten to use this technique. By applying a similar optimization to an actual LAPACK routine for solving triangular systems of equations, Demmel and Li have obtained speedups ranging from 43% up to as much as a factor of four on real machines.

Obviously, this trick is in essence identical to the one used with extended range above:

```
Attempt the following:
  Perform the calculation using the hardware arithmetic;
If overflow or underflow occurs:
  Perform the same calculation with scaling;
```

Hence, all of the same implementation concerns discussed in the previous section apply.

3.3 Substitution

Reevaluating with extended range or scaling is not always the cheapest solution to range violations. If $x \oplus y$ overflows in evaluating

$$3 + \frac{1}{x+y},$$

then the $1/(x+y)$ term is plainly irrelevant; the result would round to 3 even if evaluated with infinite range.[2] In this expression, if $x \oplus y$ overflows it is sufficient to substitute a large number for the unrepresentable result and continue. The same is true if, say, $x \otimes y$ overflows in evaluating $\arctan(xy)$. Extended range would simply be wasted effort in either of these cases.

In general, consider a calculation that evaluates some expression F, where F contains a subexpression G, so that $F = f(G)$ for an appropriately defined function f. If G evaluates to a positive number whose value cannot be represented because of overflow, it may be possible to evaluate $f(\Lambda)$ in place of $f(G)$ for some value Λ if $f(z) \approx f(\Lambda)$ for all $z \geq \Omega$. Likewise, if G is negative, it may be possible to substitute $-\Lambda$ for G if $f(z) \approx f(-\Lambda)$ for all $z \leq -\Omega$. For the example above, it is easy to see that, for all $z \geq \Omega$,

$$3 + \frac{1}{z} \approx 3 + \frac{1}{\Lambda} \approx 3$$

for any sufficiently large Λ.

In making a substitution, care must be taken to ensure that any subsequent range violations in evaluating $f(\Lambda)$ (or $f(-\Lambda)$) will be properly dealt with [Sterbenz 1974]. For instance, if $x \otimes x$ (i.e., x^2) overflows in evaluating

$$\frac{1 + \sqrt{9x^2+1}}{1 + \sqrt{x^2+1}},$$

then, mathematically, Ω (the largest floating-point number) could be substituted, since

$$\frac{1 + \sqrt{9z+1}}{1 + \sqrt{z+1}} \approx \frac{1 + \sqrt{9\Omega+1}}{1 + \sqrt{\Omega+1}} \approx 3$$

for all $z \geq \Omega$. However, making this substitution results in $9 \otimes \Omega$ subsequently overflowing again, and this time there is no value that can be substituted reliably. Generally, there is little point in making a substitution if a subsequent part of the calculation can also experience a range violation for which there is no reliable substitute. If $x \otimes x$ does *not* overflow in the expression above, but $9 \otimes (x \otimes x)$ does, there is no floating-point value Λ for which

$$\frac{1 + \sqrt{z+1}}{1 + \sqrt{x^2+1}} \approx \frac{1 + \sqrt{\Lambda+1}}{1 + \sqrt{x^2+1}}$$

for all x and all $z \geq \Omega$.

Some older architectures provide an option in which $\pm\Omega$ is substituted for all overflowed results [Sterbenz 1974]. IBM mainframes have long had an option within

[2]Actually, a floating-point format could be defined with so little range that this is not true. Suffice it to say no arithmetic in common use is so crippled.

Fortran for doing this, for instance. As a rule, if any value Λ can be successfully substituted for G in $f(G)$, then Ω can also be substituted, since if $f(z) \approx f(\Lambda)$ for all $z \geq \Omega$ as stipulated above, then it must be that $f(\Omega) \approx f(\Lambda)$; and so $f(z) \approx f(\Omega)$ for all $z \geq \Omega$. Hence, if any value is a good substitute, Ω tends also to be a good substitute. Of course, that does not mean that substituting $\pm\Omega$ is going to be successful for all overflows. Rather, one would like to be able to specify where in a program substitution of $\pm\Omega$ is acceptable so that a trap can be avoided in just those cases.

Instead of using Ω many machines have an infinity value, ∞, which is substituted by default on overflow [Goldberg 1991; IEC 1989; Sterbenz 1974]. Just as with Ω, if there exists some value that is a good substitute for a particular overflowed result, then ∞ tends also to be a good substitute. Even so, Ω and ∞ are not completely interchangeable as substitutes for overflowed values. There are cases in which Ω can be successfully substituted, while substituting ∞ ultimately leads to a spurious exception such as $0 \times \infty$. And Ω is clearly superior in this respect: the relative error of substituting ∞ is always infinite, whereas the relative error of substituting Ω—although potentially bad—is at least finite.

Nevertheless, when substitutions are performed *by default*, ∞ is a slightly safer substitute than Ω, simply because, if the substitution is *not* a good idea, ∞ is less likely to disappear quietly in subsequent computation than is Ω. Additions, subtractions, multiplications, and many other operations, when applied to infinite operands, either give infinite results or signal an invalid operation exception (e.g., $0 \times \infty$). Thus when substitution is misapplied, ∞ has a greater chance than Ω of either visibly propagating through the calculation or causing an exception trap. (The arithmetic of ∞ is discussed in greater depth in Section 4.) Note that there is no strict guarantee that this will happen [Brown 1981; Lynch and Swartzlander 1991]. Simply, ∞ is *more likely* to be noticed than Ω.

The IEEE Standard requires that $\pm\infty$ be substituted on overflow, but it mitigates the trouble this substitution may cause by raising an overflow exception flag that can be tested by the program. If substitution of ∞ is not appropriate, the overflow flag can be used to trigger an alternate action within the program. When ∞ is an acceptable substitute, nothing special need be done—although, ideally, the program would ensure that the overflow flag is not raised in this case as follows:

```
oldOverflowFlag := getflag(EXC_OVERFLOW);
Perform calculation in which overflows may be safely substituted with ±∞;
setflag(EXC_OVERFLOW,oldOverflowFlag);
```

This contrivance maintains the convention that exception flags reflect only those exceptions raised that may not have been handled safely.

Substitution is more commonly applied to underflows than overflows, although the situation with underflows is actually more complex. In principle, a condition similar to the one for overflow applies: given an expression $F = f(G)$, if G evaluates to a positive number whose value cannot be represented because of underflow, then λ may be substituted for G if $f(z) \approx f(\lambda)$ for all $0 < z \leq \omega$. A similar statement applies for negative underflows. For example, if $x \otimes y$ underflows (positive or negative) in evaluating the expression

$$3 + xy,$$

then zero or some other small number can be safely substituted for the underflowed product.

Problems arise with the more general form

$$u + G$$

for some expression G. If the evaluation of G underflows, then whether or not zero can be substituted depends on the magnitude of u. If u is zero or is otherwise an extremely small number, the underflowed value of G is not necessarily insignificant to the computation. On the other hand, if u could be *any* floating-point number with equal likelihood, chances are good that substituting zero is perfectly safe, because most u's would be large enough to overwhelm any underflowed term. Thus arises the *underflow dilemma*:

> *If all underflows are signaled as exceptional, most such signals will be false alarms because the underflows would have been absorbed in subsequent additions anyway. Yet any unsignaled underflow has the potential to introduce devastating inaccuracies in a calculation.*

Sterbenz [1974] illustrates how this dilemma can frustrate efforts to keep underflow under control.

It has already been observed that efficient emulation of extended range can depend on underflow exceptions being signaled. One would not want underflows to be silently replaced by zeros when evaluating a large product, $\prod_{i=1}^{N} x_i$. Conversely, the codes in Figures 4 and 5 for evaluating the norm of a vector with scaling *rely* on zero (or some other small value) being substituted on underflow. As with overflow, the choice of a zero-substitution policy ought to be made carefully, based on the circumstances of the calculation [Brown 1981].

Substitution is most convenient when it is supported by the underlying system. The most likely substitutes for overflow are $\pm\Omega$ and ∞; for underflow, the corresponding candidates are zero and $\pm\omega$. Substitution of zero on underflow is commonly considered the proper behavior; and the IEEE Standard requires that $\pm\infty$ be substituted on overflow. The other options are less prevalent. Few systems allow a programmer to specify where in a program substitution is appropriate and what value to substitute. The IEEE Standard has an inflexible substitution policy, but the exception flags provide at least a primitive means of correcting unwanted substitutions. As noted earlier, though, access to these flags has yet to be standardized across different systems, so portable programs taking advantage of this feature cannot yet be realized.

3.4 Gradual Underflow

Rather than simply substitute zero on underflow, the IEEE Standard employs a scheme called *gradual underflow* intended to increase the chances that underflow will be innocuous if it occurs. Figures 6 and 7 illustrate the concept for binary floating point [Coonen 1981]. A special unnormalized format is added to the bottom of the floating-point range, and results that fall below the underflow threshold ω are rounded to the closest representable number in this format. For very small underflowed results, the closest representable number will in fact be zero, which is just a special instance of the unnormalized format. ($0.00000000 \times 2^{-126} = 0.$)

$$\begin{array}{r r}
 & \ldots\qquad\qquad\quad \\
 & 1.\text{xxxxxxxx} \times 2^{-122}\qquad\quad \\
 & 1.\text{xxxxxxxx} \times 2^{-123}\qquad \\
 & 1.\text{xxxxxxxx} \times 2^{-124}\quad \\
 & 1.\text{xxxxxxxx} \times 2^{-125}\; \\
\omega \rightarrow & 1.\text{xxxxxxxx} \times 2^{-126} \\
 & 0.\text{xxxxxxxx} \times 2^{-126}
\end{array}$$

Fig. 6. Gradual underflow for a binary floating-point format with 9 bits of precision and with the same range as IEEE single precision. In the figure, numbers with different exponents are aligned according to the location of their true binary points; an "x" represents either a "0" or "1" bit. Ordinarily, numbers within range must be normalized, so the leading "1" is redundant and thus not actually stored. To support gradual underflow, a special encoding allows numbers with the smallest exponent (−126) not to be normalized.

$$\begin{array}{r r}
 & \ldots\qquad\qquad\quad \\
 & 1.\text{xxxxxxxx} \times 2^{-122} \\
 & 1.\text{xxxxxxxx} \times 2^{-123} \\
 & 1.\text{xxxxxxxx} \times 2^{-124} \\
 & 1.\text{xxxxxxxx} \times 2^{-125} \\
\omega \rightarrow & 1.\text{xxxxxxxx} \times 2^{-126} \\
 & 1.\text{xxxxxxx} \times 2^{-127} \\
 & 1.\text{xxxxxx} \times 2^{-128} \\
 & 1.\text{xxxxx} \times 2^{-129} \\
 & 1.\text{xxxx} \times 2^{-130} \\
 & 1.\text{xxx} \times 2^{-131} \\
 & 1.\text{xx} \times 2^{-132} \\
 & 1.\text{x} \times 2^{-133} \\
2\epsilon\omega \rightarrow & 1. \times 2^{-134}
\end{array}$$

Fig. 7. Another way of looking at gradual underflow. For small numbers out of range, precision tapers off until none remains. Results smaller than 2^{-135} are flushed to zero.

Underflowed quantities greater than $2\epsilon\omega$ are rounded to some number of bits less than the ordinary precision, where the number of bits is determined by the size of the underflowed value. The smaller the value, the fewer bits of precision are available. The normalized floating-point numbers are often called **normal** numbers in this scheme, to distinguish them from the **denormalized** or **subnormal** numbers of the special unnormalized format.

The common policy of flushing all underflows to zero leads to an abrupt loss of all precision for underflowed values. When subnormal numbers are added to the arithmetic, loss of precision from underflow is clearly more gradual. Accuracy has been said to "degrade smoothly" as values move from the underflow threshold ω down to zero. Nevertheless, this in itself is a poor argument for gradual underflow, since there is no *a priori* reason to believe that a *largely* inaccurate result is to be preferred over a *grossly* inaccurate one. If anything, a truly impossible result may be easier to recognize than one that is plausible but still incorrect.

Yet gradual underflow is not always inaccurate, as reflected in the following theorem:

THEOREM 3.4.1. *If x and y are floating-point numbers, and if $x \oplus y$ underflows to a subnormal number, then $x \oplus y = x + y$ exactly.*

In other words, when a subnormal number is the result of an addition or subtraction, it *never* requires rounding, despite the reduced precision of the subnormal format. (For subtraction, note that $x \ominus y = x \oplus (-y)$.) Hence the result is not inaccurate at all—it could not be *more* accurate, in fact. A proof of this theorem (and all subsequent theorems) is given in the Appendix. As an informal argument, consider that in order for rounding to be necessary, there must be something to round off; that is, there must be at least one nonzero bit beyond the rounding point. Such a bit would have to come from one of the operands; yet Figure 7 suggests that there can be no such operand. The skeptical reader should attempt to construct a counterexample.

By Theorem 3.4.1, underflows in additions and subtractions are entirely benign when gradual underflow is employed. This fact allows assertions such as the following to be made:

THEOREM 3.4.2. *If x and y are any floating-point numbers with $1/2 \le x/y \le 2$, then $x \ominus y = x - y$ exactly.*

Theorem 3.4.2 states that if x and y are close enough to one another, their difference will be computed exactly, without rounding. When underflow is not gradual, this theorem is true *only if* $x \ominus y$ does not underflow. Laws such as Theorem 3.4.2 make it possible for certain critical algorithms to be more compact, and hence more efficient. A rule that must be qualified with "unless underflow occurs" is of limited value when underflow is a real possibility. Theorem 3.4.1 guarantees that at least a few useful identities such as this one will not be undermined by the possibility of underflow [Coonen 1981; Demmel 1984; Kahan 1980]. Another important law that is an immediate corollary of Theorem 3.4.1 is that $x \ominus y = 0$ if and only if $x = y$. This rule can obviously be violated if underflows on subtraction are flushed to zero.

Addition and subtraction are of course not the only floating-point operations, and gradual underflow does not eliminate all problems with underflow. Although it is not possible to make as clean a statement as Theorem 3.4.1 for other floating-point operations, some useful facts can still be proved, such as the following:

THEOREM 3.4.3. *If u, x, and y are floating-point numbers, and if u is normal (nonzero, non-subnormal) and any underflows are gradual, then*

$$u \oplus (x \otimes y) = (u + (xy \times \rho)) \times \sigma$$

with

$$\frac{1-2\epsilon}{1-\epsilon} \le \rho \le \frac{1}{1-\epsilon} \quad \textit{and} \quad 1 - \tfrac{3}{2}\epsilon < \sigma < \frac{1}{1-\frac{3}{2}\epsilon}.$$

This theorem gives bounds on the apparent error involved in evaluating the expression $u + xy$. The rounding of the multiplication $x \otimes y$ introduces an error perturbation factor ρ, and the rounding of the subsequent addition introduces another perturbation factor σ. Recall that ϵ is the maximum relative error due to ordinary rounding. The ratio $(1-2\epsilon)/(1-\epsilon)$ can be rewritten as $1 - \epsilon - \epsilon^2 - \epsilon^3 - \cdots$ which is only slightly less than $1-\epsilon$; and likewise, $1/(1-\epsilon)$ and $1/(1-(3/2)\epsilon)$ approach $1+\epsilon$ and $1+(3/2)\epsilon$, respectively. If underflow could not occur for $x \otimes y$, the bounds on ρ and σ could each be strengthened to a single rounding error: $1-\epsilon \le \rho \le 1+\epsilon$ and $1-\epsilon \le \sigma \le 1+\epsilon$. Gradual underflow has the apparent effect of weakening these bounds slightly. On the other hand, when underflows are flushed to zero, then even without any cancellation the result can be off by as much as a factor of 2. The worst case occurs when the following two conditions are met simultaneously: (1) $u = \omega$ and (2) the product xy is slightly less than ω and so is flushed to zero.

Theorem 3.4.3 illustrates how, *for certain calculations*, gradual underflow has only slightly worse impact on accuracy than ordinary rounding. For instance, using this theorem it is easy to see that if all of the coefficients of a polynomial $a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ are normal, and if the polynomial is evaluated according to Horner's rule,

$$(\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots a_1)x + a_0,$$

then even if underflow occurs the polynomial will be evaluated almost as accurately as it would with extended range (assuming overflow does not occur). In similar fashion, gradual underflow can be shown to be no worse than rounding for a significant class of algorithms, including ones for

—finding the zeros of a polynomial,
—performing Gaussian elimination,
—determining a Cholesky decomposition,
—iteratively refining a solution to a set of linear equations,
—computing the eigenvalues of symmetric tridiagonal matrices,
—performing numerical quadrature, and
—accelerating the convergence of a sequence.

An analysis of each of these is catalogued by Demmel [1984]. Some additional details can be found in Demmel [1981]; other examples are given by Kahan and Palmer [1979] and Coonen [1981]. It should be noted that many times this behavior is dependent on a certain set of the inputs or results being normal, although typically this condition is known to be satisfied in advance. When it is not certain to be true, verifying the condition involves categorizing certain inputs at execution time as either normal or subnormal.

Gradual underflow does not render underflow harmless in all situations. An earlier draft of the IEEE Standard included a "warning" mode to provide some security against loss of precision [Coonen 1980; 1981; Coonen et al. 1979; Feldman 1981; IEEE Task P754 1981]. However, the proposal was rather complex, and the protection provided would have been incomplete [Fraley and Walther 1979]. Just as with zero-substitution, the appropriateness of gradual underflow can only really be determined within the context of the calculation.

Gradual underflow is nearly inconceivable without system support. First, an encoding for the subnormal numbers must exist within the available floating-point format. Then, either the arithmetic hardware must implement gradual underflow directly, or the processor must trap on underflows and on subnormal inputs so that arithmetic with subnormal numbers can be emulated by the system software. Attempting to emulate gradual underflow entirely at a high level is futile, due to the overhead involved.

Augmenting floating-point hardware to deal directly with subnormal numbers requires significant circuitry and can slightly degrade the speed of operations on even normal floating-point numbers. Consequently, trapping has been the traditional means of supporting gradual underflow, though the time required to take the trap quickly adds up if many subnormal numbers occur. On the other hand, as processors become more complex—for instance, issuing multiple operations in a single cycle and retiring operations out of order—the trapping hardware itself becomes more costly to implement and can itself add to the time needed to perform operations on ordinary floating-point numbers [Hennessy and Patterson 1990; Hwu and Patt 1987; Johnson 1991; Smith and Pleszkun 1985; Sohi and Vajapeyam 1987]. Hence, incorporating gradual underflow into a processor's arithmetic involves engineering tradeoffs that are becoming increasingly uncomfortable. Recently, one manufacturer has decreed that subnormal numbers will be supported on their processors only in a "degraded" mode in which all floating-point arithmetic is executed with less speed [DEC 1992; Sites 1993]. In the "fast" mode, zero is substituted on underflow, and subnormal inputs are identified with zero. The programmer can then choose to pay for gradual underflow only when it is needed. But if the gradual underflow mode is *too* slow, there may never be any advantage to using it.

Ever since its original proposal, gradual underflow has been the most contentious feature of the IEEE Standard [Cody 1981; Coonen 1981; Fraley and Walther 1979; Parlett 1979; Payne 1979]. Unfortunately, arguments on this subject have rarely been based on solid information on either side. The merits of gradual underflow ought to be weighed against the costs of implementation, but to date, a careful analysis remains to be done. In the meantime, a de facto conclusion to the debate may emerge if manufacturers decide to discard gradual underflow due to perceived weakness in demand.

3.5 Alternate Number Formats

Alternatives to the usual floating-point format have been proposed that provide so much range as to preclude any possibility of overflow or underflow. These include the *symmetric level-index* representation [Clenshaw and Olver 1984; 1987; Clenshaw and Turner 1988; Olver 1987; Turner 1989; 1993] and the so-called *universal representation of real numbers* [Hamada 1987], along with an older proposal by Matsui and Iri [1981]. Of course, ordinary floating point can be given arbitrary range by increasing the size of the exponent field (recall Section 3.1); however, these proposals purport to offer extraordinary range within the confines of a standard 32- or 64-bit format (corresponding to IEEE Standard single and double precision, respectively).

This feat is accomplished (as it only could be) by sacrificing precision for unusually large or small values. Stated in floating-point terms: as the exponent value grows (positive or negative), the exponent field increases in size, forcing precision to shrink. With many of these systems, in fact, all precision can ultimately be eliminated (an implied 1 remains), so that very large or small numbers are represented solely by their exponent value. As numbers continue to grow (or shrink), the *exponent* itself becomes more coarse, jumping to ever larger (or smaller) values.

While such schemes can sometimes be used effectively, for most purposes this sliding precision makes it difficult to establish much confidence in the accuracy of a computation [Demmel 1987]. Since these alternate formats are also no cheaper to implement than ordinary floating point augmented with gradual underflow, they cannot be considered a competitive solution to the problem of range violations.

4. POLES

A real function can be said to have a *pole* at a particular argument u if the function's value becomes arbitrarily large as its argument approaches u. The functions $1/x$ and $\log(x)$ both have poles at $x = 0$, while $\tan(x)$ has a pole at $x = k\pi/2$ for every odd integer k. Clearly, even with *infinite* range, there is no ordinary floating-point number that could be returned as the correct value of a function at a pole. Consequently, attempting to evaluate a function at a pole is considered an exception distinct from overflow. The division operation provides the most familiar example of a pole—so much so, in fact, that typically all attempts to evaluate a function at a pole are placed under the heading of "division by zero."

Although division by zero does not submit to extended range, substitution is sometimes possible just as for overflow. For instance, if x is zero in

$$3 + \cfrac{1}{1 + \cfrac{1}{x}}, \tag{1}$$

then Ω can be substituted for $1 \oslash x$ to obtain the correct result of 3 for the expression.

Some machines have an infinity value, ∞, that can be given as the result of a function at a pole. Adding a single infinity to the real numbers is a well-understood mathematical extension for making poles unexceptional (known as the *one-point compactification* of the reals [Folland 1984]). Arithmetic with infinity follows a consistent set of rules: $\infty \pm a = \infty \pm 0 = \infty \times a = \infty \times \infty = \infty/a = \infty/0 = -\infty = \sqrt{\infty} = \infty$, and $a/\infty = 0/\infty = 0$, where a is any finite, nonzero real number. The operations $\infty \pm \infty$, $\infty \times 0$, and ∞/∞ are all undefined. If the arithmetic of ordinary real numbers is called $\mathbf{R}$, let $\mathbf{R}_\infty$ stand for $\mathbf{R}$ extended with a single infinity and the rules of arithmetic for infinity.

By defining arithmetic with infinity as above, the following theorem can be stated:

THEOREM 4.1. *If f is a function over $\mathbf{R}_\infty$ of one or more variables, and f is composed of the basic operations $+$, $-$, $\times$, $\div$, and $\sqrt{\ }$ (along with constants), then*

$$\lim_{\substack{x_i \to 0 \\ \text{all } i}} f\left(\frac{1}{x_1}, \ldots, \frac{1}{x_n}\right) = f(\infty, \ldots, \infty),$$

provided both sides of the equation are defined.

Theorem 4.1 asserts that for a large class of interesting functions, the result of substituting ∞ on all division by zero exceptions (the right side of the equation) is exactly the limit value one would want (the left side of the equation), so long as the substitution does not lead to an undefined expression. For instance, if ∞ is substituted for $1 \oslash x$ in evaluating expression (1) above, the value 3 is obtained exactly as a matter of course. Figure 8 illustrates how, in the same way, automatic substitution of infinity obviates the need for costly checks for zero denominators in evaluating a continued fraction approximation of a function.

Extending $\mathbf{R}$ to form $\mathbf{R}_\infty$ makes division by zero no longer undefined, but in turn introduces new undefined cases $\infty \pm \infty$, $\infty \times 0$, and ∞/∞, some of which arise in operations like $+$ and $\times$ that did not have any before. This is generally considered a good trade, since the new set of undefined cases is in some sense smaller and less troublesome. However, certain laws that are true for $\mathbf{R}$ are *not* true for $\mathbf{R}_\infty$.

```
f := a[n];
i := n;
while i > 0 do
  i := i - 1;
  d := x + f;
  if d = 0 then
    i := i - 1;
    if i < 0 then
      Signal that the result is infinite;
    else
      f := a[i];
    end if;
  else
    f := a[i] + b[i]/d;
  end if;
end while;
```

(a)

```
  i := n;
restart:
  f := a[i];
  Attempt the following:
    while i > 0 do
      i := i - 1;
      f := a[i] + b[i]/(x + f);
    end while;
  If a division-by-zero exception occurs:
    i := i - 1;
    if i < 0 then
      Signal that the result is infinite;
    else
      goto restart;
    end if;
```

(b)

```
f := a[n];
for i := (n - 1) downto 0 do
  f := a[i] + b[i]/(x + f);
end for;
```

(c)

Fig. 8. Methods for evaluating a *continued-fraction* approximation to a function without spurious division-by-zero exceptions. Many functions $f(x)$ can be approximated by a so-called continued-fraction expression of the form $a_0 + b_0/(x + a_1 + b_1/(x + a_2 + \cdots + b_{n-1}/(x + a_n)\cdots))$, where all the b_i are nonzero. (a) Straightforward code that checks for zero denominators. (b) Code to handle division-by-zero exceptions after-the-fact. (c) Obvious implementation if $b \oslash 0 = \infty$ when b is nonzero.

For instance, in $\mathbf{R}$, if $x + a = x$ then a must be zero. This common-sense fact is an application of the familiar cancelation law for addition. With $\mathbf{R}_\infty$, however, $\infty + a = \infty$ for *all* finite a, not just zero, so the cancelation law does not always hold in $\mathbf{R}_\infty$. But then again, the cancelation law does not hold for *floating-point* arithmetic anyway, since $x \oplus a = x$ whenever $|x| \gg |a|$. Luckily, the identities lost by including infinity in the arithmetic either already do not apply to floating point or have a suitable analog in $\mathbf{R}_\infty$.

So far, a single *unsigned* infinity has been considered. With an unsigned infinity, there can be no unequivocal answer to the question of whether, say, $10 < \infty$, since if it were granted that $10 < \infty$, then $10 < \infty = -(\infty) < -10$, which implies that $10 < -10$. A slightly different arithmetic is obtained if $\mathbf{R}$ is extended, not with a single unsigned infinity, but with two distinct *signed* infinities, $-\infty$ and $+\infty$, with $-\infty <$ all finite numbers $< +\infty$ (the *two-point compactification of* $\mathbf{R}$). Signed infinities can be used to distinguish the limit as numbers grow large in the *positive* direction from the limit as numbers grow large in the *negative* direction. This allows a distinction to be made, for example, between $\exp(-\infty) = 0$ and $\exp(+\infty) = +\infty$. The basic arithmetic must also be adjusted, so that $(+\infty) + (+\infty) = +\infty$, $(-\infty) + (-\infty) = -\infty$, $(+\infty) \times (-\infty) = -\infty$, and so on. Addition of infinities with opposite signs remains undefined.

Unfortunately, signed infinities introduce some new problems. To begin with, there is no compelling argument for assigning a particular sign to $1/0$, although it does seem less perverse to choose $1/0 = +\infty$ rather than $-\infty$. But if $1/0 = +\infty$, then $-\infty$ is the only x for which $1/(1/x)$ is nowhere close to x. The solution adopted for the IEEE Standard is to have a sign on zero as well, so that the reciprocals of $+\infty$ and $-\infty$ are $+0$ and -0, respectively [Coonen 1981; Hough 1981]. Multiplication and division are defined to propagate these signs consistently; so, for example, $(+0) \times (-3) = -0$, $(-0) \times (-0) = +0$, and $(+\infty)/(-0) = -\infty$.

Though not as contentious as gradual underflow, the existence of separate positive and negative zeros may be the least understood feature of the IEEE Standard. Simply stated, signed zeros were included to help deal with discontinuities around zero that occur for many standardized functions [Kahan 1986]. For example, signed zeros make it possible for the complex elementary functions to obey important laws of symmetry that they otherwise could not, due to unavoidable discontinuities in the functions' values along the real and imaginary axes. (This topic is examined in detail by Kahan [1986].) The reciprocal function $1/x$ likewise has a discontinuity at zero when infinities are signed, and the sign on zero selects between a reciprocal of $+\infty$ and $-\infty$. Related to discontinuities at zero is the fact that the signs of underflowed quantities can be preserved. Signed zeros are also naturally incorporated within the usual sign-magnitude encoding of floating point—although that in itself was not an overriding factor in the decision to include them in the IEEE Standard.

While several problems are solved with signed zeros, a new one arises: now a sign must be chosen for the result of $x - x$. For lack of a better answer, the IEEE Standard assigns $x - x = +0$. But consider the function $f(x) = 1/(x - a) + 1/(a - x)$. Algebraically, the expression defining $f(x)$ simplifies to zero; and not surprisingly, $f(x)$ evaluates to zero for all $x \neq a$. Yet because $a - a = +0$, $f(a)$ jumps suddenly to $+\infty$, which is not the result one might hope for. The crux is that the continuity represented by Theorem 4.1 is not assured when zeros and infinities are signed; the theorem is simply not valid for ± 0 and $\pm\infty$. Observe on the other hand that when zeros and infinities are *unsigned*, $f(a)$ is undefined because it involves adding $\infty + \infty$.

Signed zeros can be as much of a nuisance in some circumstances as a convenience in others. Consequently, if zeros and infinities have signs, it is best if there is a way to choose at times to ignore those signs, and instead treat the values as though they were unsigned [Coonen 1980; Kahan 1986]. At one time, a draft of the IEEE Standard included separate *affine* and *projective* modes to allow the programmer to select whether infinities and zeros should be treated as signed (affine) or unsigned (projective) [Coonen 1980; Coonen et al. 1979; Feldman 1981; IEEE Task P754 1981; Kahan and Palmer 1979]. The projective mode was ultimately dropped, however, in the interest of reducing the complexity of the final standard.[3]

Like gradual underflow, infinities and signed zeros obviously require support within the system to be practical. However, unlike subnormals, relatively little expense is entailed in extending existing arithmetic hardware to incorporate these few special values.

[3]A remnant of the projective-mode proposal can be found on the Intel 8087 and 80287 floating-point coprocessors.

5. INDETERMINATE AND UNDEFINED CASES

In addition to range violations and divisions by zero, a number of indeterminate and undefined cases can arise, such as $0/0$ or $\sqrt{x}$ with $x < 0$. Such cases are closely associated with the notion of singularities, as explained below.

The expression $0/0$ is **indeterminate** because there is no unique quotient q for which $q \times 0 = 0$; this equation is solved by any finite number. $0/0$ occurs, for example, in the expression $\sin(x)/x$ when $x = 0$, or in

$$\frac{x+y}{\sqrt{x^2+y^2}}$$

when $x = y = 0$. For each of these expressions, the fact that $0/0$ can arise is cause for the existence of a *singularity* in the expression. An expression can be said to contain a **singularity** if it cannot be evaluated or is discontinuous at some combination of arguments and yet is defined and continuous for arguments *arbitrarily close* to those that cause a problem. The expression $\sin(x)/x$, for instance, is well behaved for arbitrarily small x (both positive and negative), yet is indeterminate for $x = 0$. (Actually, most mathematicians would define the concept of *singularity* a bit differently. Nevertheless, the definition given here will suffice as a rough approximation. The expression $\sin(x)/x$ has a singularity at $x = 0$ regardless [Apostol 1974].)

The singularity at $x = 0$ in $\sin(x)/x$ is called *removable* because $\lim_{x\to 0}\sin(x)/x$ is defined—it is equal to 1. Intuitively, the equation $y = \sin(x)/x$ has a smooth curve (x, y) everywhere except at $x = 0$, where there is a small gap in the curve because of the indeterminate expression $0/0$. Filling in the gap with the point $(0, 1)$ gives a curve that is smooth everywhere—one that is exactly like the original curve, except with the singularity (the hole) removed. The function

$$f(x) = \begin{cases} \sin(x)/x & \text{if } x \neq 0; \\ 1 & \text{if } x = 0; \end{cases}$$

is important in signal processing and is known as the *sinc* function. There is no convenient expression for this function that does not also exhibit a removable singularity.

Not all singularities are removable. The singularity in the second example above is not removable because

$$\lim_{\substack{x\to 0\\ y\to 0}} \frac{x+y}{\sqrt{x^2+y^2}}$$

is again indeterminate. (Mathematically, the value depends on the direction from which the limit is approached.) Singularities involving only a single variable are often removable, whereas those involving two or more variables tend not to be removable. But counterexamples exist to both these tendencies.

Other undefined cases can arise that are not associated with indeterminate expressions but are more fundamental. Examples include the square root or logarithm of a negative number, or the arcsine of a value with magnitude greater than 1. For these, there is *no* possible correct result—such as there is no real number z for which $z^2 = x$ or $e^z = x$ if x is negative. If ∞ is not available, $1/0$ is another example of an undefined case.

When indeterminate or other undefined cases occur during program execution, they ordinarily result in the signaling of an exception, either by a processor trap or through some other means such as the IEEE Standard *invalid* exception flag. If an indeterminate case represents a removable singularity, or otherwise if a correct substitute can be determined by context, it is obviously only necessary to substitute the proper value in order to continue. Whether these cases call for any special exception-handling support is debatable. If an indeterminate case is anticipated, it is usually an easy matter to test for the case in advance and thus prevent an exception from ever occurring in the first place. For instance, the sinc function defined above is easily coded without exception as

```
if x = 0 then
  sinc := 1;
else
  sinc := sin(x)/x;
```

On the other hand, the test for zero is redundant if the hardware performs the same check. If an exception mechanism exists that allows the explicit test to be omitted, the normal case may be sped up somewhat. (At least one scheme for achieving this is advocated by Kahan [Goldberg 1991; Sterbenz 1974].)

Before concluding, it would be well to say a few words about the value of 0^0. One common opinion holds that 0^0 must be indeterminate because $\lim_{x,y\to 0} x^y$ can be any arbitrary value (again depending on the direction in which the limit is approached). While this stance is not wholy unreasonable, it does have some unfortunate consequences. For example, if 0^0 is undefined, then the expression of a polynomial as $\sum_{i=0}^{n} a_i x^i$ is not valid for $x = 0$. This one special case can be avoided only by assuming $0^0 = 1$. Or consider the standard binomial theorem: $(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}$. If 0^0 is undefined, this theorem is valid for all real numbers x and y and all nonnegative integers n, *except* if one of x or y is zero, or if $n = 0$ and $y = -x$. If 0^0 is accepted as 1, the binomial theorem is true for all nonnegative integers n without exception.

Knuth argues that, if for no better reason, 0^0 ought to be identified with 1 simply for mathematical conciseness, so that these sorts of exceptional cases can be avoided [Knuth 1992]. Ultimately, such an argument rests on the fundamental definition of x^n when n is an integer: given that x^n is the product of n numbers all having the value x, x^0 is by definition the product of *zero* numbers, which ought to be 1 regardless of x. The value of x in x^0 is simply irrelevant. The fact that $\lim_{x,y\to 0} x^y$ is indeterminate only proves that the x^y function must be discontinuous at $x = y = 0$; it does not prevent 0^0 from having a defined value.[4] Kahan [1986] goes further and gives practical justification for $0^0 = 1$ with an analysis of the circumstances in which 0^0 is likely to arise in a computation.

Thus, 0^0 is an example of an expression that is best *not* undefined, in order that programs not have to check for it as a special case. Subsequent claims can also be

[4]A critic might note that essentially the same argument for multiplication gives $0 \times \infty = 0$. However, in this case the conclusion must be rejected because it leads to a contradiction: $0 = 0 \times \infty = (1/\infty) \times (1/0) = 1/(0 \times \infty) = 1/0 = \infty$. The contradiction arises only because ∞ is *defined* in such a way that $0 \times \infty$ is not necessarily 0. No such contradiction arises in defining $0^0 = 1$.

made for $\infty^0 = \sqrt[\infty]{0} = \sqrt[\infty]{\infty} = 1$ and $\log_0 1 = \log_\infty 1 = 0$.

6. SOFTWARE SUPPORT FOR FLOATING-POINT EXCEPTIONS

Like most other software, numeric programs are written in high-level languages. Even more than most other software, libraries of advanced numeric functions are also written in high-level languages (usually Fortran) with the intention that the code will be immediately portable across various machines and operating systems. Thus, in order for floating-point exception handling to be usable by numeric software, it must be accessible from high-level languages, and preferably in a way that is common across multiple platforms.

Based on the evidence above, the most basic programming tool needed for handling floating-point exceptions is a construct of the form

```
Attempt the following:
  Perform some calculation;
If one or more out of a specified set of exceptions occurs:
  Perform an alternate calculation;
```

Precise termination within the first attempted calculation is not crucial. However, imprecise termination may require more care by the programmer to ensure that the first attempt will not do something dangerous if an exception does occur.

Since one important use for such a construct would be for optimizing scaling and extended-range calculations, it must be possible to catch underflows as well as overflows with the construct. On the other hand, it would hardly be an advance if *every* floating-point underflow had to be handled through such a construct. Hence, as part of or in addition to a construct such as above, there needs to be a way for the programmer to select whether underflows will be automatically handled or whether they should be caught and some alternate code executed [Hull 1981]. Likewise, it is convenient to be able to choose to have overflows substituted with $\pm\Omega$, or with ∞ if it exists, and to choose to substitute ∞ on division-by-zero exceptions. If other options such as unsigned versus signed infinities are to be supported, control over these must be made accessible within high-level languages, too.

To summarize, numeric software needs the following:

(1) Programs need some means for first attempting one computation, and then if an exception occurs, performing an alternate computation. The exceptions of interest are usually overflow and underflow, but could at times be division by zero or an invalid operation. Knowing which exception actually occurred can save time in the alternate computation.

(2) Programs need control over whether range violations and divisions by zero are to be handled with this *attempted/alternate* form or whether they should be automatically substituted. Division by zero is best substituted with ∞, overflow with either ∞ or $\pm\Omega$, and underflow with either 0 or a subnormal approximation.

(3) Programs need control over any additional arithmetic modes, such as gradual underflow versus flush-to-zero on underflow or affine (signed) versus projective (unsigned) infinities.

Various features have been built or proposed for exception handling over the years, but none so far is entirely satisfactory for numeric code. A cursory look at the most important schemes follows. A more thorough survey and critique of language features for exception handling is provided by Hauser [1994].[5]

6.1 Termination Exception Mechanisms

More and more languages are being outfitted with a termination exception mechanism organized around a construct such as above. Ada has already been mentioned [ANSI 1983; 1995; ISO 1995a]; the upcoming ANSI/ISO standard for C++ is another important example [Koenig and Stroustrup 1993; ISO 1995b; Stroustrup 1991]. In these languages, the attempted computation is aborted as soon as an exception is discovered. Implementations are thus naturally based around hardware exception traps.

As noted earlier, the exception mechanism in Ada can be used to catch numeric exceptions such as overflow and division by zero. However, all floating-point exceptions fall under one generic heading of `NUMERIC_ERROR` (`CONSTRAINT_ERROR` in Ada 95), so it is not possible for a handler to distinguish among the different classes of floating-point exceptions that might occur. No mechanism exists in Ada for requesting that overflows or divisions by zero be substituted with ∞ or Ω. At the other extreme, underflows are never considered exceptional but are always substituted with zero. (Ada 95 sanctions gradual underflow.) Thus about half of the techniques listed in this article cannot be coded in Ada according to the standard.

The proposed ANSI/ISO standard for C++ also includes a termination exception mechanism. Two predeclared exception classes are defined for numeric "runtime errors": `range_error` and `overflow_error`. Despite defining these identifiers, the proposed standard leaves undefined what happens when a floating-point exception such as overflow occurs. This at least leaves open the possibility that an implementation will respond to floating-point exceptions by "throwing" one of the built-in exceptions, which can then be caught by the language construct. At least two compilers for Intel-80x86-based PCs claim to conform to the expected standard: Borland's C/C++ compiler (version 4) and Microsoft's C/C++ compiler (version 7). Neither allows floating-point exceptions to be caught by the language exception mechanism, although the floating-point hardware of the PC is capable of supporting such behavior.

With the ANSI/ISO standard C library (and hence in C and C++), the `signal` routine can be used to specify a subroutine to be called on a given system event [ISO 1990]. One of the defined events is an "arithmetic error," labeled `SIGFPE` (*f*loating-*p*oint *e*rror *sig*nal). However, according to the language standard, if `SIGFPE` is signaled by an intrinsic floating-point operation like addition or multiplication, and a handler subroutine for `SIGFPE` exists, then the behavior of the program is undefined either if the handler returns or if the handler calls any library routine other than `signal` (including, for instance, `longjmp` or `exit`). In other words, any action that a `SIGFPE` handler takes is implementation defined. Likewise, nothing in the standard C library or in C++ provides any control over exception substitutions, either to request that overflows and divisions by zero be substituted or to request

[5]The current article corrects some known errors in that work.

that underflows be trapped.

Although most numeric software is written in Fortran, standard Fortran has no features addressing floating-point exception handling at all.

6.2 IEEE-Style Exception Flags

Partly because of the difficulty of influencing the design of programming languages, the treatment of exceptions in the IEEE floating-point standard is an intentional compromise: substitutions are prescribed for all exceptions; but, at the same time, exception flags are raised that a program can test to control selection of an alternate computation. Gradual underflow is required, on the grounds that it is more robust than simple zero-substitution. Floating-point exception handling is thus minimally supported, so long as subroutines are available for examining and setting the exception flags. In particular, the exception flag scheme obviates any need to preselect whether floating-point exceptions should be either substituted for or trapped. In exchange, it does demand more care from the programmer to guarantee that program execution progresses properly in the face of exceptions.

Hardware support for the IEEE Standard has existed on a variety of platforms for some years. Nevertheless, portable exception handling as intended by the standard is not yet a reality, simply because there is as yet no standard library for accessing the IEEE exception flags. Among systems that do attempt to provide access to the flags, no two systems currently have the same interface.

For Intel-80x86-based PCs, Microsoft and Borland libraries include a few subroutines for setting the control modes and checking the status of an 80x87 floating-point coprocessor. The `_status87` routine returns the current coprocessor status, which includes the standard exception flags. The coprocessor status can be read and cleared in one operation by `_clear87`, and various operating modes can be set with `_control87`. No routine allows the coprocessor exception flags to be raised directly.

Apple Macintoshes support equivalents of the `getflag` and `setflag` routines in system libraries. In high-level languages, these routines go by the names of `TestException` and `SetException`, respectively. Additional library routines provide control over other IEEE Standard features such as directed roundings. The Macintosh `SetException` routine does not return the old value of the flag, so most flag manipulations require back-to-back calls to `TestException` and `SetException`.

Unix System V Release 4 also defines library routines that provide good control over IEEE Standard features. Two routines, `fpgetsticky` and `fpsetsticky`, give access to the IEEE exception flags. Unlike the Macintosh, the `fpsetsticky` routine does return the previous values of the flags. Also unlike the Macintosh, unfortunately, `fpsetsticky` sets *all* of the flags at once. Setting only a single flag requires first a call to `fpgetsticky`, followed by one or more logical operations, followed finally by `fpsetsticky`.

Sun workstations with SunOS have a single library routine called `ieee_flags` for manipulating various IEEE Standard features. The parameters to `ieee_flags` are all character strings, and the routine is bizarrely cumbersome. Examining the overflow flag, for example, requires the following in C:

```
overflow =
  (ieee_flags("get","exception","overflow",&out)>>fp_overflow) & 1;
```

Setting the overflow flag back to the same value requires a different incantation:

```
ieee_flags(overflow ? "set" : "clear","exception","overflow",&out);
```

In both cases, the `&out` argument is superfluous.

Judging from these results, companies may be depending on the initiative of others to generate and distribute standardized `getflag` and `setflag` routines for various machines. However, for the outsider, getting such routines to work with existing compilers and runtime systems is not always trivial. For example, recall that the usual pattern for accessing the flags is

```
oldFlag := setflag(exception name,FALSE);
Perform some calculation;
flag := setflag(exception name,oldFlag);
```

Compilers that reorder (schedule) code as an optimization often have no qualms about moving parts of the calculation out from between the subroutine calls if the variables involved in the calculation could not possibly be visible to the `setflag` subroutine. In extreme cases, the result of compiler optimization is something like

```
Perform some calculation;
oldFlag := setflag(exception name,FALSE);
flag := setflag(exception name,oldFlag);
```

Even after the programmer determines what is happening, finding a good way to selectively disable this optimization can be maddening. For code written in C, it is usually necessary either to declare some of the variables involved in the calculation as "`volatile`" or, alternatively, to declare them as global to the entire program. This disables the optimization, but at the expense of restricting register allocation of variables. Such a hack is unacceptable for a commonplace library. Few if any compilers offer an immediate solution to this problem.

The LIA-1 standard requires a language implementation to provide an exception flag mechanism if the language does not support floating-point exception handling in another way. At least four flags must be implemented—integer overflow, floating-point overflow, underflow, and undefined—and equivalents to the following routines must be made available:

clear_indicators(*exception set*)	clear all of the specified exception flags
set_indicators(*exception set*)	set all of the specified exception flags
test_indicators(*exception set*)	return **true** if any of the given flags is raised
current_indicators()	return the set of all raised exception flags

This set does not correspond exactly with the `getflag`/`setflag` pair, but it is roughly equivalent in functionality. Which collection is better is a matter of taste. Nevertheless, whether language implementors will eventually abide by LIA-1 remains to be seen.

6.3 More Abstract Language Constructs

Termination exception mechanisms like in Ada and C++ are supposed to terminate an unsuccessful computation as soon as possible after an exception occurs. However, none of the examples of numeric exception handling presented earlier depends on the immediate termination of a calculation signaling an exception. The IEEE exception flags scheme actually takes advantage of the fact that an immediate jump is not necessary; by raising a flag, making a substitution, and continuing, the IEEE Standard supports *both* an attempted/alternate form *and* a default substitution with a single, simple reponse to exceptions.

A detraction of the IEEE flag solution, though, is its obvious lack of structure. Instead of being forced to set and reset flags, one would ideally have available a language construct that more directly reflected the attempted/alternate algorithm structure. Such a construct would allow the important semantics of the attempted/alternate form to be divorced from what are more properly considered implementation concerns, such as whether traps or flags are used to catch exceptions.

As a candidate, Hull et al. [1994] propose a simple **enable-handle** construct:

```
enable
  Perform some calculation;
handle
  Perform an alternate calculation (if an exception occurred above);
end
```

The alternate code is executed if a range violation or "undefined" exception occurs during the first attempt. Hull et al. define their construct specifically to give no assurance at all as to how long execution of the first attempt might continue after an exception: an exception might cause the "enable" part to be aborted immediately with a trap, or it might raise an internal flag that is tested at completion of the attempt (or anything conceivably in between). Hull et al. also choose not to inform a handler which exception led to its being invoked. Rather, the alternate code must start over again from scratch in all cases.

Hull et al.'s **enable-handle** construct is not a complete solution to the needs listed earlier, but it does illustrate how some middle ground might be found between traditional well-structured termination constructs on the one hand and unstructured IEEE-style flag manipulation on the other.

7. CONCLUSIONS

This article has presented the most common techniques for handling floating-point exceptions in numeric code. Exception handling is not a *necessary* feature. In every instance, a solution can be coded that avoids any occurence of an exception. But standardized support for exception handling allows many numeric routines to be better optimized. Instead of pessimistic code that evades exceptions, a faster, optimistic style can be used in which exceptional cases are dealt with only if they actually arise. In addition, much exception handling can be automated if options like substitution and gradual underflow are available.

The algorithmic techniques in this article ought to be an important basis for evaluating any support for floating-point exception handling. Currently, the best expedient for satisfying existing needs would be for the industry to standardize and implement correctly a few library routines for accessing the hardware floating-point exception flags and for controlling any special operating modes such as gradual underflow versus zero-subsitution on underflow. The LIA-1 standard is a good target for consensus in this regard. In the long run, better-structured support for floating-point exception handling ought to be provided within the more general exeption-handling mechanisms of new programming languages.

More than any of the other techniques, gradual underflow is a subject of contention and will likely remain so for some time. However, gradual underflow performs a greater service than is generally realized and probably deserves to be maintained, especially as it continues to be a required part of the IEC/IEEE floating-point standard. Some techniques for reducing the cost of gradual underflow appear never to have been published, so it may be that implementation difficulties have been at least partially overstated.[6] On the other hand, if a faster zero-substitution mode can be made to coexist, there is no reason not to provide it.

It would be impossible to overstress the importance of high-level standardization for any exception-handling support. Features implemented with a different interface on each system are unlikely to be used, regardless of their potential for improving numeric programs that real people care about. The watchword among implementors of numeric libraries is nearly always *portability.* Only if systems pay attention to portability for exception handling will hardware that is nearly ubiquitous today actually begin to serve the people for whom it was intended.

APPENDIX

The theorems of the article are proved in this appendix.

THEOREM 3.4.1. *If x and y are floating-point numbers, and if $x \oplus y$ underflows to a subnormal number, then $x \oplus y = x + y$ exactly.*

Actually, a slightly stronger statement can be made:

THEOREM 3.4.1a. *If x and y are floating-point numbers and $|x + y| < 2\omega$, then $x \oplus y = x + y$ exactly.*

PROOF. The smallest positive subnormal number is $2\epsilon\omega$, and every finite floating-point number—positive or negative, normal or subnormal—is an integer multiple of $2\epsilon\omega$. (Refer back to Figures 6 and 7. Most floating-point numbers are very *large* multiples of this value.) Hence, x and y are each integer multiples of $2\epsilon\omega$, and consequently their *real* sum $x + y$ is also an integer multiple of $2\epsilon\omega$. But since $|x + y| < 2\omega$, $x + y$ is representable exactly, either as a normal floating-point number with minimum exponent, or as a subnormal number. (Refer to Figures 6 and 7 again.) Thus $x \oplus y = x + y$ exactly. □

THEOREM 3.4.2. *If x and y are any floating-point numbers with $1/2 \le x/y \le 2$, then $x \ominus y = x - y$ exactly.*

PROOF. Theorem 3.4.1 proves the case in which $x \ominus y$ underflows. Otherwise, assume without loss of generality that x and y are both positive, and that $x \ge y$. (Clearly x and y are required to have the same sign, and the theorem is not affected by whether that sign is positive or negative. Likewise, the theorem is symmetric with respect to x and y, so if $x < y$ simply swap the two.)

[6]Unfortunately, this topic is outside the scope of this article. The extent to which such techniques are known commercially is difficult to gauge.

Given that y is a floating-point number of n binary digits, y can be expressed as $b \cdot 2^e$ for some integers b and e with $2^{n-1} \leq b < 2^n$. Since $x \geq y$, $x = a \cdot 2^e$ for some integer a and the same exponent e. ¿From the theorem statement together with the assumption that $x \geq y$, we know that $y \leq x \leq 2y$, so $0 \leq x - y \leq y$. Substituting, it follows that $0 \leq (a-b) \cdot 2^e \leq b \cdot 2^e$, so $0 \leq a - b \leq b$. Since $b < 2^n$, we have $0 \leq a - b < 2^n$. Hence, $x - y = (a-b) \cdot 2^e$ is representable exactly as a floating-point number, and thus $x \ominus y = x - y$. □

Theorem 3.4.3. *If u, x, and y are floating-point numbers, and if u is normal (nonzero, non-subnormal) and any underflows are gradual, then*

$$u \oplus (x \otimes y) = (u + (xy \times \rho)) \times \sigma$$

with

$$\frac{1-2\epsilon}{1-\epsilon} \leq \rho \leq \frac{1}{1-\epsilon} \quad \textit{and} \quad 1 - \tfrac{3}{2}\epsilon < \sigma < \frac{1}{1-\frac{3}{2}\epsilon}.$$

Proof. If neither the multiplication nor the addition underflows, the theorem is trivial. If the multiplication does not underflow but the addition does, by Theorem 3.4.1 the addition is exact, so set $\sigma = 1$; and again the theorem is trivial. It remains to prove the theorem when $x \otimes y$ underflows. Define the absolute error of the multiplication $\alpha = (x \otimes y) - xy$, so $x \otimes y = xy + \alpha$. Since $|xy| < \omega$, we know $|\alpha| \leq \epsilon\omega$. (Observe Figures 6 and 7.) ¿From the theorem statement, we have that $|u| \geq \omega$. The remainder of the proof is divided into three cases:

Case $2\omega \leq |u + (x \otimes y)|$. Define $\tau = (u \oplus (x \otimes y))/(u + (x \otimes y))$. Since τ is the perturbation factor of a floating-point addition that does not underflow, $1 - \epsilon \leq \tau \leq 1 + \epsilon$. Set $\rho = 1$, and set

$$\sigma = \frac{\tau}{1 - \dfrac{\alpha}{u + (x \otimes y)}}.$$

Then

$$\begin{aligned}
(u + xy\rho)\sigma &= (u+xy)\frac{\tau}{1 - \dfrac{\alpha}{u + (x \otimes y)}} = (u+xy)\frac{\dfrac{u \oplus (x \otimes y)}{u + (x \otimes y)}}{1 - \dfrac{\alpha}{u + (x \otimes y)}} \\
&= (u+xy)\frac{u \oplus (x \otimes y)}{u + (x \otimes y) - \alpha} = (u+xy)\frac{u \oplus (x \otimes y)}{u + xy} \\
&= u \oplus (x \otimes y),
\end{aligned}$$

as required. Furthermore, since

$$\left| \frac{\alpha}{u + (x \otimes y)} \right| \leq \frac{\epsilon\omega}{2\omega} = \tfrac{1}{2}\epsilon,$$

it follows that $(1-\epsilon)/(1+(1/2)\epsilon) \leq \sigma \leq (1+\epsilon)/(1-(1/2)\epsilon)$. It is a simple matter to show that these bounds satisfy the necessary constraints on σ.

Case $|u + (x \otimes y)| < 2\omega$ *and* $(1-\epsilon)\omega \le |xy| < \omega$. Set $\rho = 1 + \alpha/xy$, and set $\sigma = 1$. Then $(u + xy\rho)\sigma = u + xy(1 + \alpha/xy) = u + xy + \alpha = u + (x \otimes y)$, which by Theorem 3.4.1a is exactly $u \oplus (x \otimes y)$. And since

$$\left|\frac{\alpha}{xy}\right| \le \frac{\epsilon\omega}{(1-\epsilon)\omega} = \frac{\epsilon}{1-\epsilon},$$

we have that $(1-2\epsilon)/(1-\epsilon) \le \rho \le 1/(1-\epsilon)$.

Case $|u + (x \otimes y)| < 2\omega$ *and* $|xy| < (1-\epsilon)\omega$. Define $k = |xy|/(1-\epsilon)\omega$, so $|xy| = k(1-\epsilon)\omega$ with $0 \le k < 1$. Set $\rho = 1 + k\alpha/xy$ and $\sigma = 1 + (1-k)\alpha/(u + xy + k\alpha)$. Then

$$\begin{aligned}(u + xy\rho)\sigma &= \left(u + xy\left(1 + \frac{k\alpha}{xy}\right)\right)\left(1 + \frac{(1-k)\alpha}{u + xy + k\alpha}\right)\\ &= (u + xy + k\alpha)\left(1 + \frac{(1-k)\alpha}{u + xy + k\alpha}\right)\\ &= (u + xy + \alpha) = u + (x \otimes y),\end{aligned}$$

which again by Theorem 3.4.1a is $u \oplus (x \otimes y)$. Moreover,

$$\left|\frac{k\alpha}{xy}\right| = \frac{|\alpha|}{(1-\epsilon)\omega} \le \frac{\epsilon}{1-\epsilon},$$

which once again implies $(1-2\epsilon)/(1-\epsilon) \le \rho \le 1/(1-\epsilon)$. To check the bounds on σ, first observe that $|xy| + |k\alpha| \le k(1-\epsilon)\omega + k\epsilon\omega = k\omega < \omega \le |u|$, and so $|u + xy + k\alpha| \ge |u| - |xy| - |k\alpha| \ge \omega - k(1-\epsilon)\omega - k\epsilon\omega = (1-k)\omega$. Consequently,

$$\left|\frac{(1-k)\alpha}{u + xy + k\alpha}\right| \le \frac{(1-k)\epsilon\omega}{(1-k)\omega} = \epsilon,$$

from which it follows that $1 - \epsilon \le \sigma \le 1 + \epsilon$. □

THEOREM 4.1. *If f is a function over $\mathbf{R}_\infty$ of one or more variables, and f is composed of the basic operations $+$, $-$, $\times$, $\div$, and $\sqrt{\ }$ (along with constants), then*

$$\lim_{\substack{x_i \to 0\\ \text{all } i}} f\left(\frac{1}{x_1}, \ldots, \frac{1}{x_n}\right) = f(\infty, \ldots, \infty),$$

provided both sides of the equation are defined.

Although relatively intuitive, it would be well first to make precise the meaning of limit points when ∞ is involved. If a sequence contains finitely many ∞'s (possibly none), then if the finite elements converge to a limit, that is the limit of the whole. If a sequence contains finitely many zeros, and if the reciprocals of the nonzero elements converge to zero, the limit of the sequence is ∞. A sequence for which neither of these cases applies has no limit.

For conciseness, capital letters are used in the following to denote vectors of argument values: $X = x_1, \ldots, x_n$, $A = a_1, \ldots, a_n$, etc. In addition, $a_k \to b$ will be written to mean that the limit of the sequence a_k is b.

Theorem 4.1 is a corollary of a more general theorem:

Theorem 4.1a. *If the following conditions are met:*

(1) f is a function over $\mathbf{R}_\infty$ of one or more variables,
(2) f is composed of the basic operations $+$, $-$, $\times$, $\div$, $\sqrt{\ }$, and constants,
(3) X_k is a sequence of vectors with $X_k \to A$,
(4) $f(X_k)$ is defined for each k, and
(5) $f(A)$ is defined;

then $f(X_k) \to f(A)$.

Proof. The proof presented here is by structural induction on the operations composed to define f. The base cases are when f is defined by a single constant or argument variable, such as $f(X) = 3$, or $f(X) = x_4$. Proofs of the base cases are simple. The inductive cases are naturally separated according to the outermost operation composed to define f:

Case $f(X)$ is Defined as $g(X) + h(X)$. Given that $f(A)$ is defined, $g(A)$ and $h(A)$ must both be defined. (If $g(A)$ or $h(A)$ were not defined, $f(A)$ would not be defined.) By the same argument, $g(X_k)$ and $h(X_k)$ must be defined for each k. Since g and h are each made up of fewer operations than f, the inductive hypothesis requires that $g(X_k) \to g(A)$ and $h(X_k) \to h(A)$. If $g(A)$ and $h(A)$ are both finite, it is well known that

$$\lim_{X \to A} f(X) = \Bigl(\lim_{X \to A} g(X)\Bigr) + \Bigl(\lim_{X \to A} h(X)\Bigr) = g(A) + h(A) = f(A)$$

(since addition is continuous over the reals). Consequently, $f(X_k) \to f(A)$ as desired.

It is not possible for both $g(A)$ and $h(A)$ to be ∞, as that would mean that $f(A) = \infty + \infty$, which is not defined. That leaves the subcase in which exactly one of $g(A)$ and $h(A)$ is ∞. Assume that $g(A) = \infty$ and $h(A)$ is finite, so $g(X_k) \to \infty$ and $h(X_k) \to h(A)$, with $h(A)$ finite. A consideration of the possibilities shows that the limit of the sequence of sums $g(X_k) + h(X_k)$ is ∞; hence $f(X_k) \to \infty = f(A)$ as desired.

Case $f(X)$ is Defined as $g(X) - h(X)$. This is nearly identical to the case of addition above.

Case $f(X)$ is Defined as $g(X) \times h(X)$. The only interesting subcases are when one or both of $g(A)$ and $h(A)$ are infinite. Assume $g(A) = \infty$. Given that $f(A)$ is defined, $h(A) \neq 0$. By the inductive hypothesis, $g(X_k) \to \infty$ and $h(X_k) \to h(A)$, where $h(A)$ could be finite or infinite. Under all possibilities, the sequence of products $g(X_k) \times h(X_k) \to \infty$, so $f(X_k) \to \infty = f(A)$.

Case $f(X)$ is Defined as $g(X) \div h(X)$. The values $g(A)$ and $h(A)$ cannot both be zero or infinity. The important subcases are when $g(A) = \infty$ and/or $h(A) = 0$, in which case $g(X_k) \div h(X_k) \to \infty$; or when $h(A) = \infty$, for which $g(X_k) \div h(X_k) \to 0$. Proof proceeds as above.

Case $f(X)$ is Defined as $\sqrt{g(X)}$. Obviously, $g(A)$ cannot be negative, as neither can any of the $g(X_k)$. $g(A)$ is either zero, positive finite, or infinity; in each subcase, $\sqrt{g(X_k)} \to \sqrt{g(A)}$. $\square$

ACKNOWLEDGEMENTS

This article would not be the same without the hard patience and efforts of William Kahan and Sue Graham, or without various contributions from Krste Asanović, John Boyland, Jim Demmel, Xiaoye Li, and of course the reviewers.

REFERENCES

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. 1995. *LAPACK Users' Guide, Release 2.0.* SIAM, Philadelphia, Pa.

ANSI. 1983. *American National Standard Reference Manual for the Ada Programming Language.* ANSI/MIL-STD 1815A-1983. American National Standards Institute, New York.

ANSI. 1995. *Americal National Standard for Information Technology—Programming Languages—Ada.* ANSI/ISO/IEC 8652-1995. American National Standards Institute, New York.

Apostol, T. M. 1974. *Mathematical Analysis*, 2d ed. Addison-Wesley, Reading, Mass.

Assadullah, M. M., Demmel, J., Figueroa, S., Greenbaum, A., and McKenney, A. 1992. On finding eigenvalues and singular values by bisection. *LAPACK Working Note 19.*

Blue, J. L. 1978. A portable Fortran program to find the Euclidean norm of a vector. *ACM Trans. Math. Softw. 4,* 1 (Mar.), 15–23.

Brown, W. S. 1981. A simple but realistic model of floating-point computation. *ACM Trans. Math. Softw. 7,* 4 (Dec.), 445–480.

Clenshaw, C. W. and Olver, F. W. J. 1984. Beyond floating point. *J. ACM 31,* 2 (Apr.), 319–328.

Clenshaw, C. W. and Olver, F. W. J. 1987. Level-index arithmetic operations. *SIAM J. Num. Anal. 24,* 2 (Apr.), 470–485.

Clenshaw, C. W. and Turner, P. R. 1988. The symmetric level-index system. *IMA J. Num. Anal. 8,* 4 (Oct.), 517–526.

Cody, W. J. 1981. Analysis of proposals for the floating-point standard. *Computer 14,* 3 (Mar.), 63–68.

Coonen, J., Kahan, W., Palmer, J., Pittman, T., and Stevenson, D. 1979. A proposed standard for binary floating point arithmetic. *SIGNUM Newslett. 14*, 4–12. Special issue.

Coonen, J. T. 1980. An implementation guide to a proposed standard for floating-point arithmetic. *Computer 13,* 1 (Jan.), 68–79.

Coonen, J. T. 1981. Underflow and the denormalized numbers. *Computer 14,* 3 (Mar.), 75–87.

DEC. 1992. *Alpha Architecture Handbook.* Digital Equipment Corporation, Maynard, Mass.

Demmel, J. 1981. Effects of underflow on solving linear systems. In *Proceedings of the 5th Symposium on Computer Arithmetic.* IEEE Computer Society Press, New York, 113–119.

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

Demmel, J., Dhillon, I., and Ren, H. 1994. On the correctness of parallel bisection in floating point. Tech. Rep. UCB//CSD-94-805, Computer Science Division, Univ. of California, Berkeley, Calif. Also available as *LAPACK Working Note 70*, `http://www.netlib.org/lapack/lawns/lawn70.ps`.

Demmel, J. W. 1987. On error analysis in arithmetic with varying relative precision. In *Proceedings of the 8th Symposium on Computer Arithmetic*, M. J. Irwin and R. Stefanelli, Eds. IEEE Computer Society Press, Washington, D.C., 148–152.

Demmel, J. W. and Li, X. 1994. Faster numerical algorithms via exception handling. *IEEE Trans. Comput. 43,* 8 (Aug.), 983–992.

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

Feldman, S. 1981. Language support for floating point. In *Proceedings of the IFIP TC 2 Working Conference on the Relationship between Numerical Computation and Programming Languages*, J. K. Reid, Ed. North-Holland, Amsterdam, 263–274.

FOLLAND, G. B. 1984. *Real Analysis: Modern Techniques and Their Applications.* John Wiley and Sons, New York.

FRALEY, B. AND WALTHER, S. 1979. Proposal to eliminate denormalized numbers. *SIGNUM Newslett. 14*, 22–23. Special issue.

GOLDBERG, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23,* 1 (Mar.), 5–48.

HAMADA, H. 1987. A new real number representation and its operation. In *Proceedings of the 8th Symposium on Computer Arithmetic*, M. J. Irwin and R. Stefanelli, Eds. IEEE Computer Society Press, Washington, D.C., 153–157.

HAUSER, J. R. 1994. Programmed exception handling. M.S. thesis, Univ. of California, Berkeley, Calif.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, Calif.

HOUGH, D. 1981. Applications of the proposed IEEE 754 Standard for floating-point arithmetic. *Computer 14,* 3 (Mar.), 70–74.

HULL, T. E. 1981. The use of controlled precision. In *Proceedings of the IFIP TC 2 Working Conference on the Relationship between Numerical Computation and Programming Languages*, J. K. Reid, Ed. North-Holland, Amsterdam, 71–84.

HULL, T. E., FAIRGRIEVE, T. F., AND TANG, P. T. P. 1994. Implementing complex elementary functions using exception handling. *ACM Trans. Math. Softw. 20,* 2 (June), 215–244.

HWU, W. W. AND PATT, Y. N. 1987. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, Washington, D.C., 18–26.

IEC. 1989. *Binary Floating-Point Arithmetic for Microprocessor Systems.* IEC 559:1989. International Electrotechnical Commission, Geneva.

IEEE. 1985. *IEEE Standard for Binary Floating-Point Arithmetic.* ANSI/IEEE Std 754-1985. Institute of Electrical and Electronics Engineers, New York.

IEEE TASK P754. 1981. A proposed standard for binary floating-point arithmetic. *Computer 14,* 3 (Mar.), 51–62. With introductory comments by David Stevenson.

ISO. 1990. *Programming Languages—C.* ISO/IEC 9899:1990(E). International Standards Organization, Geneva.

ISO. 1994. *Information Technology—Language Independent Arithmetic—Part 1: Integer and Floating Point Arithmetic.* ISO/IEC 10967-1:1994(E). International Standards Organization, Geneva.

ISO. 1995a. *Information Technology—Programming Language—Ada.* ISO/IEC 8652:1995. International Standards Organization, Geneva.

ISO. 1995b. *Information Technology—Programming Languages, Their Environments and System Software Interfaces—Programming Language C++.* ISO/IEC CD 14882. International Standards Organization, Geneva.

JOHNSON, M. 1991. *Superscalar Microprocessor Design.* Prentice-Hall, Englewood Cliffs, N.J.

KAHAN, W. 1986. Branch cuts for complex elementary functions; or much ado about nothing's sign bit. In *Proceedings of the Joint IMA/SIAM Conference on the State of the Art in Numerical Analysis.* Clarendon Press, Oxford, England, 165–211.

KAHAN, W. AND PALMER, J. 1979. On a proposed floating-point standard. *SIGNUM Newslett. 14*, 13–21. Special issue.

KAHAN, W. M. 1980. Interval arithmetic options in the proposed IEEE Floating Point Standard. In *Proceedings of the International Symposium on Interval Mathematics.* Academic Press, New York, 99–128.

KNUTH, D. E. 1992. Two notes on notation. *Am. Math. Mon. 99,* 5 (May), 403–422.

KOENIG, A. AND STROUSTRUP, B. 1993. Exception handling for C++. In *The Evolution of C++: Language Design in the Marketplace of Ideas*, J. Waldo, Ed. MIT Press, Cambridge, Mass., 137–171.

LYNCH, T. W. AND SWARTZLANDER, E. E. 1991. A formalization for computer arithmetic. In *Proceedings of the 3rd International IMACS-GAMM Symposium on Computer Arithmetic and*

Scientific Computing, L. Atanassova and J. Herzberger, Eds. North-Holland, Amsterdam, 137–145.

Matsui, S. and Iri, M. 1981. An overflow/underflow-free floating-point representation of numbers. *J. Inf. Process. 4,* 3, 123–133.

Olver, F. W. J. 1987. A closed computer arithmetic. In *Proceedings of the 8th Symposium on Computer Arithmetic*, M. J. Irwin and R. Stefanelli, Eds. IEEE Computer Society Press, Washington, D.C., 139–143.

Parlett, B. 1979. An open letter to the community of computer users. *SIGNUM Newslett. 14*, 2–3. Special issue.

Payne, M. H. 1979. Floating point standardization. In *Proceedings of the 19th IEEE Computer Society International Conference*. Institute of Electrical and Electronics Engineers, New York, 166–169.

Sites, R. L. 1993. Alpha AXP architecture. *Commun. ACM 36,* 2 (Feb.), 33–44.

Smith, B. T., Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe, Y., Klema, V. C., and Moler, C. B. 1976. *Matrix Eigensystem Routines: EISPACK Guide*. Lecture Notes in Computer Science, vol. 6. Springer-Verlag, Berlin.

Smith, J. E. and Pleszkun, A. R. 1985. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Silver Springs, Md., 36–44.

Smith, J. M., Olver, F. W. J., and Lozier, D. W. 1981. Extended-range arithmetic and normalized Legendre polynomials. *ACM Trans. Math. Softw. 7,* 1 (Mar.), 93–105.

Sohi, G. S. and Vajapeyam, S. 1987. Instruction issue logic for high-performance, interruptible pipelined processors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Washington, D.C., 27–34.

Sterbenz, P. H. 1974. *Floating-point Computation*. Prentice-Hall, Englewood Cliffs, N.J., 39–70.

Stroustrup, B. 1991. *The C++ Programming Language*, 2d ed. Addison-Wesley, Reading, Mass.

Turner, P. R. 1989. A software implementation of sli arithmetic. In *Proceedings of the 9th Symposium on Computer Arithmetic*. IEEE Computer Society Press, Washington, D.C., 18–24.

Turner, P. R. 1993. Complex SLI arithmetic: Representation, algorithms and analysis. In *Proceedings of the 11th Symposium on Computer Arithmetic*, E. Swartzlander Jr., M. J. Irwin, and G. Jullian, Eds. IEEE Computer Society Press, Los Alamitos, Calif., 18–25.

Received March 1995, Revised August 1995, Accepted August 1995