Computer System Support for Scientific and Engineering Computation

Lecture 12 - June 9, 1988 (notes revised July 11, 1988)

1 Critique of Kulisch-Miranker Theory

The heart of the Kulisch-Miranker theory is a nice set of axioms, which they call the axioms for a *semimorphism*. A semimorphism consists of a rounding function $\Box$ which maps a set R (think of the real numbers) to a subset F (think of the floating point numbers representable in a computer) satisfying

$$\begin{array}{rll} \Box a = a & \text{for all } a \in F & \text{(rounding)} \\ a \le b \Rightarrow \Box a \le \Box b & \text{for all } a,b \in R & \text{(monotonicity)} \\ \Box(-a) = -\Box a & \text{for all } a \in R & \text{(antisymmetry).} \end{array}$$

Then they define operations in F based on those in R using the formula

$$a \boxed{\circ} b = \Box(a \circ b) \quad a,b \in F, \tag{1}$$

which in words says that floating point operations are computed exactly and then rounded, so they have less than 1 ulp of rounding error. The operations $\circ$ that they consider are addition, subtraction, multiplication, division (for scalar sets R) and inner product. Since inner product must satisfy (1), it must be computed in a (conceptually) very long register, the super-accumulator. How long does the super-accumulator have to be? It must be able to hold a sum whose summands can be a big as the square of the largest number and as small as the square of the smallest nonzero number in F. If floating point numbers have p digits in the significand, and exponents ranging between e_1 and e_2, then the largest square is about β^{2e_2} and the smallest square is about β^{2e_1} and so the super-accumulator must have at least $2e_2 + 2p + 2|e_1|$ digits, plus a little more to allow for overflow on addition. The product will be computed exactly in the super-accumulator and then rounded with less than 1 ulp error. The set R can be more general than just the real numbers. R could represent the complex numbers, or vectors, or matrices or intervals. The reason why the operator $\circ$ is only allowed for scalar sets R is that computing the inverse of a matrix to within 1 ulp of the true answer would be an extremely demanding requirement. In the pure theory, you can't reference the super-accumulator directly. Instead your program can do things like declare `A`, `B` and `C` to be matrices, set `A = B * C` and have the product computed to within 1 ulp via the super-accumulator.

Just because the semimorphism axioms are elegant, that doesn't imply that they are an appropriate basis for floating point computation. The first problem with the semimorphism axioms is that they are too weak. In particular, the concept of semimorphism is not transitive. If we want to compute the matrix product ABC, the semimorphism property guarantees that AB is accurate to 1 ulp, but is too weak to say anything about the final product $(AB)C$. On the other hand, the semimorphism axioms are too strong. They allow floating point implementations that do not have good properties. For example, imagine representing floating point numbers in a logarithmic fashion, where

$$\boxed{\pm}\boxed{\qquad \pm i \qquad}$$

would represent the number $\pm r^{\pm i}$, and $r = 1.0000\cdots 00xxx$ is a number very close to one. Such a representation is used in the FOCUS system. Multiplication and division of numbers in this scheme is very easy. Simply add or subtract the integers i. Addition is more complicated, and is typically implemented using table lookup, which is an idea that goes back to Gauss. Although addition isn't exact, the distributive law $a(b+c) = ab + ac$ does hold exactly. Another peculiarity of this system is that it can represent 2 exactly, or it can represent 3 exactly, but not both. The reason is simple. If there were integers i, j satisfying $r^i = 2$ and $r^j = 3$, then $r^{ij} = 2^j = 3^i$, which is impossible since 2^j is even and 3^i is odd! Thus this system can't simultaneously represent both 2 and 3.

However, the most serious problem with logarithmic representation, is that some of our error analyses no longer hold. To give a concrete example, in the logarithmic representation, it is no longer true that $\frac{1}{2} \leq x/y \leq 2$ implies $x - y$ exact, so the formula that we gave for accurately computing Heron's formula can no longer be depended upon. To summarize, the semimorphism axioms are so strong that they only apply to a single operation, and so we can't get good bounds on the error of a simple expression involving two operations such as ABC, and they are too weak to rule out a logarithmic representation, so we can't figure out from the axioms whether a simple program for Heron's formula will work.

Let's consider the problem of multiplying three matrices in more detail. We know that in general, it won't be possible to compute the product to within 1 ulp, because the super-accumulator is only big enough to hold the exact product of two floats, and to compute a product of three matrices to within 1 ulp would require an accumulator big enough to hold the product of three floating point numbers. However, we can use the Kulisch-Miranker theory to multiply three matrices by formulating it as a bigger matrix problem.

$$\begin{pmatrix} 1 & 0 & 0 \\ B & -1 & 0 \\ 0 & A & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} C \\ 0 \\ 0 \end{pmatrix}$$

The value of z should be the matrix product ABC. There are two problems with this method. The first is that it is nonobvious. It doesn't seem reasonable to ask users who want to compute the product of three matrices to have to recast it as a linear system. The second problem is that if iterative refinements are needed, it will be slow. We noted earlier (lecture 10) that each interative refinement requires recalculation of vectors that were computed in the previous step. Perhaps simply doing a straightforward matrix multiply using a multiprecision floating point package would be faster.

As a final example of where the Kulisch-Miranker method falls down, consider evaluating a polynomial $\sum_j a_j x^{n-j}$. Kulisch and Miranker use the polynomial $\sum_{n=0}^{N} (-x)^n/n! \approx e^{-x}$ with $x = 20$ to show the problems with cancellation. The first 20 summands grow starting with 1 until they reach ±4309600, then they drop off. Since e^{-20} is a very small number, we know that the large summands must all cancel. Thus unless the computation is done accumulating sums in a register large enough to hold the first term 1 and the sum of all the terms up to 4309600 exactly, roundoff error will result in a very inexact result. The larger N is, the larger the accumulator must be.

The straightforward approach to evaluating a polynomial would be to use Horner's rule

```
P = 0;
for j = 1 to N {
    P = P*x + A[j];
}
```

possibly using a multiprecision package if needed. In the Kulisch-Miranker paradigm, since the super-accumulator is only big enough to hold any square, it doesn't work to simply add the terms of the polynomial in the super-accumulator. What ACRITH actually does to evaluate $\sum_{n=0}^{N}(-x)^n/n!$ is to convert it to a matrix equation, estimate the solution to that matrix equation, and then use iterative refinement to close in on the final answer. At each step the computation uses either the very long super-accumulator, or else arithmetic in a fixed working precision. The problem with insisting that multiprecision computation can only be done in the super-accumulator can be seen by imagining how the initial estimate of the polynomial $\sum_{n=0}^{N}(-x)^n/n!$ might be done. The formula

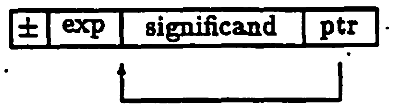
$$\sum_{n=0}^{N}(-x)^n/n! = 1 - x(1 - \frac{x}{2}(1 - \frac{x}{3}(\cdots - \frac{x}{N}))\cdots)),$$

appears to be an excellent way to estimate the sum, because it doesn't suffer from underflow or overflow. To see this, assume we have taken N large enough so that the sum is less than 1. When $0 < x < 1$, also $0 < 1 - x < 1$, so the quantity to the right of each minus sign is between 0 and 1, and you wouldn't expect to get underflow or overflow. However, each time you perform an arithmetic operation, there is a roundoff error. And the first $\lfloor x \rfloor - 1$ numbers in the sequence $x/2$, $x/3$, $x/4$, ...are bigger than 1, so multiplying by them will magnify the roundoff error. If x and N are large enough, computing in single precision can have a roundoff error that grows enough to cause an overflow in this initial guess. In other words, since ACRITH limits itself to a fixed precision, it can get spurious overflows in its inital guess, failing before it can even begin iterative refinement. Here is another example where the Kulisch-Miranker methodology rejects a straightforward method in favor of a complex one. And in this case, the complex method causes an overflow that wouldn't appear using the straightforward approach.

2 Overflow

In the *Eighth IEEE Symposium on Computer Arithmetic*, several papers* discuss methods of avoiding overflow and underflow. The effect of these methods is similar to using a few bits for a pointer which indicates where the boundary between the exponent and significand fields is.

*See the articles by Hamada (p 153), Olver (p. 144) and Demmel (p. 148).



When the exponent gets too big to fit in the exp field, the pointer is changed to make the exp field larger (at the expense of the significand field). These methods have been analyzed by Demmel in those same proceedings. His conclusion can be summarized as follows. When the exp field is made larger at the expense of the significand field, roundoff errors become larger. Therefore, these methods trade off troubles with over/underflow for troubles estimating the effect of roundoff with unpredictably varying precision. Since every calculation has trouble with roundoff error, but only a few calculations involve overflow, the tradeoff is not a good one.

A different approach to the over/underflow problem is to observe that it frequently occurs in calculations of the form

$$\prod \frac{a_i + b_i}{c_i + d_i}.$$

A straightforward way of avoiding over/underflow in this case is to use the functions `frexp` and `ldexp` (or the IEEE alternatives scalb, logb). Then the exponent range is extended to be the same as the range for integers. However, this requires extra `frexp` and `ldexp` operations regardless of whether the data would generate over/underflows. There is a method that does extra work only if over/underflow actually occurs in that product/qutotient; it is *over/underflow counting* which was introduced by Kahan[†]. This can be implemented on a machine that allows a user trap handler to be installed for over/underflow, and that makes available to the trap handler the result of the operation, but with the exponent field wrapped round. This is recommended but not required by the IEEE standard. The way over/underflow counting works is simple. There is a global counter that is initialized to 0 before the calculation starts. Each time there is an overflow, the counter is incremented by 1. Each time there is an underflow, the counter is decremented by 1. When the calculation is complete, if the counter is 0 the final result is a representable floating point number. If the counter is positive, the final result must overflow, otherwise underflow. The advantage of this scheme is that no cost is incurred unless an over/underflow actually occurs. Over/underflow counting has been implemented on the Vax$^{\mathrm{TM}}$ by David Barnett. Sample algorithms were slowed down by about a factor of 2[‡]. Most of this extra time was in operating system overhead to handle the trap.

3 CORDIC Algorithms

The next two sections are out of logical sequence in order to give some background for the guest lecture by Jim Valerio (lecture 13). The goal of CORDIC[§] algorithms is to allow transcendental functions to be implemented on a chip. Both the Motorola 68881 and the Intel 8087 use CORDIC methods. The best reference for CORDIC algorithms is *A Unified Algorithm for Elementary Functions* by Steve Walther, which is reprinted in Swartzlander's

[†]See the section on counting mode in the book *Floating-Point Computation* by Sterbenz, or the original SHARE report *7094 II System Support for Numerical Analysis* by Kahan.

[‡]That is, the program was first run without over/underflow trapping turned on (getting the wrong answer) and then with it turned on.

[§]CORDIC is an acronym for Coordinate Rotation Digital Computer.

book. The original CORDIC paper by Volder is also in that collection. In this section, we will explain pseudo-multiplication and pseudo-division, which are the essential ideas behind the CORDIC method.

One common way of implementing transcendental functions is to first use argument reduction to put the argument into a small range, and then use a polynomial or rational approximation to evaluate the function in that range. For example, to evaluate a logarithm in base 2, the formula $\log_2(2^r y) = r + \log_2(y)$ enables us to reduce the argument to the range $\sqrt{\frac{1}{2}} \le y < \sqrt{2}$. Then we can use the Taylor series approximation

$$\log_2(y) = \frac{\log_e(y)}{\ln(2)} = \frac{y-1}{\ln(2)}\left(1 - \frac{1}{2}(y-1) + \frac{1}{3}(y-1)^2 - \cdots\right) \tag{2}$$

to evaluate $\log_2(y)$. The reason for doing the argument reduction first is that the closer $y - 1$ is to zero, the faster the series converges.

The idea of CORDIC algorithms is to perform not just one range reduction, but many range reductions. In the case of logarithms, we want to represent y as

$$y = 2^{q_0}(1+\frac{1}{2})^{-q_1}(1+\frac{1}{4})^{-q_2}\cdots(1+\frac{1}{2^l})^{-q_l}(1-\eta)$$

where η is a correction factor smaller than 2^{-l} and for $i > 0$, q_i is either 0 or 1. Then $\log_2(y) = q_0 - q_1\log_2(1+\frac{1}{2}) - q_2\log_2(1+\frac{1}{4}) - \cdots + \log_2(1-\eta)$ is just a sum of terms of the form $\log_2(1+2^{-j})$ which can be read out of a small table, plus $\log_2(1-\eta)$. Since η is extremely close to 0, $\log_2(1-\eta)$ is approximated by very few terms of (2).

It is very easy to compute the q_i. Here is an algorithm that illustrates the ideas; it has been oversimplified in a way that loses accuracy when y slightly exceeds 1.0, so it should not be used in practice. First choose the smallest q_0 with the property that $y_0 = y2^{-q_0} < 1$. When the radix is 2, q_0 is just the exponent[¶]. We have

$$\frac{1}{2} \le y_0 < 1, \quad y_0 = 2^{-q_0}y. \tag{3}$$

Next if $y_0(1+\frac{1}{2}) \ge 1$ set $q_1 = 0$ otherwise $q_1 = 1$. Computing $y_0(1+\frac{1}{2})$ is easy because it involves just a shift and an add. This quantity is greater than or equal to 1 exactly if the addition overflows. It is this method of computing q_i that is called pseudo-division. Now let $y_1 = (1+\frac{1}{2})^{q_1}y_0$. Then using (3)

$$1 - \frac{1}{3} \le y_1 < 1, \quad y_1 = 2^{-q_0}(1+\frac{1}{2})^{q_1}y. \tag{4}$$

We continue the process. Compute $(1+\frac{1}{4})y_1$ by shifting and adding. If there is overflow ($(1+\frac{1}{4})y_1 \ge 1$) set $q_2 = 0$, otherwise $q_2 = 1$. Using (4)

$$1 - \frac{1}{5} \le y_2 < 1, \quad y_2 = 2^{-q_0}(1+\frac{1}{2})^{q_1}(1+\frac{1}{4})^{q_2}y. \tag{5}$$

In general,

$$1 - \frac{1}{2^n+1} \le y_n < 1, \quad y_n = 2^{-q_0}(1+\frac{1}{2})^{q_1}\cdots(1+\frac{1}{2^n})^{q_n}y.$$

Thus after n steps, the correction factor $1-\eta$ differs by at most 2^{-n} from 1. The calculation of the q_i required only shifting, adding and testing for overflow. On the Intel chip, the

[¶]Adjusting for any bias, of course.

algorithm is carried out for 14 steps, on the Motorola chip 31 steps. Thus the Intel chip requires a slighly more sophisticated approximation to $\log_2(1-\eta)$ than the Motorola chip. To summarize

$$\log_2(y) = q_0 - q_1\Lambda_1 - q_2\Lambda 2 - \cdots - q_l\Lambda_l + \log_2(1-\eta) \quad (6)$$

$$\Lambda_k = \log_2(1+2^{-k}) \approx \frac{2^{-k}}{\log_e(2)}. \quad (7)$$

Notice that the Λ_k get smaller as k increases. In order to minimize roundoff error, these should be summed from smallest to largest. However, this means that they can't be summed as they are produced, but have to be stored in memory until the pseudo-division process is complete. Alternatively, the addition can be done from large to small provided that several guard digits are provided for the summation[‖].

When putting an algorithm on chip, there are always some implementation details that need to be considered. We list one of these details for the pseudo-multiply/divide algorithm. The quantity y_n is very close to 1, and so its binary representation looks something like $.111\ldots11xxx$. Thus its significant digits are far to the right, which increases the effect of roundoff error. It is better to recast the algorithm in terms of $1-y_n$. In fact, since $y_n > 1-2^{-n}$, we can use $2^n(1-y_n) < 1$. For example, to test for overflow, use the fact that

$$(1+\frac{1}{2^n})y_{n-1} \geq 1 \Leftrightarrow (1+\frac{1}{2^n})\left(2^{n-1}(1-y_{n-1})\right) \leq \frac{1}{2}.$$

When is it appropriate to use CORDIC algorithms? Their great strength is that they do almost no multiplies, but instead do their work with shifts, adds, and table lookups. In addition, they can be pipelined, since the pseudo-divide bits can be computed in parallel with table lookup, and also in parallel with adding the table elements, provided guard digits are used. However, on machines with fast multiply hardware, the CORDIC approach is usually inferior to a more conventional approach using approximation by rational functions.

4 The Stack in Floating Point Computation

It is not unusual for the depth of the subroutine call stack to vary quite dramatically during a computation. However, floating point computation almost always has a very short stack depth. This has implications for the design of a stack meant to hold floating point procedure call arguments. It suggests that a good way to design such a stack is to have a modest stack in registers and put the overflow into ordinary memory. Since the depth of floating point procedure calls is usually short, overflow into memory should rarely occur. This was the original intent for the 8087, and was finally properly implemented in the 80387.

[‖]Steve Walther works for Hewlett-Packard. While the Motorola chip was being designed, Motorola engineers would call him from time to time for implementation advice. Walther's manager got wind of this and put a stop to it just before Motorola could inquire about the proper number of guard bits. As a result, the Motorola chip doesn't have enough guard bits, and can get logarithms inaccurate in the lower 4 bits.