Computer System Support for Scientific and Engineering Computation

Lecture 26 - July 28, 1988 (notes revised June 14, 1990)

1 Signalling NaNs

The IEEE standard contains signaling NaNs, which can be used to extend the set of numbers that can be represented. For example, the exact value of π could be represented by a signalling NaN. Then each time π appeared as the operand to an arithmetic operation, a trap handler would be called to specially handle the situation.

The IEEE standard leaves open to implementors whether copying a signaling NaN or changing its sign should raise a signal. When using signaling NaNs to represent special kinds of numbers, it seems most appropriate to not raise a signal. In the example above, you wouldn't expect copying π to raise a signal. Also, these new numbers will typically have signs that obey the usual operations, so it would cause unnecessary overhead to raise a signal when only the sign bit was modified. If signals are not raised for copying and changing the sign bit, then `x = abs(y)` will go through without raising a signal.

2 Test Suites

There are a number of test suites for checking the correctness of floating point arithmetic. One is distributed by NAG and is based on an earlier program of Norm Schryer. This test uses operands of very special form, such as $100\cdots00100\cdots0$ or $11\cdots1100\cdots00$, and then tests to see if the sum, product, difference and quotient of these numbers is correctly computed. This test is good at detecting problems with propagation of carries. Also, the test can take as parameters values for the base and precision, and test whether those are indeed the parameters being used. Paranoia, written by W. Kahan, will actually attempt to deduce the parameters of a floating point system. In addition to computing the base and precision, it will also try to deduce whether operations are rounded or truncated. However, this test is not designed to detect machines that round, but occasionally round incorrectly. The handouts contain two test programs for this purpose. One of them computes operand values that when multiplied together, will result in a value almost halfway between two representable numbers. Thus if rounding is only slightly wrong, this test has a good chance of detecting it. The other program computes operand values for division.

Benchmarks are another kind of test program. Unfortunately, most test only for speed, not for accuracy. Benchmarks usually consist of a fixed program to be run and timed on

various machines. A better way to do benchmarking might to be consider a particular problem, like inverting a matrix, and time how long it takes a computer (by whatever algorithm is most appropriate to its architecture) to compute the result to within a specified accuracy. Sorting and transposing a matrix are other candidates for this benchmarking approach.

Another problem with benchmarks is that they are not diagnostic. That is, if they run more slowly than they should, they don't offer any suggestions as to what is wrong.

3 Computing Transcendental Functions

One method for computing transcendental functions is the CORDIC algorithm, presented in earlier lectures. This is really only appropriate for hardware implementions. Software algorithms are almost always done by approximating the transcendental function with polynomials or rational functions. Polynomial approximations will always exist, because of the Weierstrass approximation theorem.

Theorem 1 (Weierstrass) *Given a finite interval [a,b], a function f continuous on that interval, and a tolerance ϵ, there exists a polynomial p such that $|f(x) - p(x)| < \epsilon$ for all x satisfying $a \leq x \leq b$.*

It is easy to see how the theorem can be extended to require the polynomial to take on a predefined value at a particular point (for example, when approximating the logarithm by a polynomial, you would like that polynomial to be exactly zero when $x = 1$). To specify the polynomial to be $f(c)$ at $x = c$, first find a polynomial p that approximates f to within $\epsilon/2$. Then replace p with $p - p(c) + f(c)$. This new polynomial approximates f to within ϵ, and has the value $f(c)$ when $x = c$.

Unfortunately, Weierstrass' theorem doesn't tell us anything about the degree of the polynomial. When f has a vertical tangent (such as $\sqrt{x}$ does at $x = 0$), the degree of the polynomial can get very high. It also doesn't say anything about how to construct the approximating polynomial.

The classical schemes for constructing explicit approximations use *interpolation*, that is, finding a polyomial p that agrees with f exactly at n fixed points. Since a polynomial p of degree $n - 1$ has n coefficients, specifying the value of p at n points will result in n equations in the n unknown coefficients. However, this does not always lead to a reasonable approximation. Runge discovered that for $f(x) = 1/(1 + 25x^2)$ on the interval [-1, 1] and equally spaced points, the interpolating polynomials not only don't approximate f, but that $p_n(x) \to \infty$ for x near the endpoints ± 1 as $n \to \infty$. However, there are unequally spaced points for which the interpolation polynomials will converge to f.

Polynomials can only approximate functions over a fixed finite interval, but rational functions can approximate over an infinite interval. For example, to approximate a function $f(x)$ for $0 < x < \infty$, consider the function $g(x) = f(1/x)$ which is defined for $0 < x < 1$ and approximate it by a polynomial $p(x)$. Then the rational function $q(x) = p(1/x)$ will approximate the original function f.

3.1 Remes Algorithm

Although polynomial approximation can be done using interpolation if the interpolation points are carefully chosen, there is a better method. Fix an interval [a,b] and a function

f to be approximated on that interval. Then among all polynomials of degree n, what will the best approximation look like? The answer is given by

Theorem 2 (Chebyshev) *Given a continuous function f defined on the interval $[a,b]$, then p is the the polynomial with the smallest value of* $\max_{a\le x\le b}|f(x)-p(x)|$ *if and only if the error $E(x)=f(x)-p(x)$ satisfies $E(x_i)=(-1)^i\epsilon$ for $n+2$ points x_i, where $a\le x_0<x_1<\cdots x_{n+1}\le b$ and $\epsilon=\max_{a\le x\le b}|f(x)-p(x)|$.*

In other words, the best approximation by a polynomial of degree n has the property that the error oscillates at least $n+2$ times. This theorem suggests that the way to find the best approximation is to attempt to constrain the error to have $n+2$ extreme points. And such an algorithm exists.

Algorithm 1 (Remes) *Let f be a function analytic on an interval $[a,b]$, and let $p(x,a_0,\ldots,a_n)$ be an analytic approximation to f parametrized by $n+1$ variables. Let $E=f-p$, and guess an initial set $\{x_i\}$ of $n+2$ extrema satisfying $a\le x_0<x_1<\cdots x_{n+1}\le b$. Then solve the following system of equations for the $n+2$ unknowns $\{a_j\}$ and ϵ*

$$(-1)^{i+1}\epsilon+E(x_i,a_0,\ldots,a_n)=0,\ 0\le i\le n+1 \tag{1}$$

Next, using these values of $\{a_j\}$, solve the equations

$$(x_i-a)(x_i-b)\frac{\partial}{\partial x}E(x_i,a_0,\ldots,a_n)=0,\ 0\le i\le n+1 \tag{2}$$

to get new values of $\{x_i\}$. Repeat solving equations (1) and (2) until the $n+2$ unknowns $\{x_i\}$, $\{a_j\}$ and ϵ converge.

The second set of equations (2) are uncoupled, and can be solved one at a time using the ordinary one dimensional Newton's method. However, (1) is a coupled set of equations, and must be solved using the n-dimensional Newton's method, which involve computing the partial derivatives $\frac{\partial}{\partial a_j}$.

The usual form of the Remes algorithm restricts the approximation p to be a polynomial or rational function with all of its coefficients as the unknown a_j. This more general form is useful for rational functions with some of its coefficients fixed (see below), or even for approximations involving non-rational functions such as square roots. Using square roots might be appropriate for machines that have square root implemented in hardware at almost the same speed as divide. However, this more general form doesn't always converge, unlike the special case where p is a polynomial of degree n, with all the coefficients of p used as the parameters a_j. In that special case, the algorithm is guaranteed to converge, no matter what the initial guess for the x_i are.

It turns out (see the handout *Superlinear Convergence of a Remes Algorithm*) that under a few mild conditions on the f and p, this Remes algorithm will converge if the inital guesses for x_i and a_j are close enough to the true values. And the convergence rate is *superlinear*, which means that the number of correct digits increases in a geometric progression. That is, the number of correct digits progresses like n, rn, r^2n, ..., for some $r>1$. Since the algorithm might not converge if the initial guesses are bad (or the conditions of the convergence theorem turn out not to be satisfied), it is comforting to know that when it does converge, it converges rapidly. As a practical matter, this means that if running the algorithm doesn't converge after a few rounds, it probably never will converge.

When solving equations (2), various problems can come up. For example, there may be more than $n+2$ solutions. There is an improved algorithm that deals with this situation by introducing the zeros of E as another set of unknowns. After all, since $E(x_i)E(x_{i+1}) = -e^2 < 0$, E must have a zero somewhere between x_i and x_{i+1}. And similarly, between two zeros of E there must be a point where the derivative is zero (Rolle's theorem from elementary calculus courses).

Algorithm 2 (Improved Remes Algorithm) *Let f and p be as in algorithm 1. First guess a set $\{y_k\}$ of $n+1$ numbers, and use these numbers to solve*

$$E(y_k, a_0, \ldots, a_n) = 0,\ 0 \le k \le n \tag{3}$$

for $\{a_j\}$. Next use these values of a_j to solve

$$(x_i - a)(x_i - b)\frac{\partial}{\partial x}E(x_i, a_0, \ldots, a_n) = 0,\ 0 \le i \le n+1 \tag{4}$$

for $\{x_i\}$, with the restriction that $x_i < y_i < x_{i+1}$. Then use these values of x_i to solve

$$(-1)^{i+1}\epsilon + E(x_i, a_0, \ldots, a_n) = 0,\ 0 \le i \le n+1 \tag{5}$$

to get new values of $\{a_j\}$. Finally use these values of a_j to solve

$$E(y_k, a_0, \ldots, a_n) = 0,\ 0 \le k \le n \tag{6}$$

and get new values of $\{y_k\}$. Go back to equation (3) and repeat.

Since each y_k is a zero of E, the function E will cross the x-axis in either a downward or upward direction. When solving equation (6), you should make sure that the crossing directions alternate between successive values of y_k. It is also important to graph the function E to make sure that the extreme points x_i represent the maximum value of $|E|$, that is, to check that there are no spikes in E that shoot past the extrema. If a spike occurs during an iteration, an extrema adjacent to the spike should be moved to where the spike occurs.

3.2 Other Details

If a transcendental function has symmetries, you normally want that symmetry to be reflected in the function's rational approximation. For example, since $\sin(-x) = -\sin x$, you would like an approximation to sin to be an odd rational function.

For ln, the situation is slightly more complicated, since in this case the identity is $\ln 1/x = -\ln x$. To translate that into a condition on the rational approximation, note that if $x = \frac{s+1}{s-1}$, then $1/x = -\frac{(-s)+1}{(-s)-1}$. So let $R(x)$ be a rational function satisfying $R(1/x) = -R(x)$. Then let $g(s) = R(\frac{s+1}{s-1}) = R(x)$. Then $g(-s) = R(\frac{(-s)+1}{(-s)-1} = R(1/x) = -R(x) = -g(s)$. So g is an odd function. To summarize, a rational function with the property $R(x) = -R(1/x)$ (which we would want an approximation to ln to have) must be of the form $R(x) = g(\frac{x+1}{x-1})$ for some odd rational function g.

When x is near 1, $\ln(x) \approx x - 1$, so an approximation to ln will look something like $(x-1) + C(x-1)^2R(x)$. On machines with a guard digit, $x-1$ will be exact, so all the rounding error will come from the second term $C(x-1)^2R(x)$. If this term is much smaller than $x-1$, then the final result will have practically no rounding error at all, because

the rounding error in $C(x-1)^2R(x)$ is in the lower order bits which get shifted off when it is added to $x-1$. However, if $C(x-1)^2R(x)$ is about the same size as $x-1$, then its rounding error becomes important. And a big contributor to that rounding error is C, which is a floating point approximation to some transcendental number obtained from the Remes algorithm (it was one of the a_j). A way to avoid this is to change C to an exactly representable floating point number, and rerun the Remes algorithm with one less parameter a_j. The result will be a new approximation p with an error ϵ which is a tiny fraction of an ulp larger than before. However, C will now be exact, so the net effect is that $(x-1)+C(x-1)^2R(x)$ is a much closer approximation to ln than before.

3.3 Piecewise Approximations

As the number of parameters in p increases (which translates into increasing degree if p is a rational function), the error ϵ will decrease. If $w=b-a$ is the width of the approximating interval, then $\epsilon \approx Cw^j$, where j is the number of parameters. This means that by shortening the interval, the number of parameters can be decreased without decreasing the accuracy. And this translates into an approximation that is faster to compute. So one method of computing transcendentals is to chop the range into subintervals, and use a separate approximation in each subinterval. Athough this requires more code for the logic that decides what interval to use, and more space for tables of coefficients, the time to compute the approximations is decreased. So this method is faster, unless the time to access the extra tables from memory becomes significant.

If the approximations are of the form

$$f_k+(x-x_k)(g_k+(x-x_k)h_k), \tag{7}$$

and the interval is so small that $(x-x_k)(g_k+(x-x_k)h_k) \ll f_k$, then the multiplication of $(x-x_k)$ and $(g_k+(x-x_k)h_k)$ need only be done to half precision, which may be substantially faster than a full multiply. This trick is due to Farmwald (1982).

One problem with (7) is that f_k, which represents the value of the function at x_k won't be exact, since in general $f(x_k)$ is a transcendental number. So one useful trick is to slighly adjust the intervals so that $f(x_k)$ is very close to an exactly representable floating point number. This trick is due to J.C.P. Miller around 1958, and also independently Gal. Approximations constructed using this technique can give correclty rounded values over 90% of the time. One fine point: it isn't always possible to find an x_k which closely approximates a floating point number, in which case an extra addition must be performed to compensate for the error. Also, as we mentioned earlier, this method can result in large tables, although sometimes the tables can be collapsed due to special properties of the function being approximated. For example, the values of e^x in two different intervals are just multiples of one another.

4 Models

Modern computer science books emphasize the importance of proving programs correct. Even though it is currently impractical to carry out a proof for large programs, taking this point of view is sometimes helpful. For example, it can help guide the design of programming languages. Proving things about floating point is complicated even when you assume IEEE arithmetic, as some of the proofs in these lectures have demonstrated. When IEEE is not

assumed, proofs become greatly complicated by the large variety of floating point hardware. Even among machines of the same manufacturer floating point can differ (as in the CDC 6600 and 7600). Despite these obstacles, several people have attempted to construct general theories of floating point. However, these theories only consider the problem of rounding error, and ignore exceptions.

One approach is to use axioms to characterize floating point arithmetic, motivated perhaps by the fact that all the properties of real numbers can be deduced from a few axioms. A. van Wijngaarden (best known for his work on Algol 68) introduced 32 axioms that he claimed characterized the behavior of floating point arithmetic on all hardware, but his axioms did not cover the CDC 6600, which was the fastest computer at that time. Stan Brown introduced 20 axioms that did cover the 6600, however this introduced so much complexity that he dropped 6600 compatibility from his axioms when he finally published them in *Transactions on Mathematical Software.*

Brown considers a subset of all floating point numbers that he calls *model numbers*, and characterizes the model numbers by a radix, precision, and exponent range. His axioms constrain the behavior of the model numbers in the following way. Let a and b be model numbers, and let $c = a \otimes b$ be the exact the result of operating on them. Then c lies in an interval bounded by model numbers, say $[c_1, c_2]$. Brown's model requires that the result of applying the floating point operation $\otimes$ must result in a floating point number in the interval $[c_1, c_2]$.[1] The behavior of non-model numbers is defined in terms of the containing interval with model number endpoints. For a machine like the Cray, where the result of a multiply can be quite far (in terms of ulps) from the true product, Brown's model will apply if the model numbers are taken to be all floating point numbers with their last bits zero. That is, the precision of the model numbers is less than the actual precision of the hardware.

The problem with Brown's model is that it tries to apply to all existing hardware, and so suffers from the fact that it doesn't consider any of the newer developments in floating hardware, such as guard digits, NaN's etc. In fact, it is actually is worse than any existing machine, because there are codes that work on all existing machines, but can't be proven correct in Brown's model.

[1] Actually, he calls operators with this property *strongly supported*, and for operations that are merely supported, lets them stray into an interval expanded on each end by one model number.