Frexp/Ldexp vs. Logb/Scalb

W. Kahan
May 23, 1988

Abstract: The functions mentioned in the title do similar things; they manipulate the exponent and significant-digits fields of floating-point numbers. But the functions *frexp* and *ldexp* found in the Math. Library of the language C are not so well defined as are *logb* and *scalb* recommended by the IEEE standards 754/854 for floating-point arithmetic. The latter functions make sense on all machines regardless of conformity to those standards, whereas the C functions may be unfortunate choices for machines whose floating-point arithmetic is Hexadecimal or Decimal instead of Binary. That is why a programmer might well prefer the logb and scalb functions whenever they are available, and why every run-time Math. library should provide them regardless of language.

Introduction

C started out as an idiosyncratic language for PDP-11 hackers who hated extra key-strokes. Now in its maturity it pretends to universality, and its devotees have to generalize its features to machines whose architectures diverge radically from the PDP-11's. Of all divergencies, those pertaining to floating-point cause the worst troubles; here is one example.

The definition of frexp(x, n) in some books about C cannot be correct. For instance, the otherwise very reliable book C: *a Reference Manual* by Harbison and Steele (1984, Prentice-Hall) says on p.273

> " **11.3.13 frexp**
>
> ```
> double  frexp(x, nptr)
>     double x ;
>         int *nptr ;
> ```
>
> **frexp** splits a floating-point number into a fraction f and an exponent n, such that the absolute value of f is less than 1.0 but not less than 0.5 and such that f times *radix* raised to the power n is equal to x, where *radix* is the radix used by the floating-point representation (typically 2). The fraction f is returned, and as a side effect the exponent n is stored into the place pointed to by *nptr*."

(A very similar definition appears on p. 322 of C *Programming Language* by Miller and Quilici (1987, Wiley) but without the phrase "the absolute value of", so it is wrong when $x < 0$.)

The trouble with the foregoing definition becomes evident when the radix exceeds 2. On a decimal machine, whose radix is ten, a number like $x = 2.9$ cannot be handled because $x > 1$ but $x/10 < 0.5$. To make the definition feasible, either replace "0.5" by "1/ *radix*" or replace "*radix*" by "2." Which of these replacements is best? It is not obvious.

Fixing up frexp

If compatibility with prior practice is paramount, restricting *radix* to 2 is best regardless of the machine's own radix. This definition of frexp can be implemented without any anomaly on a machine whose radix is actually a power of 2 , for instance on a Burroughs B65xx (Octal machine, radix = 8) or on an IBM 370 or its imitations (Hexadecimal machines, radix = 16); and then most old C programs that were intended to be portable to a wide range of machines with Binary floating point will run correctly after recompilation onto the machines with these larger radices. Those old programs are less likely to run correctly on a Decimal machine unless they are rewritten, regardless of how frexp is defined.

The ostensibly portable implementations of frexp and its inverse ldexp that I have seen in some Math libraries for C will run anomalously on non-binary machines and unnecessarily slowly on all. Here is the gist of those implementations of frexp :

```
*nptr = 0 ;
if ( x != 0.0 )
     {
          while ( fabs(x) >= 1.0 )  {  *nptr += 1 ;  x *= 0.5 ;}
          while ( fabs(x) < 0.5 )   {  *nptr -= 1 ;  x *= 2.0 ;}
     }
return (x) ;
```

When run on either an octal or hexadecimal machine, this program can strip off the last octal or hexadecimal digit of x. For example, if $x = F.EDCBA12ABCDEF$ on an IBM 370, then it will be replaced by successively halved values 7.F6E5D0955E6F7 , 3.FB72E84AAF37B , 1.FDB97425579BD and 0.FEDCBA12ABCDE0 of which the last will be returned instead of 0.FEDCBA12ABCDEF . A better program frexp(x,*nptr*), faster and more accurate, follows:

```
 /* When floating-point is  Binary,  Octal  or  Hex., ... */
#define  K = ... ;      /*       4  ,    3  ,       4       */
#define  M = ... ;      /*    16.0 ,   8.0 ,    16.0       */
 const double  R = 1.0/M ;
 {
   *nptr = 0 ;
   if ( x != 0.0 && finite(x) )
     {
       while ( fabs(x) >= 1.0 ) { *nptr += K ;  x *= R ; }
#ifdef CDC_CYBER_17x  /*... special code for a  CDC Cyber 176 */
       while ( fabs(x)-0.25 < 0.25 ) { *nptr -= 1 ;  x += x ; }
#else                 /*... ordinary code for everyone else   */
       while ( fabs(x) < R ) { *nptr -= K ;  x *= M ; }
       while ( fabs(x) < 0.5 ) { *nptr -= 1 ;  x += x ; }
#endif
     }
   return (x) ;   /*   COMPATIBLE WITH  ldexp  BELOW   */
 }
```

This program improves upon the former in four ways:

- A loop is avoided when $x = \pm\infty$ (infinity), which can be encountered on Cybers, CRAYs and machines that conform to IEEE 754/854; on other machines, finite(x) = 1 for all x. As is traditional, when $x = 0$, frexp($x, nptr$) = x and $*nptr = 0$; and by analogy the same holds when $x = \pm\infty$.
- The pseudo-zeros on CDC Cyber 17x's are handled as nonzero quantities; if you prefer to treat them as zeros, start the program with the statement $x* = 1.0$, and retain instead of removing the middle while-loop. (The pseudo-zeros on those Cybers are tiny numbers, less than twice as big as the underflow threshold, that are treated as usual by addition, subtraction and comparison, but treated as zeros by division and multiplication.)
- The appearance of 0.25 twice compensates for a peculiarity of *comparisons* on Cyber 17x's; otherwise they could not tell the difference between 0.5 and the next smaller number.
- The values of K and M have been chosen both to preclude any loss of accuracy on machines with radices 2, 8, or 16, and to make the program run faster.

Of course, the improved program is not so portable as the first one. That could be remedied with conditional compilation using compile-time environmental enquiries, provided all C compilers provided those things; but then all C compilers could just as well supply versions of frexp and ldexp in machine code that might run much faster than could any such program written in C.

Fixing up ldexp

The header definition of ldexp begins thus:

```
double  ldexp(x, N)
     double x ;
     int N ;
```

ldexp(x, N) should return either $radix^N x$ or $2^N x$ according to how frexp is defined; a compatible choice has to ensure that ldexp(frexp($x, \&N$), N) = x for all x, provided the compiler does not pass N to ldexp before it has been changed as a side-effect of frexp. Neither pow($radix, N$) $\times$ x nor pow($2, N$) $\times$ x is correct since they can over/underflow spuriously when the desired value of ldexp(x, N) would not. Here is a slow but serviceable program to return ldexp(x, N) = $2^N x$:

```
/*  K and M  are as defined above. */
const double a[] = { 1.0, 2.0,  4.0,  8.0 } ;
const double b[] = { 1.0, 0.5, 0.25, 0.125 } ;
int i, j, L ;
{
  i = abs(N) ;  j = i/K ;  L = i%K ;  /*  L = 0, 1, 2 or 3  */
  if ( N > 0 )
          {  for ( i=0 ;  i<j ;  i++ )  x *= M ;
             x *= a[L] ; }
  else if ( N < 0 )
          {  for ( i=0 ;  i<j ;  i++ )  x *= R ;
             x *= b[L] ; }
  return (x) ;  /*  COMPATIBLE WITH  frexp  ABOVE  */
}
```

This program will not over/underflow undeservedly, nor will it lose accuracy unnecessarily on a Binary, Octal or Hexadecimal machine; but it is too slow compared with machine code to be used for more than a model of what ldexp should do on machines with reasonably regular arithmetic. On a CDC Cyber 17x, this program malfunctions when $N > 0$ and x is a pseudo-zero; on a CRAY the program may overflow even though its result ought to be barely bigger than half the overflow threshold. Those perversities can be handled by a more elaborate ldexp program whose derivation will be left to those readers who need it, with condolences from the rest of us.

On non-Binary machines the definition $\mathrm{ldexp}(x, N) = 2^N x$ exposes its user to an unavoidable rounding error whenever $L > 0$ in the program above. That error is not often so harmful as to vitiate the usefulness of ldexp, but situations do exist when scaling by powers of the radix, rather than by mere powers of 2, is the only way to avoid potentially fatal rounding errors. Those are the situations that motivated the introduction of logb and scalb.

Logb and Scalb

The definitions of these functions are taken from the ANSI/IEEE Standard 854 for *Radix-Independent Floating-Point Arithmetic* (1987), item SH11460 available from the IEEE Inc., 345 East 47th St., New York NY 10017. The functions are not a mandatory part of that standard, but are two of twelve *Recommended Functions and Predicates* in an appendix.

Let β be the radix of the floating-point arithmetic; β must be either 2 or 10 to conform to IEEE 854, and 2 to conform to IEEE 754, but the definitions below work for any radix $\beta \geq 2$.

```
double scalb(x,N)
     double x ;
     int N ;
```

scalb(x, N) returns $\beta^N x$ for integral values N without computing β^N first. Overflow and underflow should be handled in the same way as for (decimal string) → (floating-point) conversion (when a floating-point datum, entered from a keyboard or read from a file as a decimal string in, say, ASCII, turns out to be too big or too close to zero); ideally this should produce a reasonable value like $\pm\infty$

or 0.0 by default, emit a signal, and continue computation unless the program has opted to abort instead. When $\beta = 2$, scalb = ldexp .

```
double logb(x)
     double x ;
```

logb(x) returns the integer part N of $\log_\beta \mid x \mid$ as a signed integer in x's floating-point format. For nonzero finite x

$$1 \le \beta^{-N} = \text{ abs(scalb}(x,\text{-(int)logb}(x))) \ < \beta.$$

Special cases: logb($\pm\infty$) = $+\infty$ without any signal, and
logb(NaN) is NaN without any signal, but
logb(0.0) = $-\infty$ and signals *divide-by-zero.*

One special case covers the *infinities* available on CDC Cybers and CRAYs as well as on machines that conform to IEEE 754/854. The second covers the *indefinites* on Cybers and CRAYs and the *reserved operands* on DEC VAXs, as well as *Not-a-Numbers* in IEEE 754/854. The *divide-by-zero* exception is associated with (finite nonzero)/0 expressions or any others that produce exactly infinite results from finite operands; they should produce $\pm\infty$ (or something as close to it as possible) by default, emit a signal, and continue computation unless the program has opted to abort instead.

When $\beta = 2$, frexp(x, *nptr*) sets $*nptr$ = (int)logb(x) + 1 except in the special cases; then $*nptr = 0$ by tradition but for no other good reason. The trouble with that tradition is that $*nptr = 0$ in two very different situations, $0.5 \le \mid x \mid < 1$ and $x = 0$, that can be confounded unless an extra test is inserted into the program that uses frexp. Moreover, some of the early machine code implementations of frexp for the PDP-11 and VAX would return $*nptr = -128$ when $x = 0$ because that was fastest; therefore prudent programmers might best assume no more about *nptr* in those special cases than Harbison and Steele say about them: nothing.

All special cases can be handled in a reasonable way by logb because it returns a floating-point value instead of one of type int. The floating-point value is convenient for most of logb's applications; the others that require (int)logb may have to prevent overflow during conversion from floating-point to type int by inserting tests like

```
if (fabs(logb(...)) < maxint) ...
```

with an aptly chosen value for maxint .

Lest the reader conclude that logb and scalb are now defined universally and unambiguously by the IEEE standards, three cautionary remarks are in order. First, the IEEE standards are not universally in force among computer systems. Although more arithmetic hardware conforms to IEEE 754 than to any other single specification, the non-conformists cannot be disregarded; among them are the IBM 370, DEC VAX, Univac 11xx, CDC Cyber, CRAY, Burroughs B65xx, ... Second, even if a machine's hardware does conform to IEEE 754, the compilers that run on it are unlikely to honor all the requirements of the standard, and the functions logb and scalb in particular may well be missing. Third,

even when they are present in the compiler or the run- time library, scalb(x, N) may be implemented as scalb(N, x) with its arguments reversed, as is the case in the Standard Apple Numeric Environment described in the *Apple Numerics Manual* (1986, Addison-Wesley); that environment also supplies the IEEE standards' recommended function copysign(x, y) with its two real arguments reversed! The reason for those reversals might make an amusing story were it not so perverse.

None the less, logb and scalb are defined well enough for every commercially significant machine, regardless of IEEE standards, and useful enough to justify implementation as intrinsic functions (compiled in-line) regardless of language. If not intrinsic, they surely merit inclusion in the run-time library among the few functions coded in machine-language. That is the point of this paper.

Some Applications of frexp, ldexp, logb and scalb

The crucial applications lie within the library programs that convert between the machine's floating-point format and the decimal strings, usually ASCII, that are sent to be printed or displayed on a screen. These programs are not normally expected to survive transportation to a machine with a different floating- point radix, although in principle they could be written portably using either of the frexp/ldexp or logb/scalb pairs defined above. If last-digit accuracy of conversion is unimportant, some codes using frexp/ldexp will function adequately regardless of which definition, ours with $radix = 2$ or another with $radix$ set to the machine's radix, is used for frexp and ldexp. But the most accurate codes are easiest to write with logb and scalb.

Similar comments apply to cosh, erf, exp, gamma, hypot (cabs), ln, sinh and sqrt.

Many matrix computations can be protected from spurious over/underflow by prescaling of data and postscaling of results. For best results prescaling should be performed exactly, without any rounding errors, since any rounding errors suffered during prescaling could easily far exceed those that are generated during subsequent computation. The benefits conferred by scaling are universally acknowledged but almost never realized in portable libraries of matrix-handling software because that software is coded mostly in Fortran, which lacks anything like logb and scalb in its standard library.

Logb and scalb are not always preferable to frexp and ldexp. Consider the product $P = x[1] \times x[2] \times ... \times x[m-1] \times x[m]$ when m is extremely big, so big that a partial product could easily over/ underflow even though P itself, or a quotient of two such products, is believed for good reason to lie well within range. The computation of P could be accomplished safely but slowly first by sorting the array $x[..]$ in order of magnitude, and then by multiplying alternately biggest and smallest $x[..]$'s ; but since sorting must consume time proportional to $m \ln(m)$ a faster way is worth considering. Computing $\exp(\Sigma_k \ln \mid x[k] \mid)$ is a bad idea because it can lose accuracy badly to roundoff; but a similar scheme based upon logb and scalb is just as accurate as the method that involves sorting and much faster if they are are compiled in-line. The scheme computes the product of terms scalb($x[k]$,-(int)logb($x[k]$)) simultaneously with the sum $s = \Sigma_k$ logb($x[k]$), after which P = scalb(product,(int)s). Replacing logb and scalb here by frexp and ldexp runs faster, and is almost equally accurate provided the implementation of frexp, like the one described above, incurs no rounding error, and provided the compiler does not overlook frexp's side-effect. Another very much faster way

uses the "Counting Mode" for over/underflows, implementable within the over/underflow trap handler on machines that support the IEEE standards' optional trapping modes as well as on the IBM 370 and the DEC VAX; see the book *Floating-Point Computation* by P. H. Sterbenz (1974, Prentice-Hall) or the report *A Portable Floating-Point Environment* by David Barnett (1987, University of California at Berkeley, for course CS281).