Computer System Support for Scientific and Engineering Computation

Lecture 7a - May 24, 1988 (notes revised June 24, 1988)

1 Numerical Analysis in C

Consider the problem of computing $x \cdot y/z$ when the final result does not overflow or underflow. Is there a way to compute this expression so that none of the intermediate results overflow or underflow? Although this appears to be a tricky problem in general, the C library provided with UNIX™ (and some other operating systems as well) contains ldexp and frexp library routines which make the problem quite easy. The definition of these functions is

$$\text{double ldexp(x,n)} = x \cdot 2^n$$

$$\text{double frexp(x,\&n)} = x/2^k \text{ where } k \text{ is chosen so that } 0.5 \leq |x/2^k| < 1.0$$

As a side effect, frexp sets the variable n to the value k. The variable x is a double, and n is of type int. The problem has the following one line solution:

$$x \cdot y/z = \text{ldexp(frexp(x, \&i)} \cdot \text{frexp(y, \&j) / frexp(z, \&k), i+j-k)} .$$

There are two problems with this solution. The first is minor; the compiler might evaluate the second argument to ldexp before the first, in which case i, j, and k would be unitialized, instead of having the values computed by frexp. This problem can be easily fixed by breaking the expression into two lines. The second is more serious: frexp and ldexp implicitly assume you are on a binary machine. It isn't obvious what ldexp should do on a non-binary machine with base β. If the definition is changed to $x\beta^n$ it will be exact, but will break code that assumed the base was 2. If it is left as $x2^n$ it won't be exact. The appendix contains more details on frexp and ldexp.

1.1 Changes to C

This suggests some changes that might be made to C and its standard libraries. The first would be to introduce logb and scalb and use them in place of frexp and ldexp in new codes. Logb and scalb are recommended in IEEE 854 and are similar to ldexp and frexp, except they are defined to use the machines's radix. This would solve the problem mentioned in the previous section, namely provide a way of computing $x \cdot y/z$ without overflow or underflow.

The second change has to do with multiword arithmetic. Most hardware provides access to the carry bit after an add instruction, enabling a very efficient multiword add to be programmed. However, most high level languages don't give the programmer access to the carry bit. That means that multiword arithmetic packages can't be written both portably and most efficiently. To be most efficient it would have to be in assembly language in order to get at the carry bit, but then it wouldn't be portable. This problem would be solved if the standard C library provided routines for multiword arithmetic. When C was ported to a new machine, an assembly coded version of the multiword arithmetic routines would be expected to reside on that machine. [1]

The third change has to do with multi-dimensional arrays. In C, the declaration for a one dimensional array doesn't need to specify the size of the array. Thus you can easily write a subroutine to sort an array of arbitrary size. However the situation is different for two dimensional arrays. The size of one of the array dimensions must be specified as a constant at compile time. This makes it awkward to write subroutines that handle two dimensional arrays. In particular, it complicates translating numerical routines from FORTRAN to C. Its not difficult to add this extension to C. In fact, the GNU C compiler allows array declarations to have variable dimensions.

2 The Case for Guard Digits

Since floating point hardware can only represent a subset of all floating point numbers, most calculations on floating point hardware will incur some error. An earlier lecture discussed how the addition of a guard bit could make subtraction more accurate. Does this one extra bit really matter? Here's an example where it does. The appendix discusses Heron's formula $\sqrt{s(s-a)(s-b)(s-c)}$. This formula is numerically unstable, and can give inaccurate results for triangles that are needle-like, that is, where the length of one side is close to the sum of the lengths of the other two sides. The appendix shows a way to rearrange the calculation so that it is stable. The proof uses the fact that if $1/2 \leq p/q \leq 2$ then $p - q$ is exact. This fact holds on hardware with a guard digit, but may not hold on other hardware.

In other words the guard digit is important not because it adds one more bit of accuracy, but because it guarantees an algebraic relation that lets you reason about computations.

3 Error Analysis of Inner Products

Recall that the classical model of roundoff error goes like this. If X, Y, and Z are the names of variables, and if x, y, and z are their values as represented in the computer, and if a program assigns $X = Y \otimes Z$, then $x = (y \otimes z)(1 + \xi)$, where $|\xi| \leq \epsilon$, and ϵ is a constant that depends only on the floating point hardware being used. We previously used this model to analyze polynomial arithmetic. In this section, we will use it to study scalar products. The formula for scalar product is $S = \sum_{j=1}^{n} A_j B_j$, or to put it another way, $S = S_n$, where $S_0 = 0$ and $S_j = S_{j-1} + A_j B_j$. In the classical model this means that

$$s_j = (s_{j-1} + a_j \cdot b_j \cdot (1 + \pi_j))/(1 - \sigma_j),$$

[1] A future lecture will present an efficient method of multiword arithmetic for machines that don't provide access to the carry bit.

where $|\pi_j| < \epsilon$, $|\sigma_j| < \epsilon$, and $s_0 = \sigma_0 = 0$. Multiplying thru by $1 - \sigma_j$ gives

$$(s_j - s_{j-1}) - a_j b_j = s_j \sigma_j + a_j b_j \pi_j,$$

and then summing on j gives the error

$$s_n - \sum_1^n a_j b_j = \sum_1^n a_j b_j \pi_j + \sum_2^n s_j \sigma_j,$$

and taking the absolute value gives the error bound

$$|s_n - \sum_1^n a_j b_j| \leq \epsilon(\sum_1^n |a_j b_j| + \sum_2^n |s_j|).$$

So if a program that computes an inner product also wants to compute an error estimate, it could use the following program to compute E.

```
S = 0; E = 0;
if N > 0 then {
    E = -|A[1] * B[1]|;
    for j = 1 to N do {
        P = A[j] * B[j];
        S = S + P;
        E = E + |S| + |P|
    }
}
```

A bound for the error would then be $\epsilon \cdot$E. This calculation shows the usefulness of an add-magnitude instruction (that is, an instruction that takes the absolute value of a register and adds it to another register). If a machine has an add-magnitude instruction, then the calculation above can be performed with fewer instructions and using fewer registers.

One thing you might notice about the error is that it can be bigger than the inner product $\sum a_j b_j$ itself. Is this is an anomaly that would go away with a different kind of error analysis? The answer is no. Interval analysis may give a slightly tighter error bound, especially on a base 16 machine. The reason is that the ulp can be smaller than ϵ by a factor of β, but this is ignored by classical error analysis. When β is 16 this could make error bounds from interval analysis as much as 16 times smaller than the one above. However when the inner product is close enough to zero, that is the vectors $\vec{a}$ and $\vec{b}$ are close to orthogonal, the error will become larger than the inner product no matter what kind of error analysis you use.

The error committed in computing inner products can be reduced slightly by sorting the numbers. However, a much better way of reducing the error is to use *distillation*, which will be covered later. Distillation improves precision by splitting a_i and b_i into two pieces of equal size, and computing $a_i b_i$ using four multiplies.

3.1 Backward Error Analysis

We can also do a backward error analysis for inner products. Looking at the formula

$$s_n = \frac{\frac{a_1 b_1 (1+\pi_1) + a_2 b_2 (1+\pi_2)}{1-\sigma_2} + a_3 b_3 (1+\pi_3)}{1-\sigma_3} \cdots,$$

we see that if we set

$$a_1'b_1' = \frac{a_1b_1(1+\pi_1)}{(1-\sigma_2)(1-\sigma_3)\cdots(1-\sigma_n)}$$

$$a_2'b_2' = \frac{a_2b_2(1+\pi_2)}{(1-\sigma_2)(1-\sigma_3)\cdots(1-\sigma_n)}$$

$$\cdots$$

then $s_n = \sum a_i'b_i'$ exactly. That is, the error in s_n can be thought of as coming from errors in the input. From the formula above, the error in computing an inner product of n terms can be accounted for by an error of about n places in the last digits of the a_i and b_i. For more details, see J.H. Wilkinson's book *Rounding Errrors in Algebraic Processes.*

Miriam Blatt's One-line C Program

to compute

$$Q = X*Y/Z$$

without spurious over/underflow:

Uses double ldexp(X, N) = $2^N * X$, and

double frexp(X, &N) = $X/2^N$, with

```
double  X ;
int     N ;
```

and frexp sets N as a SIDE-EFFECT so that

$1/2 \leq$ abs(frexp(X, &N)) < 1.0 .

```
double  X, Y, Z, Q ;
int     i, j, k ;

Q = ldexp( frexp(X,&i)*frexp(Y,&j)/frexp(Z,&k), i+j-k ) ;
```

FEATURES WORTH ADDING TO C:

1. logb & scalb, ultimately to supplant frexp & ldexp.

2. Multi-word integer ADD & SUBTRACT & MULTIPLY acting on arrays of type int

3. Ability to declare & reference MULTI-SUBSCRIPT ARRAY ARGUMENTS *inside* a subroutine just as they are declared & referenced outside before they are passed as arguments

i.e. arraytype A [m] [n]
↑ ↑ VARIABLE int's

cf. R. Stallman's GNU C

Running Error-Analysis of SCALAR PRODUCT

$S := 0$;
for $j = 1$ to N do $S := S + A_j * B_j$.
... Presumably now $S = \Sigma_1^N A_j * B_j$.

Actually $s_j = (s_{j-1} + a_j \cdot b_j \cdot (1 + \pi_j)) / (1 - \sigma_j)$
where $|\pi_j| < \varepsilon$, $|\sigma_j| < \varepsilon$, $s_0 = \sigma_1 = 0$.

$$\therefore \quad s_N - \Sigma_1^N a_j \cdot b_j = \Sigma_1^N a_j \cdot b_j \cdot \pi_j + \Sigma_2^N s_j \cdot \sigma_j$$
$$|s_N - \Sigma_1^N a_j \cdot b_j| \leq \varepsilon \cdot (\Sigma_1^N |a_j \cdot b_j| + \Sigma_2^N |s_j|) .$$

$S := 0$; $E := 0$;
if $N > 0$ then {
$E := -|A_1 * B_1|$;
for $j = 1$ to N do {
$P := A_j * B_j$;
$S := S + P$;
$E := E + |S| + |P|$ } }

... now $|S - \Sigma_1^N A_j * B_j| \lesssim \varepsilon \cdot E$.

Note usefulness of ADD·MAGNITUDE instruction to save register-space.

Choose a positive integer $m = 1, 2, 3, \ldots$.
Suppose for real x and y that $0 < x < m \cdot y$.

Must Computed $(x/y) < m$?

$m = 1$ Yes, computed $(x/y) < m$
for every radix, every reasonable rounding.

$m = 2$ Yes for radix $\beta = 2$;
No for even radix $\beta > 4$;
Try $x = \beta^p - 3$, $y = \frac{1}{2}\beta^p - 1$.

$m \geq 3$ No.
Try $\beta = 2$, $x = 3 \cdot 2^p - 4$, $y = 2^p - 1$.

Proof for $m = 1$: Say $\beta^{p-1} \leq y \leq \beta^p$, $0 < x < y$.

Max (x/y) occurs when $x = y - 1$.

Then $x/y = 1 - 1/y \leq 1 - \beta^{-p}$

$\therefore$ Rounded $(x/y) \leq 1 - \beta^{-p} < 1$

Representable!

Exercises

1. Prove that the computed value of

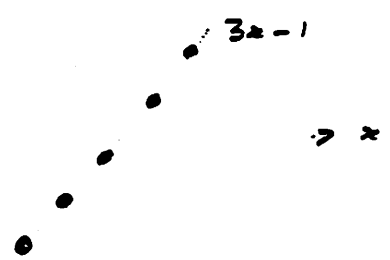
$$2.0 * X / (1.0 + X * X)$$

cannot exceed 1.0 for any floating-point X if arithmetic is ROUNDED correctly, or CHOPPED correctly.

2. For what RANGES of floating-point numbers X will the expression

$$(X - 0.5) * 2.0 + X$$

be computed EXACTLY? The answer depends upon the radix; try 2, 10, 16.



DOES SQRT(X*X) = ABS(X) ?

Try it for 10 sig. decimals ROUNDED correctly.
(NOT CHOPPED ; why ?)

Let $x = 0.3162277661$

$x^2 = 0.1000000053\ldots$

$\rightarrow y = 0.1000000001 = x*x$ rounded

$\sqrt{y} = 0.3162277661\,75\ldots$

$\rightarrow z = 0.3162277662 = \sqrt{y}$ rounded.

$\therefore$ SQRT(X*X) $\neq$ ABS(X) here!

For what proportion of numbers X
can we expect SQRT(X*X) = ABS(X) ?

$0.5 < x < \sqrt{10}$	All !
$\sqrt{10} < x < 5.0$	about 82 %
$\sqrt{10} < x < \sqrt{10.1}$	about 63 %

In general if $\sqrt{0.1} < \check{x} < x < \hat{x} < 0.5$
the proportion of SQRT(x^2) = x is
about $\check{x} + \hat{x}$.

Computer System Support for Scientific and Engineering Computation

Lecture 7b - May 24, 1988 (revision date June 24, 1988)

1 More on quotient

The floating point numbers are represented in a rather specific way in order to handle certain kinds of problems. One of them was the one that arose on the Cray. Here I generalize it a little bit. Assuming $0 < X < Y$, should you expect AMOD(X, Y) to return X and not, as the Cray did, to return $X - Y < 0$? I explained that happened on the Cray because the divide on the Cray is a little bit hard to predict. On all other machines, given $0 < X < Y$, the quotient $(X/Y)_{rounded}$ is always strictly less than 1. That is true on IBM machines, Burroughs', DEC's, calculators, Sun's, and so on, but not true on the Cray. And that was the problem that led to the letter (in real life) that I paraphrased for distribution in a previous lecture. Often AMOD is programmed as

$$X - \lfloor X/Y \rfloor \cdot Y.$$

The trouble with this formula is that the rounding errors in the quotient and the multiplication can cause too much damage even before the subtraction.

There are machines which do AMOD correctly in the sense that you get an exact result. For example: those machines that have IEEE remainder in them, the Dec VAX running under VMS (VMS fortran library), and APL running under IBM 370.

But on machines that merely use the above formula for AMOD, rounding each term, one could ask a reasonable question: assume $0 < X, 0 < Y$, will AMOD(X, Y) assuredly return a non-negative result? For some machines like IBM 370 the answer is yes, and easy to prove. On an IBM 370 the quotient and multiplication are chopped, making the result of computing $\lfloor X/Y \rfloor \cdot Y$ always a little smaller than X, and hence the subtraction is always non-negative. Perhaps it is for that reason that people haven't noticed that on many occasions the AMOD on IBM 370s returns a result that is bigger then Y, violating the definition of AMOD.

Now consider the following question: Choose the positive integer $m = 1, 2,$ or 3. Suppose for real x and y that $0 < x < m \cdot y$.

$$\boxed{\text{Must } computed(x/y) < m \text{ ?}}$$

when m=1: Yes, *computed*$(x/y) < m$ for every radix, every reasonable rounding.
Proof. Say $\beta^{p-1} \le y \le \beta^p, 0 < x < y$. Max$(x/y)$ occurs when $x = y - 1$. Then $x/y = 1 - 1/y \le 1 - \beta^{-p}$. Hence *computed*$(x/y) \le \underbrace{1-\beta^{-p}}_{\textit{Representable!}} < 1$

when m=2: Yes for radix $\beta = 2$; No for even radix $\beta > 4$; Try $x = \beta^p - 3, y = \frac{1}{2}\beta^p - 1$.

when m=3: No. Try $\beta = 2, x = 3 \cdot 2^p - 4, y = 2^p - 1$.

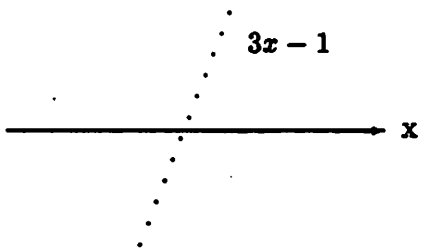
Exercises.

1. Prove that the computed value of

$$\frac{2.0 * X}{1.0 + X * X}$$

cannot exceed 1.0 for any floating-point X if arithmetic is rounded correctly, or chopped correctly.

2. For what *ranges* of floating-point numbers X will the expression $(X - 0.5) * 2.0 + X$ be computed *exactly*? The answer depends upon the radix; try 2,10,16.



Remark on exercise 1. Despite 3 or 4 rounding errors ($2.0 * X$ in general is not exact because the machine may not be binary), you can prove that the displayed quantity can never exceed 1. But you cannot prove it by using the ϵ-like model that we used for polynomial and scalar product. This is a valuable thing because often people who do this are going to compute something like $\arcsin(\frac{2x}{1+x^2})$ (the function arcsin cannot accept arguments greater than 1).

Remark on exercise 2. The second exercise has to do with the comment that was made toward the end of lecture 6. The expression $(X - 0.5) * 2.0 + X$ happens to be computed exactly in the neighborhood of the place it vanishes. The expression looks like $3x - 1$ rearranged in a funny way. It would vanish when X equaled 1/3, but 1/3 is not representable, so that can't happen. If X is near 1/3, it will in fact never vanish. If you believe an equation solver should be made to stop when the function vanishes, or when it is smaller then the rounding error in the computation, then this is an instance for which your program will never stop (because there is no rounding error).

2 Does SQRT$(X * X) = |X|$?

Everybody who works in the customer support department in a computing company is exposed to this problem from time to time when naive customers complain that the square root of a square (or the square of a square root) didn't come back. Here is an example using 10 significant decimals *rounded* correctly (not *chopped*; why not?):

Let	x	=	0.31622 77661	
	x^2	=	0.10000 00000 53...	
$\hookrightarrow$	y	=	0.10000 00001	$= x * x$ rounded
	$\sqrt{y}$	=	0.31622 77661 75...	
$\hookrightarrow$	z	=	0.31622 77662	$= \sqrt{y}$ rounded

Hence SQRT$(x * x) \neq |x|$ here.

Notice that in each instance rounding has been done in the best possible way. Since the relation is not satisfied in general we can ask for what proportion of numbers x can we expect SQRT$(x * x) = |x|$? We find

for $0.5 < x < \sqrt{10}$		All!
$\sqrt{10} < x < 5$		about 82%
$\sqrt{10} < x < \sqrt{10.1}$		about 63%

In general if $\sqrt{0.1} < \check{x} < x < \hat{x} < 0.5$ the proportion of SQRT$(x * x) = x$ is about $\check{x} + \hat{x}$. These figures are obtained experimentally but it is possible to show that you couldn't do better than that. It is impossible, no matter how you round, to satisfy the identity with higher proportion in those particular ranges. I am going to show you how to deal with this problem, up to a point.

3 How often at best can SQRT$(X * X) = |X|$?

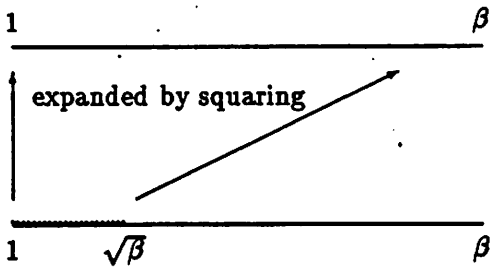
3.1 A Counting Argument.

Assume p significant digits of radix β in the usual notation. How many floating-point numbers lie in a β-ade (all the numbers that have the same exponent)? Since the answer depends only on the word size, we simply choose a typical β-ade interval $[\beta^{p-1}, \beta^p - 1]$ and count. It happens that all floating-point numbers in this interval are integers and one can easily see that there are $\beta^p - \beta^{p-1}$ of them.

1. Consider first $1 < \check{x} < x < \hat{x} < \sqrt{\beta} < \beta$. The range of $y = x^2$ is $1 < \check{x}^2 < y < \hat{x}^2 < \beta$. The number of x's and y's are

$$\begin{aligned} \#(x\text{'s}) &= \tfrac{\hat{x}-\check{x}}{\beta-1} \cdot (\beta^p - \beta^{p-1}) = (\hat{x} - \check{x}) \cdot \beta^{p-1} \\ \#(y\text{'s}) &= (\hat{x}^2 - \check{x}^2) \cdot \beta^{p-1} = (\hat{x} + \check{x}) \cdot \#(x's) \\ &> 2 \cdot \#(x's) \end{aligned}$$

   Hence x *could conceivably* be recovered from y: $x = \sqrt{y}$.



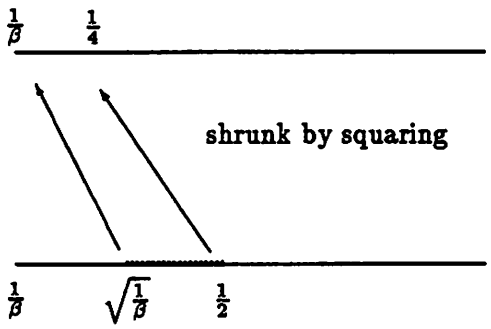
2. Consider second $\sqrt{\frac{1}{\beta}} < \check{x} < x < \hat{x} < 1$. The range of $y = x^2$ is $\frac{1}{\beta} < \check{x}^2 < y < \hat{x}^2 < 1$. The number of x's and y's are

$$\begin{aligned} \#(x\text{'s}) &= \tfrac{\hat{x}-\check{x}}{1-1/\beta} \cdot (\beta^p - \beta^{p-1}) = (\hat{x} - \check{x}) \cdot \beta^p \\ \#(y\text{'s}) &= (\hat{x}^2 - \check{x}^2) \cdot \beta^p = (\hat{x} + \check{x}) \cdot \#(x\text{'s}) \end{aligned}$$

Thus

> If $\hat{x} + \check{x} < 1$ then at most $(\hat{x} + \check{x})$ of the x's can be recovered from y's.

Hence if $\frac{1}{\beta} < \check{x} < x < \hat{x} < \frac{1}{2}$ then $\mathrm{SQRT}(x * x) = x$ at most $(\hat{x} + \check{x})$ of the time.



Exercise. How often at best can we expect $\mathrm{SQRT}(x) * \mathrm{SQRT}(x) = x$? Assume p significant digits of radix β, and consider $1 < \check{x} < x < \hat{x} < \beta$. Show that the fraction of such x's for which $\mathrm{SQRT}(x) * \mathrm{SQRT}(x) = x$ cannot exceed

$$\frac{1}{\sqrt{\check{x}} + \sqrt{\hat{x}}} < \frac{1}{2}.$$

3.2 More analysis on whether $\mathrm{SQRT}(x * x) = |x|$.

We need consider only the ranges $\max(\sqrt{1/\beta}, \frac{1}{2}) < x < 1$ and $1 < x < \sqrt{\beta}$, since a counting argument says that we cannot always expect $\mathrm{SQRT}(x*x) = x$ elsewhere in $\sqrt{1/\beta} < x < \sqrt{\beta}$. We will use the following notation:

$$\begin{aligned}[expression] &= \text{correctly rounded value of } (expression) \\ &= (expression) \pm \tfrac{1}{2}\text{ulp}(expression)\end{aligned}$$

$$\text{ulp}(x) = \begin{cases} \beta^{-p} & \text{if } \frac{1}{\beta} < x < 1 \\ \beta^{1-p} & \text{if } 1 < x < \beta \end{cases}$$

We also assume multiplication is correctly rounded. Define

$$\begin{aligned} y &= [x \cdot x] &&= x^2 + \eta \cdot \text{ulp}(x^2) && \ldots |\eta| \le \tfrac{1}{2} \\ z &= \text{SQRT}(y) &&= \sqrt{y} + \zeta \cdot \text{ulp}(\sqrt{y}) && \ldots |\zeta| \le \sigma \end{aligned}$$

where σ = error bound for SQRT program. SQRT is usually done in software, and the question is how accurate the software has to be in order to be able to satify the identity $z = x$. If SQRT is correctly rounded, then $\sigma = \frac{1}{2}$. Otherwise one could expect an error bound up to about a unit in the last place on a reasonable quality SQRT program. Note that the IEEE standard requires a correctly rounded square root. On a VAX, σ is less than something like 0.50001. So let's allow an error bound a little bit looser than the perfect SQRT program. We assume $1 > \sigma \ge \frac{1}{2}$.

The interval $\sqrt{1/\beta} < \{x : x \ne 1\} < \sqrt{\beta}$ is chosen so that we always have $\text{ulp}(x) = \text{ulp}(y) = \text{ulp}(z)$ (to simplify the error analysis). Here is the reasoning for $\sqrt{1/\beta} < x \le 1 - \beta^{-p}$:

$$\begin{array}{rccccc} & \sqrt{1/\beta} & < & x & \le & 1-\beta^{-p} \\ \Longrightarrow & \frac{1}{\beta} & \le & y = [x \cdot x] & \le & 1 - 2\beta^{-p} \\ \Longrightarrow & \frac{1}{\beta} & < & z = \text{SQRT}(y) & \le & 1 \\ \Longrightarrow & \multicolumn{4}{c}{\text{ulp}(x) = \text{ulp}(y) = \text{ulp}(z) = \beta^{-p}.} \end{array}$$

And here is the reasoning for $1 + \beta^{1-p} \le x < \sqrt{\beta}$:

$$\begin{array}{rccccc} & 1+\beta^{1-p} & \le & x & < & \sqrt{\beta} \\ \Longrightarrow & 1+2\beta^{1-p} & \le & y = [x \cdot x] & \le & \beta \\ \Longrightarrow & 1 & \le & z = \text{SQRT}(y) & < & \beta \\ \Longrightarrow & \multicolumn{4}{c}{\text{ulp}(x) = \text{ulp}(y) = \text{ulp}(z) = \beta^{1-p}.} \end{array}$$

So $y = x^2 + \eta \cdot \text{ulp}$ and $z = \sqrt{y} + \zeta \cdot \text{ulp}$. Now although z may not equal x, the difference must be a multiple of ulp. Hence $|z - x|/\text{ulp}$ must be a small integer, and "$z = x$" is equivalent to "$|z - x|/\text{ulp} < 1$". Our question is then

> How accurate must SQRT be (i.e., how small $\sigma \ge |\zeta|$) to imply $\frac{|z-x|}{\text{ulp}} < 1$?

This question turns out not very hard to answer. In general, evaluation of $f(p) - f(q)$ can be rewritten as $\frac{f(p)-f(q)}{p-q} \times (p-q)$. And $\frac{f(p)-f(q)}{p-q}$ can often be simplified a lot symbolically if f is an algebraic function (something that you compute with a finite number of $+, -, \times, \div$, and $\sqrt{}$, or you solve a equation with the left hand side of the equation computed that way). In our case, it is the square root function, and we have

$$\sqrt{p} - \sqrt{q} = \frac{\sqrt{p} - \sqrt{q}}{p - q}(p - q) = \frac{1}{\sqrt{p} + \sqrt{q}}(p - q).$$

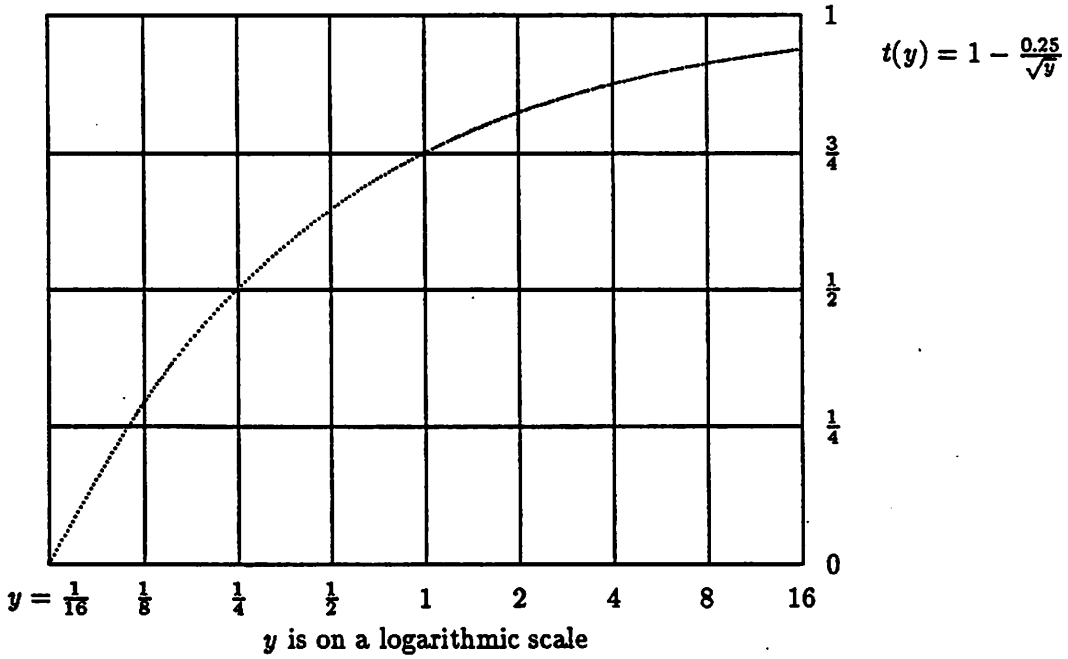
Thus,

$$\begin{aligned}\frac{z-x}{\text{ulp}} &= \frac{(\sqrt{y}+\zeta\cdot\text{ulp})-\sqrt{y-\eta\cdot\text{ulp}}}{\text{ulp}} \\ &= \zeta+\frac{\eta}{\sqrt{y}+x} \qquad \ldots x=\sqrt{y-\eta\cdot\text{ulp}} \\ &\approx \zeta+\frac{\eta}{2\sqrt{y}}\end{aligned}$$

near enough provided the radix is not equal to 4 (you have to do something differently when $\beta = 4$). Therefore, if we take a bound for ζ and a bound for η, we arrive an approximate bound

$$\frac{|z-x|}{\text{ulp}} \lesssim \sigma + \frac{1/2}{2\sqrt{y}}$$

If $\sigma \le t := 1 - 0.25/\sqrt{y}$ then surely $|z-x|/\text{ulp} < 1$, and hence $\text{SQRT}(x*x) = x$. The critical value of t is $\frac{1}{2}$, since we *can't* keep $\sigma < \frac{1}{2}$. Here we plot the graph of $t := 1-0.25/\sqrt{y}$ as a function of y:



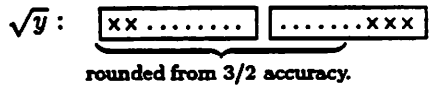
y is on a logarithmic scale

The graph above shows that, if $y = x^2$ falls into the region where $t(y) \ge \frac{1}{2}$, then it is possible that $\text{SQRT}(x*x) = |x|$, provided that $\sigma \le t(y)$. For a binary machine the range for $y(= x^2)$ is $\frac{1}{2} \le y \le 2$. On that range $t(y) \ge t(\frac{1}{2}) = 1 - 1/\sqrt{8} = 0.6464466\ldots$. So for a binary machine you don't need a perfect SQRT program in order to get $\text{SQRT}(x*x) = |x|$. Therefore, checking that identity on a binary machine is not a good test for square root. In quaternary, the bound is not good enough around $y = \frac{1}{4}$ and we will look at that later.

What about a hexadecimal machine? If you have a number in $[\frac{1}{4}, \frac{1}{2}]$, then its square will fall into $[\frac{1}{16}, \frac{1}{4}]$. Since $t(y) < \frac{1}{2}$ in that region, there is just no way to get the square root of a square always to come back. There are simply not enough eligible squares (certain distinct numbers in that region will have their squares coalesced after rounding). But we do expect the square root of a square to come back if the SQRT is correctly rounded and $y \ge \frac{1}{4}$.

3.3 Conclusions

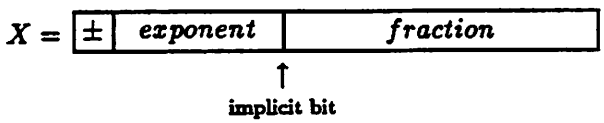
1. if $\sigma < \frac{3}{4}$ (i.e. error in SQRT(Y) is bounded by $\frac{3}{4}$ulp($\sqrt{Y}$)) then SQRT($X * X$) = X throughout $1 \le X \le \sqrt{\beta}$.
2. If $\sigma < 1 - 1/\sqrt{8} = 0.6464466\ldots$ and $\beta = 2$ then SQRT($X * X$) = $|X|$ for *all* X (a benefit of *binary*).
3. If $\sigma = \frac{1}{2}$ (i.e. SQRT is correctly rounded) then SQRT($X * X$) = $|X|$
   - for all X if $\beta \le 4$,
   - for $\frac{1}{2} < X < \sqrt{\beta}$ if $\beta > 4$,
   - but not for all X in the range $\sqrt{1/\beta} < X < \frac{1}{2}$ if $\beta > 4$.
4. For quaternary, $\beta = 4$, it is not necessary for SQRT to be correctly rounded in order to have the identity. There is a delicate argument that shows SQRT($X * X$) = $|X|$ for all X only if the result is computed with more than half again as many quaternary digits as you will return, and then rounded.



5. I hope you can appreciate that none of the proofs I showed above will work if all we had was the standard model. If you use the ϵ-model, it's too weak to permit us to draw appropriate conclusions. Instead, by knowing exactly how the arithmetic works you can rigorously prove certain empirical observations not explainable by conventional means.

4 A Preview of IEEE Arithmetic: Gradual Underflow

In the next lecture I will tell you more about IEEE arithmetic. You have already had some idea of how we do error analysis and that will help to justify some of the things in the standard. For instance, we can understand why the standard specifies binary arithmetic at least in 754 because binary is clearly best. There are things that work in binary that don't work in any other radix. An example is the square root of a square (it also works for quaternary but I don't think anybody would use quaternary). So you can understand one of the reasons for binary. The other reason is that when you use the cruder argument for your error analysis the error bound that you got could be sloppy by factors as big as the radix. That's why we choose a small radix instead of a big one in order to avoid this type of floppyness (called "wobbling precision").



The standard takes advantage of the fact that a normalized number in binary always has a leading bit 1. You can imagine there is a 1 just before the fraction digits for almost all the numbers (there are some exceptions). The leading bit of a normalized floating-point number is always non-zero, so why bother to store it? You gain a little bit of memory

space. This means in consequence the IEEE Standard single precision word is noticeably better than other machines (they exist) that use an explicit leading bit. Where does this trick come from? Well, it's actually call Goldberg's Variation, it appeared in Comm. ACM sometime around 1961. He really wrote an article to explain why an IBM 7090, which has 27 significant bits, really wasn't quite so good as 8 significant decimals. He said if they had merely put in one more bit, which they could have done as above, then everything would be fine and they would have 28 bits with the same word size, always better than 8 significant decimals.

Goldberg was also responsible for gradual underflow. The numerical interpretation of a typical floating-point representation is

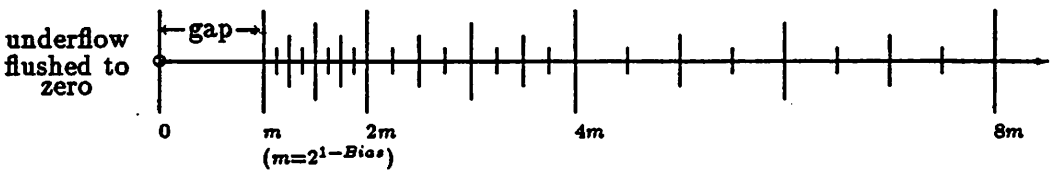
$$X = \pm 2^{exponent-Bias} \cdot (1 + .fraction), \quad \text{if } exponent > 0 \tag{1}$$

You may ask why not use $exponent \geq 0$ in (1)? (On a VAX, "$exponent = 0$" is a signal that the whole result X will be regarded, regardless of fraction, as zero if the sign is positive, "Reserved Operand" if negative. Why IEEE doesn't do that will become clear later.) But when $exponent = 0$, you simply could move the implicit leading 1 to the exponent field as is done below.

$$X = \pm 2^{1-Bias} \cdot (0 + .fraction), \qquad \text{if } exponent = 0$$

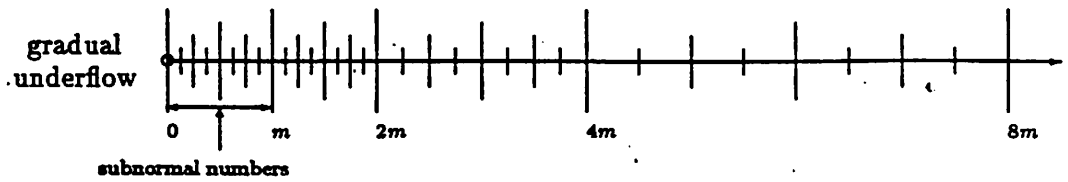
This was invented by Goldberg. It provides "gradual underflow". Actually I implemented it in a 7090 at about the same time. It works.

A picture may help to understand the meaning of gradual underflow. We can consider only positive floating-point numbers. In this whole scheme there is a smallest normalized number $m = 2^{1-Bias}$.

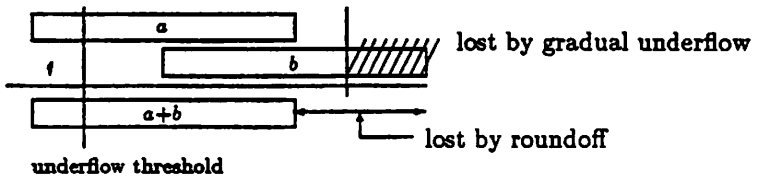


Here each vertical tick stands for a floating-point number. Please observe that each time you cross a power of 2, the density of numbers changes by a factor of 2. In other words, the gap on one side of the power of 2 is twice (or half) as big as the gap on the other. But what is going to happen at m? The figure above shows what happened on a VAX: underflow gets flushed to zero. Unfortunately that means that when an underflow occurs the error is comparable with the gap between 0 and m, which is enormously bigger (relatively and absolutely) than the error in numbers that you try to create somewhere in the next binade $(m, 2m)$. If you create a number in $(m, 2m)$, it will be rounded to one of the floating-point number in that range and the error would be bounded by $m \cdot 2^{-precision}$, but over the region $(0, m)$ the error could go up to m. The ratio of the rounding error bound across m is thus something of the order of $2^{precision}$. So for single precision ($precision = 24$), the gap on the left hand side of m is 2^{24} ($\sim$16 million) times bigger than the other side. Therefore when underflow occurs it makes an error look enormous compared with a rounding error that you would make in a nearby region.

What Goldberg's Variation does is merely fill in this $(0, m)$ gap the same way as on the right of m. The density is now constant from $2m$ to zero.



Now, you see, when you try to create a number in $(0, m)$, you will only make an absolute error which is comparable with the error you make on $(m, 2m)$. That says, at least for some purposes (when absolute error is important), underflow makes an error comparable with rounding error. When you expect to add whatever it is that may have underflowed to something else probably bigger, you will lose no more to gradual underflow then must be lost to roundoff anyway.



Consider adding a and b where a is slightly bigger than the underflow threshold and b is slightly less. If underflow got flushed to zero, then b is lost entirely. But if underflow is gradual, then the part lost to underflow is the shadowed portion, which is unimportant because it will be lost in roundoff anyway. This kind of computation occurs so often that it is worth going to the bother of doing this, especially on machines with a relatively narrow exponent range. For example, a PDP-11, or a VAX in F and D format has exponent range out to only $10^{\pm 38}$; the same is true for IEEE single. Or even an IBM machine, because its range gets around $10^{\pm 76}$. That is a very narrow exponent range. Why I say that will become clear later when you look at the relationship between the number of significant figures you might carry and the exponent range, and you will see that you start to feel very uneasy. On the other hand, it is true that in VAX G or H format, or IEEE double you might be less inclined to worry about underflow because the exponent range is $10^{\pm 308}$, and it is not likely to bother most people. It will bother only programmers who are very conscientious. Remember I said: if $\frac{1}{2} \leq p/q \leq 2$ then $p - q$ is exact. It seems like a valuable property. Unfortunately on machines that flush underflow to zero, that property will be violated. You see if $p, q \in (m, 2m)$, then $p - q < m$ will be flushed to zero. If underflow is gradual, however, $p - q$ will be represented *exactly* by a subnormal number. So there is an example of a theorem, which is preserved by gradual underflow, and lost if you flush to zero. Let's take another example.

```
IF (X.NE.Y) ...  { Z/(X-Y) will be O.K. }
```

That's a predicate you might see in Fortran. Somewhere else in the program you might see

```
IF (X-Y.NE.0) ...  { Z/(X-Y) will be O.K. }
```

However, unfortunately, one of these predicates can malfunction if underflow is not gradual. On a VAX or on an IBM 370, if you use `IF (X.NE.Y) ...`, you may discover X and Y are allegedly unequal, and then you may go to do that divide but that's too bad, because $X - Y$ may underflow to zero. That can't happen on a machine with gradual underflow. So there are some valuable uses of gradual underflow. You may find some extensive discussion on this matter in a paper by Jim Demmel in `SIAM J SCI STAT. COMP Vol 5 #4 Dec 1983 pp887-919`. In Demmel's paper there are significant examples where reliable software is much easier to write with gradual underflow then without.

What we have done here is *not* to extend the exponent range. The extension of range is inconsequential. What's happening is that certain kind of theorems are true with gradual underflow and not with flush to zero. Furthermore, gradual underflow is not something that was invented for the IEEE Standard. It was implemented for the 7090 and 7094 in about 1962. It worked and made a lot of matrix codes ran better. In 1966 it was implemented at Stanford on a Burroughs B5500, a machine that was replaced by an IBM 360/67 in 1967, so it didn't have a very long life span. It was implemented on an IBM 360 at the University of Waterloo in 1966 also, but it was never adopted by IBM. It is feasible on IBM 370 hardware, but to make it work, you have to change the underflow trap handler to make underflow gradual. All codes will continue to work except a test for underflow that tests whether a number is zero. When gradual underflow occurs, bits can be lost without losing all of them. So if you have a code that tests for underflow by looking for zero, it could be fooled. The right thing to do is to test the underflow flag. On an IBM 370 if you enable the underflow trap, which you must do if you want underflow to be gradual, you would get an underflow flag. But people in Fortran didn't want any flags back in 1967, none at all.

How often at best can SQRT(X*X) = AS...

Counting Argument

p sig. digits of radix β.

Interval $\beta^{p-1} \le x \le \beta^p - 1$ has $\beta^p - \beta^{p-1}$ arguments x. } Typical β-ade

Consider first $1 < \check{x} < x < \hat{x} < \sqrt{\beta}$

$$\#(x's) = \frac{\hat{x} - \check{x}}{\beta - 1} \cdot (\beta^p - \beta^{p-1}) = (\hat{x} - \check{x}) \cdot \beta^{p-1}$$

$y = x^2$: $\quad 1 < \check{x}^2 < y = x^2 < \hat{x}^2 < \beta$

$$\#(y's) = (\hat{x}^2 - \check{x}^2) \cdot \beta^{p-1} = (\hat{x} + \check{x}) \cdot \#(x's) > 2 \cdot \#(x's)$$

$\therefore$ x could be recovered from y : $x = \sqrt{y}$.

Consider second $\sqrt{1/\beta} < \check{x} < x < \hat{x} < 1$.

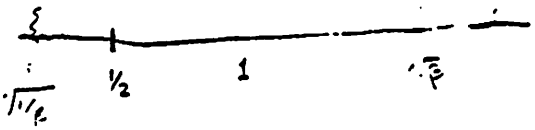
$$\#(x's) = \frac{\hat{x} - \check{x}}{1 - 1/\beta} \cdot (\beta^p - \beta^{p-1}) = (\hat{x} - \check{x}) \cdot \beta^p$$

$y = x^2$: $\quad 1/\beta < \check{x}^2 < y = x^2 < \hat{x}^2 < 1$

$$\#(y's) = (\hat{x}^2 - \check{x}^2) \cdot \beta^p = (\hat{x} + \check{x}) \cdot \#(x's)$$

If $\hat{x} + \check{x} < 1$ then at most $(\hat{x} + \check{x})$ of the x's can be recovered from y's.

$\therefore$ if $\sqrt{1/\beta} < \check{x} < x < \hat{x} < 1/2$



then SQRT(Y*X) = X at most $(\check{x} + \hat{x})$ of the time.

Kahan 24 May

Exercise :

HOW OFTEN AT BEST CAN WE EXPECT SQRT(X)**2 = X ?

Assume p sig. digits of radix β,

and consider $1 < \check{x} < x < \hat{x} < \beta$.

Show that the fraction of such x's for which SQRT(X)**2 = X cannot exceed

$$\frac{1}{\sqrt{\check{x}} + \sqrt{\hat{x}}} < 1/2 .$$

Back to WHETHER SQRT(X*X) = ABS(X)
in arithmetic with p sig. digits of radix β.

We need consider only the ranges
$\max(\sqrt{1/\beta}, 1/2) < x < 1$ and $1 < x < \sqrt{\beta}$,
since a counting argument says that we
cannot expect SQRT(X*X) = X elsewhere in $\sqrt{1/\beta} < x < \sqrt{\beta}$

Notation: [expression] = correctly rounded value of (expression)
= (expression) ± ½ ulp(expression)

If $1/\beta < x < 1$ then $1\ ulp(x) = \beta^{-p}$
$1 < x < \beta$ " " β^{1-p}

Assume Multiplication is correctly rounded.

Define $y = [x \cdot x] = x^2 + \eta \cdot ulp(x)$ $|\eta| \le \frac{1}{2}$
$z = SQRT(y) = \sqrt{y} + \zeta \cdot ulp(\sqrt{y})$ $|\zeta| \le \sigma$.
σ = error bound for SQRT program;
Assume $1 \ge \sigma \ge \frac{1}{2}$; $\sigma = \frac{1}{2}$ if SQRT is correctly rounded.

$$y = [x^2] = x^2 + \eta \cdot \mathrm{ulp}(x^2)$$
$$z = \mathrm{SQRT}(y) = \sqrt{y} + \varepsilon \cdot \mathrm{ulp}(\sqrt{y})$$

$1/\sqrt{\beta} < x \le 1 - \beta^{-p}$ or $1 + \beta^{1-p} \le x < \sqrt{\beta}$

$1/\beta \le y = [x^2] \le 1 - 2\cdot\beta^{-p}$ or $1 + 2\beta^{1-p} \le y = [x^2] \le \beta$

$1/\beta < z = \mathrm{SQRT}(y) \le 1$ or $1 \le z = \mathrm{SQRT}(y) < \beta$

$\mathrm{ulp}(z) = \mathrm{ulp}(y) = \mathrm{ulp}(z) = \beta^{-p}$ or $\ldots = \beta^{1-p}$

$$y = x^2 + \eta \cdot \mathrm{ulp} \qquad z = \sqrt{y} + \varepsilon \cdot \mathrm{ulp}$$

$\therefore$ $\dfrac{|z - x|}{\mathrm{ulp}}$ = a small integer

"$z = x$" $\Longleftrightarrow$ "$\dfrac{|z-x|}{\mathrm{ulp}} < 1$".

How accurate must SQRT be
(i.e. how small $\alpha \ge 1.5$!)
to imply $|z - x|/\mathrm{ulp} < 1$?

$$y = x^2 + \eta \cdot ulp \qquad z = \sqrt{y} + 1.5 \cdot ulp$$

$$\therefore \quad \frac{z - x}{ulp} = \frac{(\sqrt{y} + 1.5 \cdot ulp) - \sqrt{y - \eta \cdot ulp}}{ulp}$$

$$= 1.5 + \frac{\eta}{\sqrt{y} + x} \qquad \ldots \quad x = \sqrt{y - \eta \cdot ulp}$$

$$\lesssim 1.5 + \frac{\eta}{2\sqrt{y}}$$

$$\therefore \quad \frac{|z - x'|}{ulp} \lesssim 2 + \frac{1/2}{2\sqrt{y}}$$

If $0 \le t := 1 - \frac{1/4}{\sqrt{y}}$ then surely $\frac{|z-x|}{ulp} < 1$

i.e. SQRT(X·X) = X

Critical t is $\frac{1}{2}$, since we can't keep $\sigma < \frac{1}{2}$.

Plot $t := 1 - \frac{1/2}{\sqrt{y}}$ vs y.

CONCLUSIONS:

If $\sigma < 3/4$ (i.e. SQRT(Y) = $\sqrt{Y} \pm \frac{3}{4}$ ulp($\sqrt{\ }$))
then SQRT(X*X) = X throughout $1 \le X \le \sqrt{\beta}$.

If $\sigma < 1 - \frac{1}{\sqrt{8}} = 0.6464466$ and $\beta = 2$
then SQRT(X*X) = ABS(X) for ALL X
(a benefit of BINARY).

If $\sigma = 1/2$ (i.e. SQRT is correctly rounded.
then SQRT(X*X) = ABS(X) for all X if $\beta \le 4$

and for $1/2 < X < \sqrt{\beta}$ if $\beta > 4$.

but not for $\sqrt{1/\beta} < X < 1/2$ if $\beta > 4$.

Delicate argument shows SQRT(X*X) = ABS(X) for all X
and $\beta = 4$ only if SQRT(Y) = [$\sqrt{Y}$] ... YYYYY
rounded from 3/2 ulp's.