IMPLEMENTATION OF ALGORITHMS

PART I

Technical Report 20

W. Kahan

1973

Lecture Notes By

W.S. Haugeland and D. Hough

Department of Computer Science
University of California
Berkeley, California 94720

1973

The Best Possible Square Root Routine

This column describes a few hours' work by an amateur, using techniques ranging in age from a few decades to a few centuries. We'll now briefly glimpse how a world-class numerical analyst attacked the problem of constructing a numerical routine. Professor W. Kahan lectured on "Implementation of Algorithms" in the early 1970s; lecture notes taken by Haugeland and Hough appeared as Berkeley Computer Science Technical Report #20 and are now available as National Technical Information Service Report AD-769 124. Of the 339 pages in the report, 53 are devoted to the construction and error analysis of an unbeatable square root routine for the IBM 7094.

Kahan starts by specifying the properties that the routine should have. Monotonicity implies that if $x \geq y$, then $sqrt(x) \geq sqrt(y)$. He demands that $sqrt(x*x) = x$, but observes that one cannot ensure that $sqrt(x)*sqrt(x) = x$. He gives particularly stringent requirements on the accuracy of the routine: his code gives incorrectly rounded answers for only 29 distinct mantissas.

Kahan's routine uses a few loop-unrolled Newton iterations (which he calls Heron's rule) after getting a good starting value. He observes that the best method for finding a starting value is quite machine-dependent (on such issues as the relative cost of table lookup versus multiplication). Kahan therefore built his routine by considering a tree of all possible sequences of IBM 7094 instructions:

> You begin by doing necessary things, like loading the arguments. . . . I used to work on [the tree] in the evenings. It took several—3 or 4 or 5. It did cover a big table. I would connect one branch to another, indicating that they computed the same function, using leftover telephone wire.

His routine was about 20 percent faster than the previous system routine, and much more accurate. Because he considered the tree of possible programs, Kahan could ensure that his program could never be beaten.

Kahan knew that his methods weren't easy:

> [This analysis] may have frightened you into thinking that to write a square root routine you have to have spent years studying abstruse theories. I guess if you want to write the best possible square root routine, maybe you do. There is a limit to how near perfection it is worthwhile to come, and it is not my intention to suggest that you should write a program in this way, since only a simple program could be optimized by examining a tree structure in this way.

And he kept perspective on his work:

> You're trying to ask me, was it worth the money spent. Of course it wasn't worth the money spent if you want to figure it in terms of the number of happier users. I probably tested more numbers than will be run through the SQRT in a year on the 7094.
>
> But we are trying to see how well we can do. For the practical question, I hope most people would have stopped [at an earlier routine]. We can't afford too many guys like me. But we can't afford to do without them, either.

I enthusiastically recommend this document. If you're fascinated by a "best possible" program for a substantial task, you can't afford to do without reading Kahan's report.

ABOUT THESE NOTES

These notes consist primarily of transcriptions of lectures given in the fall of 1970 and the winter of 1972. For publication purposes they have been somewhat arbitrarily divided into two parts. The first part contains basic material while the second discusses some problems arising at a slightly higher mathematical level and includes some appendices.

Within parts the order is roughly the order of presentation in 1972, but the reader need not feel bound to read the topics in the order presented. Cross references between sections are indicated thus: [1].

There is some duplication of material from the two sets of notes which were merged to form the present parts. We have taken the course of not removing duplicate material whenever it seemed possible that something of value might be lost. Furthermore, another technical report[†] discusses some of these same topics in less detail, and can be recommended as a summary.

D. Hough

[†]W. Kahan, "A Survey of Error Analysis," Computer Science Technical Report #4, University of California, Berkeley, 1971.

CONTENTS

---

[1] Includes paper by F. Dorr and C. Moler.

[2] Includes report by students.

[3] Includes report by students.

[4] B. Bridge, B. Deutsch, and R. Gordon

[5] By students.

[6] Condensed from report by D.S. Lindsay.

0. INTRODUCTORY REMARKS: MOTIVATION AND OUTLINE

An Example for Motivation - An Anomaly

Consider a FØRTRAN program that contains the following statements:

```
      X = ...
      Y = ...
      IF(0 .LT. X .AND. X .LT. 0.1) GØ TØ 1
      .
      .
      .
1     IF(10. .LT. Y .AND. Y .LT. 100.) GØ TØ 2
      .
      .
      .
2     P = X*Y
      IF(P .EQ. 0) GØ TØ 3
      .
      .
      .
3     CONTINUE
```

It is possible to reach statement 3 on the CDC 6400 even though you've checked for x and y not being zero. How can this happen? Is there anything wrong with it? Which laws of arithmetic can you expect a computer to obey?

Typical Difficulties

1) There's the problem that only a finite number of numbers can be represented on the machine.

2) CDC supplies something called ∞ and $\pm$ (indefinite). Are CDC's rules in handling these reasonable? Is it reasonable that you should get thrown off the machine if you try to use these numbers?

3) We'll discuss how hardware and software design influence how careful the programmer has to be, and what can be coded around economically.

4) You are to write a subroutine that will solve for the roots of a quadratic equation, given A, B, and C. The equation is $Ax^2 - 2Bx + C = 0$; A, B, and C are single precision, floating point

numbers. The roots are to be accurate to within a few units in the last place, or if a root is out of range, there should be an appropriate message.

5) How can you solve $f(x) = 0$, where f is supplied by some subroutine? Is it possible to write such a program, given an 'a' and 'b' such that $f(a) \cdot f(b) < 0$?

If you use the binary chop method (bisection) it is costly. You have to compute

$$C = \frac{A+B}{2.0}$$

and on some machines C need not lie between A and B. (This is on octal or hexadecimal machines.) What if A+B overflows?

6) Think of $Z = X + iY$ and wanting to compute $CABS(Z) = \sqrt{X^2 + Y^2}$.

If X or Y is about half way to the overflow threshhold, the square will overflow (same for underflow). It would be worse if a power greater than 2 were involved.

So you try the subterfuge:

$$CABS(Z) = ABS(X)*SQRT(1 + (Y/X)**2), \quad |X| \geq |Y|$$

Then you only get an overflow message when you do deserve it, from the multiplication between ABS(X) and SQRT. But you might get an underflow message, which doesn't interest you, except that you'd get thrown off. Should that happen? If you turn off the message, you might miss an important underflow. Should things be this way?

7) Computation of elementary functions: ln, sqrt

Someone was computing:

SQRT(SQRT(X**2 + Y**2) - X)

and iterating it. He got the message that he was trying to take the square root of a negative number.

It turned at that on his machine, SQRT(.499...9) was slightly greater than SQRT(.500...0). The discrepancy was only 1 in the last place. Should the machine mimic the monotonicity of the square root function? What properties of elementary functions should be preserved by the machine? Should $f(f^{-1}(x)) = x$ always hold? How can you insure that elementary function subroutines will do reasonable things?

8) Perhaps you have a system of linear equations that have no solution:

$$x + y = 1$$
$$x + y = 2$$

Then consider the system:

$$x + y = 1$$
$$(1 + 10^{-10^{10}})x + y = 2$$

This second one is not singular, but is it reasonable to expect an answer? Should the computer distinguish the two systems? Usually it isn't practical to do so.

Outline of Topics To Be Covered

Topics are not necessarily to be covered in the order to be discussed below, because they interlock to a substantial extent.

1) Can we axiomatize the design of computer floating point hardware? There are two ways of looking at this question.

i) Set up the axioms in advance and design the hardware to fit. Most

of the sets of axioms proposed are too expensive to implement. One of the causes of the expensiveness is called (by Kahan) the table-maker's dilemma: How do we construct a table (or subroutine) which contains entries correct to half a unit in the last place? For instance if we compute a value

3728.49

and we wish to carry only four digits, can we safely call it 3728? We all know that binary machines often yield SQRT(4) = 1.999999999.... to the limit of their accuracy. Perhaps the answer to the cited problem is 3728.499999..., that is, 3728.50, so we should write 3729 in the table.

Now, by increasing precision, the table maker can get more digits. Suppose they continue to be nines. The table-maker's dilemma is when to stop computing and start trying to prove a theorem that the answer is precisely 3728.5. And the table-maker's dilemma inexorably causes the cost of floating point hardware to go up, if it is to yield correctly chopped or rounded results on all computations.

ii) Another approach to axiomatization is to find a set of axioms that describes the hardware on the better existing computers. But the axioms would not be categorical because computers differ so much. You could not prove programs correct with such axioms.

Examples are the "multi-precision swindles." A program will be displayed which appears to be machine-independent, and, by practically every test, should work on every computer with every input. But there are rare examples for which the program will not work, which of course prevent us from proving that the program will work.

Another problem with non-categorical axiom systems is that they may lead to proofs that certain calculations can't be done. The proofs are

correct deductions from the axioms, but on many computers the calculations can still be carried out with good results!

An excellent discussion of axiom systems may be found in Knuth's volume 2. Unfortunately his axioms are too expensive to implement. But we can describe a set that are similar to his but reasonable in cost.

2) For concreteness we shall attempt to solve the quadratic equation. Can we get the roots accurate to a few units in the last place? If we think we have done so, can we prove it? We shall discover that, as in all of numerical error analysis, we need to learn more about the problem than we had thought we needed. We shall see that error analysis is so unpleasant that it should be facilitated by the hardware design to the greatest possible extent.

3) Next we will see what has been done in the area of automatic error analysis. The inadequacy of the conventional wisdom with regard to significant figures will be demonstrated with an example: the QR algorithm applied to find the eigenvalues of a tridiagonal symmetric matrix. (See Wilkinson's book on the algebraic eigenvalue problem.) This algorithm uses similarity transformations which preserve eigenvalues. However, it is perfectly possible that the elements of the matrix produced by the QR algorithm by exact computation differ in every significant figure from those computed in finite precision. Yet the end result eigenvalues may still be correct to within a few units in the last place! Clearly the traditional ideas about significant figures are unreliable. Yet were we to alter any element in its last place, we would perturb the final eigenvalues far more than any rounding error! Though no figure of the elements of the intermediate matrix is correct, they are all "significant."

Interval arithmetic is a refinement of the significant figure idea which can be very helpful when not abused. Yet it is not difficult to use correctly, as we shall see. R. E. Moore has written a book, Interval Analysis, and an article, in English, in the Czech journal Aplikace Matematiky in 1968.

4) We shall decide what to do about overflow and underflow. Most people think of over/underflow as a blunder. Yet we shall see that the wider the exponent range on a machine, the more likely people are to be troubled by over/underflow, even though each calculation is less likely to over/underflow. The reason is that they attempt larger problems over a greater range of data, so that their intuition will be more likely to fail -- causing unexpected overflow or underflow. Yet in many cases the job should not be aborted but merely computed in a different manner.

5) This leads to the question of execution-time diagnostics. Can we design a system that will tell the user what went wrong, and where, without drowning him in an octal dump? Several good systems have been designed, and an excellent project would be to study these systems. How are they related to interactive computing? Does the environment make any difference in how we treat errors? What will the user do with a diagnostic? Does the error have any significance to the user? Perhaps we need to send the diagnostic information to the calling subroutine rather than perplexing the ultimate user unnecessarily. Can we design a system that will never bother the user unless it is really necessary?

6) How do we prove that our programs are correct? Proofs are as susceptible to error as programs; Kahan's Theorem asserts that any proof longer than four pages is likely to be wrong. We shall prove a square root

subroutine in a few pages.

Most of the standard proven programs are combinatorial in nature and suggest their proofs by induction. In numerical analysis the proof usually is not so directly suggested by the problem. Algorithms in, for instance, differential equations, tend to bear little resemblance to the problems they attempt to solve. An appeal to complex variables will be required in the proof of our quadratic solver. In general we try to show that our incorrect calculation yielded the slightly modified result of a correct calculation on a slightly modified input. This is not always possible!

1. SIGNIFICANT DIGITS, CANCELLATION, AND ILL-CONDITION

How Many Digits Should We Carry?

We shall consider a specific, simple calculation to demonstrate how our usual rules for carrying digits are misleading. Then we will be better able to decide on a reasonable set of axioms for floating point arithmetic.

When we write $A = B + C$ in a Fortran program we are really thinking of three variables a, b, c which reside in memory cells labeled A, B, C. Our hope is that when this statement is performed the sum $b + c$ will be placed in cell A. In general, however, the sum is rounded:

$$a = (b+c)(1+\alpha) \quad .$$

To be specific, let $b = 1.732$ and $c = .004290$, on a 4-digit machine. Now $b + c = 1.736290$. But by chopping or rounding the machine will actually set

$$a = 1.736 = 1.736290\left(1 - \frac{.000290}{1.736290}\right) \quad .$$

In general we don't try to keep track of $\alpha = \frac{-.000290}{1.736290}$ because that would be equivalent to carrying all figures. We only retain the information that $|\alpha| \leq \varepsilon$, for some specified ε. What is the worst value that α could take? This is attained in the case

$$b + c = 1.0005 \quad ,$$
$$a = 1.001 \quad ,$$
$$\alpha \doteq 5 \times 10^{-4} \quad .$$

Note also the case

$$b + c = .9995 \quad ,$$
$$a = 1.000 \quad ,$$
$$\alpha \doteq 5 \times 10^{-5} \quad .$$

One might suppose that carrying four digits would limit the size of α to $\frac{1}{2} * 10^{-4}$, but the first example is disparate by a factor of ten. (On a binary machine this factor is two, from which we shall deduce later that binary is a better way of packing precision into storage.)

If arithmetic is always done by performing an exact calculation and then rounding, we can treat addition, subtraction, multiplication, and division in a convenient and uniform way. This assumption is almost but not quite true on most machines, but we will assume it for the present analysis.

When you introduce all the Greek letters into a program that deserve to be there, it can become quite complicated:

$$D = \mathrm{SQRT}(B**2 - A*C) \quad ,$$

$$d = (1+\theta)\sqrt{(b^2(1+\beta) - ac(1+\gamma))(1+\sigma)} \quad .$$

We assume that the square root subroutine gives an error of a few units in the last place so that θ is nearly as small as the other errors.

Let us consider a quadratic $x^2 - 2bx + c$ so that $a = 1$, $b \doteq \frac{1}{2}$, $c \doteq 1$. Then $b^2 \doteq \frac{1}{4}$, $ac \doteq 1$, and $b^2 - ac \doteq -\frac{3}{4}$. But the error will be restricted to a few units in the last place. The error will increase somewhat after the square root is taken, but will still be quite small, so that we can write

$$d = \sqrt{|b^2-ac|}(1+\xi) \quad .$$

And we can see that the ultimate solution will be correct with real part b/a and imaginary part d/a both quite close to the true value.

Cancellation

When can we lose accuracy? On a true subtraction we would have

$$x(1+\xi) - y(1+\eta) = (x-y)(1+\zeta)$$

or

$$\frac{x\xi - y\eta}{x - y} = \zeta \quad .$$

If x is near y, ζ will be large. We examine a different quadratic: $a = 10^{-5}$, $b = 10^{10}$, $c = 10^{-5}$. Then

$$d = \sqrt{10^{20} - 10^{-10}} \quad .$$

Unless we carry thirty digits, the 10^{-10} is negligible. If we only want a few digits in the answer, surely we need not carry thirty digits. So $d = 10^{10}$. If we use the quadratic formula, we get roots of 2×10^{15} and 0. But zero is clearly not a root, and is accurate to no figures. Clearly, "cancellation is to blame." But, the subtraction was done with no error. The error was made when we did not carry thirty figures earlier. Cancellation does not cause error, but reveals earlier errors.

How can we change our algorithm to avoid carrying thirty digits and still get the correct answer? We rewrite the problem in the suggestive form:

$$\sqrt{b^2} - \sqrt{b^2-ac} \quad ,$$

or in general

$$\frac{f(x) - f(x-h)}{h} h \doteq f'(x) \cdot h \quad .$$

Divided Differences

We could compute the product $f'(x)h$ accurately; unfortunately, it is

a mathematical approximation true only in the limit. But we can circumvent the latter difficulty by introducing the divided difference, which is a function of two points x_1 and x_2:

$$\Delta f(x_1,x_2) = \frac{f(x_1) - f(x_2)}{x_1 - x_2} \qquad x_1 \neq x_2 \quad ,$$
$$= f'(x) \qquad\qquad x = x_1 = x_2 \quad .$$

Suppose, for example, $f(x) = x^n$. Then

$$\Delta f(x_1,x_2) = x_1^{n-1} + x_1^{n-2}x_2 + \cdots + x_1x_2^{n-2} + x_2^{n-1} \quad .$$

In this case we can do the division symbolically, and find an expression for Δf that can be computed accurately when x_1 is near x_2.

Divided differences behave much like derivatives. For instance,

$$\Delta(f+g) = \Delta f + \Delta g \quad ;$$
$$\Delta(fg) = (wf)\Delta g + (wg)\Delta f \quad ,$$
$$\text{where} \quad wf(x_1,x_2) = \frac{f(x_1) + f(x_2)}{2} \quad ,$$
$$\Delta\left(\frac{f}{g}\right) = \frac{(ug)\Delta f - (uf)\Delta g}{g(x_1)g(x_2)}$$
$$= \frac{\Delta f - u\left(\frac{f}{g}\right)\Delta g}{ug} \quad .$$

Note that in these cases, the troublesome x_1-x_2 terms have disappeared. We can also deduce formulas for algebraic functions, which are those that can be obtained as solutions of polynomial equations

$$p(f) = \sum_0^n p_j(x)f^j = 0 \quad .$$

Each p_j is a polynomial in x.

For instance if $f(x) = \sqrt{x}$, then

$$1 \cdot f^2 + 0 \cdot f - x \cdot 1 = 0 \quad ;$$

$$\Delta f = \frac{\sqrt{x_1} - \sqrt{x_2}}{x_1 - x_2} = \frac{1}{\sqrt{x_1} + \sqrt{x_2}} \quad .$$

For our quadratic problem

$$\frac{\sqrt{b^2} - \sqrt{b^2 - ac}}{a} = \frac{c}{\sqrt{b^2} + \sqrt{b^2 - ac}} \quad .$$

When we recompute our quadratic problem we find the small root quite easily to be $\frac{1}{2} \times 10^{-15}$. The larger root cannot be computed with this formula, for the same reason that the smaller root could not be computed with the first formula. Our final algorithm yields

$$x_+ = \frac{b + (\text{sgn } b)\sqrt{b^2 - ac}}{a}$$

$$x_- = \frac{c}{a x_+} \quad .$$

With this scheme, cancellation is never a problem. And the moral of the story is that we lose accuracy by rounding. Cancellation is merely the messenger which reports the bad news.

Ill-Condition

Perhaps, after the previous discussion, we thought we could solve a quadratic equation accurately every time. We shall, however, see that even though we can get as good an answer as we could hope for with moderate precision, it still may not be as good as we want. This problem is different from the previous because no tinkering with the algorithm will

help. The general name for this problem is ill-condition.

Consider the quadratic

$$ax^2 - 2bx + c = 0 \quad .$$

The formula $x_{\pm} = x_{\pm}(a,b,c) = \frac{b \pm \sqrt{b^2-ac}}{a}$ indicates clearly enough that the roots are continuous with respect to the coefficients. A small change in the latter, however, may yield an unpleasantly large variation in the roots. Consider, for instance,

$$x^2 - 2x + 1 - \varepsilon^2 = 0 \quad ,$$

with roots $1+\varepsilon$, $1-\varepsilon$. A change in c of order ε^2 causes a change in the roots of order ε. A change of one unit in the eighth place in c changes the fourth place of the roots. This is bad because rounding errors can be thought of as being equivalent to a small perturbation of the coefficients. If we carry single precision throughout, we can expect sometimes to get only half precision in the roots. Recall that we define a good algorithm as one that delivers the slightly altered result of a correct calculation on a slightly altered input. By this standard our quadratic algorithm is a good one. Unfortunately many people do not distinguish between wrong results due to poor algorithms and those due to ill-condition.

2. RULES FOR FLOATING POINT ARITHMETIC

We now turn to the general problem of floating point design. Knuth (Vol. II) starts simply by specifying the format of floating point numbers as, say,

(sign bit)	(base)	(exponent)	(integer)
$\pm$	2^e		$I \quad , \quad M \leq I < 2M \quad .$

The inequalities are added to insure a unique representation. Note that the twos could just as well be 8, 10, or 16, among others. Our rules will not be based upon this format specifically. It illustrates two facts:

(1) the exponents are bounded -- which, however, we shall ignore for a time;

(2) the set of numbers is a discrete finite set.

We formulate rules for desirable sets of numbers.

Rule #1: The set of representable numbers should include 0, 1, and if x then -x. (There are a host of other possibilities such as, if x ($\neq$ 0) then 1/x, but these are problematical. Also note that the possibility of two representations for the same number, e.g. +0 and -0, is not excluded if they really have identical arithmetic properties.)

The next rule will have to be changed later.

Rule #2: If we perform an elementary operation $f(x_1,x_2,\ldots)$ on representable numbers, and the result is not representable, then it should be approximated by the nearest representable number. Examples of elementary operations might be +, -, /, *, and base conversion. Note that there is an ambiguity here, when the result is precisely half-way between two representable numbers. This defect will be dealt with momentarily.

More troublesome is the question, can we actually afford to implement these rules? The answer will be discussed in the next lecture.

Returning to the ambiguity in rule two, we offer another rule.

Rule #3: Resolve the ambiguous case in a way that preserves as many relations as possible. A relation is a statement like $(x+y) = -((-x)+(-y))$. The widely used rule "add one-half in the last place cited and truncate" does not preserve this relation.

Relations that could be preserved could be characterized in the following way:

Consider three functions: $f(x)$, $g(x)$, $h(x)$, which map representable numbers onto representable numbers.

Example: $f(x) = -x$ or $f(x) =$ constant

or $g(x) = 2^K x$ (binary machine, K an integer)

These map representables onto representables except for over/underflow.

To resolve ambiguities in a systematic way, we would want to preserve the following property: if $h(x \theta y) = f(x) \theta g(y)$ we want the machine to preserve that relation too. If we round $h(x \theta y)$ and round $f(x) \theta g(y)$, we want the relation to still be true.

Example: $f(x) = -x$, $g(x) = -x$, $h(x) = x$, $\theta = *$

This example is preservation of sign symmetry. If we reverse the sign of x and y, the sign of the product shouldn't change.

On one machine, this didn't happen. It used a subroutine for multiplication. If you reversed the sign of one operand (2's complement machine), it would not reverse the sign of the product. If you reversed the signs of both operands, you would get something really funny.

On another machine, people used its divide subroutine for three years

without knowing that you get the wrong result if you divide by a negative number:

$$\frac{x}{y} \neq \frac{-x}{-y} \neq \frac{+x}{y+1}, \quad x, y \text{ integers}$$

If we could follow these three rules they would be categorical. All elementary operations would have a result uniquely determined by the rule for the ambiguous case. An example of a good rounding rule is to round to the nearest "even" representable number. This rule preserves the sign symmetry. (We must, however, define the two numbers nearest zero on either side to have the same parity.)

Consider now the consequence of some other rounding rules, in this Fortran program:

```
1   X = Y+Z
    Y = X-Z
    GO TO 1
```

Suppose we have four digits and we chop. Let the starting value of $Y = 1.000$ and $Z = 10^{-5}$. As we go around the loop we get $X = 1.000$, $Y = .9999$, $X = .9999$, $Y = .9998\cdots$. Obviously Y will have many values in this loop. Is there an arithmetic system which will recover the value of Y every time? None is known. But if you satisfy rule 2 and rule 3 you will get a short finite sequence of values for Y.

Now we shall see that the question is not to round or chop, but how you resolve the ambiguity. This time we add one half in the last place and then chop. Start with $Y = .1000$, $Z = 5 \times 10^{-5}$ and find $X = .1001$, $Y = .1001$, $X = .1002$, $Y = .1002$, $\cdots$.

If, on the other hand, we round to the nearest even representable number, we find, for the same computation, $X = .1000$, $Y = .09995$,

X = .1000, Y = .09995, ··· . Y has changed, but only once.

These considerations are not trivial. When updating a decimal tape on a binary computer by copying, with some computation, the binary to decimal and decimal to binary routines should be approximately inverses. Yet on some machines that chop, they are not. Another example is certain eigenvalue algorithms which perform a "shift of origin" on the diagonal elements of a matrix, massage all of the matrix, and then undo the original origin shift.

For this general problem, see Dave Matula in CACM.

Exercise. Suppose we represent numbers as a sign bit followed by log |x| in fixed point XXXX.XX . What are the interpretations of our rules?

Discussion of Exercise

We would represent numbers by a sign bit and the log |x| in fixed point. This has been discussed by D. Matula, D. Muller, and Gauss. Clearly we can do multiplication and division completely accurately, except for overflow, because these operations involve fixed point addition or subtraction. An add or subtract is more expensive in this system! The technique is called addition logarithms. See Fletcher, Miller, and Rosenhead, An Index of Mathematical Tables. What properties would such arithmetic have?

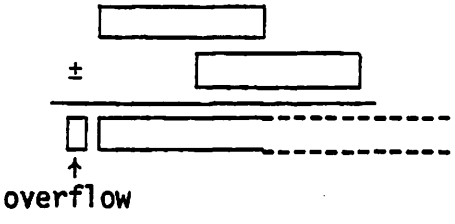
The distributive law is satisfied precisely. But only one of the integers 2 and 3 is representable in this system! [Is this worse than 2 and $\sqrt{2}$ not both representable?] This system has never been fully explored, so that we can't say that it's better or worse than the usual. Certain little tricks don't work that we depend on occasionally to give exact results, e.g. the difference of two nearly equal numbers would not be precise in the log system.

3. COST OF THE RULES

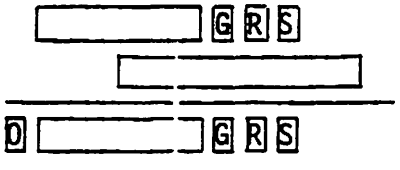
Our purpose is to demonstrate why our rules are too expensive to implement. Some of these costs are begrudged unjustifiably, though for others there are fairly good reasons.

Addition and Subtraction

Consider first addition and subtraction. We right shift and then add. How many extra digits should we carry for the result?



We know that no more digits need be carried than the range of our exponent, which, however, is much too many. It suffices to carry a guard digit, a round bit, and a sticky bit:



O = overflow
G = guard
R = round
S = sticky

You could even time-share the overflow and sticky bits, though it hardly seems worth it. The sticky bit tells you if any non-zero digits have dropped off the right in the right shift. Now the simplest case is when overflow occurs. Then we can round by adding five (for decimal) or 1 (for binary) in the last place. We will then shift right. In the ambiguous case, we instead round to even.

The next possibility is that no overflow has occurred but that the number

is already normalized. We add $\frac{1}{2}$ in the last place, i.e., 5_{10} or 1_2 in the guard digit, unless round to even is indicated by $G = 5_{10}$ or 1_2 and $R = 0$ and $S = 0$.

If the result is unnormalized with one zero on the left, we round in the round digit, after checking the sticky bit. If $S = 0$ and $R = 5_{10}$ or 1_2 a round to even is required. If there are two or more zeros on the left, the right shift of the smaller operand was not past the guard digit. No rounding is required.

Many people think they can get along with just a guard digit. But they can't give you correctly either rounded, or chopped arithmetic with such a scheme! Correct chopping is defined, by the way, as replacing the result by the nearest representable number no larger in magnitude. Suppose you have one guard digit and you subtract a much smaller number from a larger. The much smaller number is shifted far off to the right and off the end and there is nothing to show for it (without a sticky bit). The result is certainly correct to one in the last place. This result will be a little bit too big, however -- and therefore not correctly chopped. Why is this bad? The answer appears to be more accurate. On a four digit machine surely the better answer to $1.001 - 1.000 \times 10^{-10}$ is 1.001 rather than 1.000, the correct chopped answer. It is surely more accurate. But what do we know about the end result? With correct chop we know that the true answer is in the interval [1.000,1.001). But with the "better" scheme, the true answer could lie in (1.0009,1.002) which is a 10% larger interval! That is, though the accuracy is better, the uncertainty is slightly greater! And at the end of a calculation we want as small an uncertainty as possible.

If we know that, mathematically, the values in storage satisfy

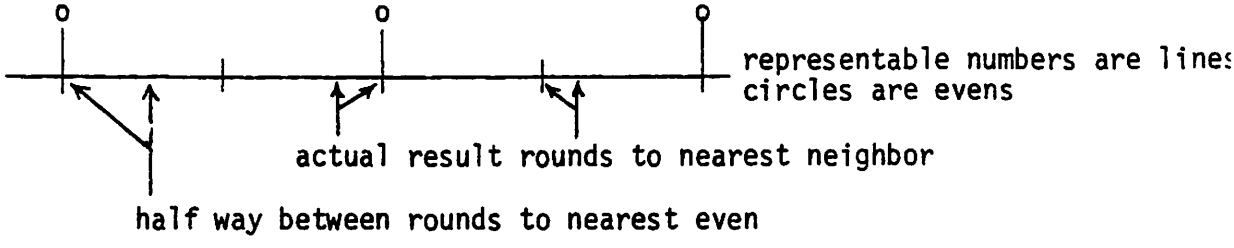
$$a + b = x + y \quad ,$$

should it be possible for A+B .EQ. X+Y to be false?

Exercise: Discover the circumstance in which this can happen.

Question: Do you round a negative number as the absolute value of the number or round it towards zero?

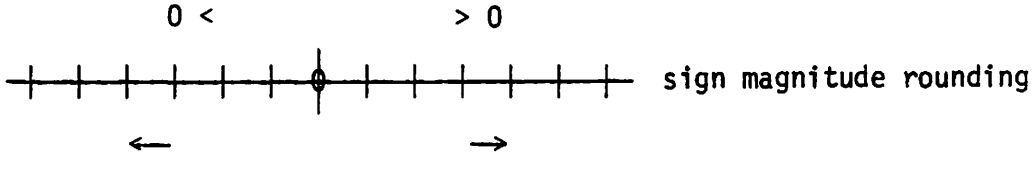
Answer: You round a negative number to its nearest neighbor unless it is half way between. That has nothing to do with where zero is.



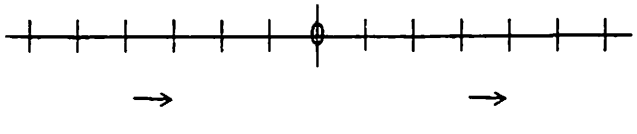
You don't care where zero is in rounding.

Question: For normal machines that round, they usually just add 1 in the last place.

Answer: Sign-magnitude machines do just add 1 in the first bit to be discarded.

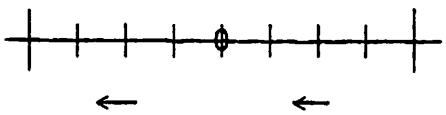


In a 2's complement machine like G.E. 635, when you add that 1 bit in, it moves everything to the right; it doesn't necessarily move the number closer to zero.



In a 1's complement machine like 6400 the same thing happens as in a sign magnitude machine.

If you truncate (throwing digits away), you are always moving to the left.



I don't want that to happen in my scheme.

Question: How does 2's complement work?

Answer: Example of 2's complement rounding.

$\bar{1}.00\ 1$ -7/8 in two's complement

$\bar{1}.00$ truncated in two's complement

representation for -1, so magnitude has been increased for a negative number

You need 1 or 2 round bits. But could you get along without the 'sticky' bit? Students should verify for themselves that you cannot round to the nearest neighbor without a 'sticky bit.'

Students should also verify that if you have to left shift the answer by more than 1 bit, the answer is exact. Only when a single left shift is required is there any problem.

How About Bias?

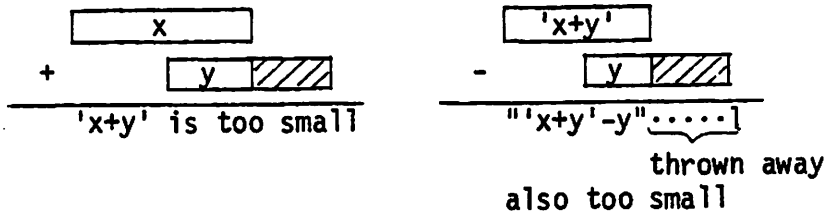
Has "bias" or "drift" been eliminated by this construct?

Will the following sequence be prevented from "drifting"? Take x, y arbitrary and $x_0 = x$. Let $x_{i+1} := (x_i+y) - y$. Is $x_1 = x_2 = x_3 = \cdots = x_{i+1}$?

Notice that x_0 does not appear in the sequence.

Test on arithmetic already known

truncated arithmetic $x, y > 0$.



$x_{n+1} < x_n$ by at least 1 unit in the last place, maybe 2, if y has some 1's that got thrown away.

You could push x down until it is comparable to y; then the process would settle down.

A similar thing can happen if you round up in the conventional way -- that is, add 1/2 in the last place and throw away the fraction. But then you drift up instead of down.

Example. x_n = 1.00001101 9 significant bits

y = .10000001

'x_n+y' = 1.100011011 ⟹ 1.10001110 stored 'x+y'

-y: .10000001

1.00001110 ⟸ 1.000011001

when stored

stored(stored(x+y) - y) ≠ x that you began with

The final value has increased by 1 in the last place. This will happen every time until the initial 1. in x has become 10. Then the extra digit in y gets right shifted off and the sequence settles down. But you can as much as double x by repeating the process long enough.

Same example by rounding to nearest even.

The first time through you get the same result:

(1) stored(x+y) = 1.10001110

(2) stored(stored(x+y) - y) = 1.00001110

Now go through the cycle again:

```
   1.00001110
 + .100000001
   1.100011101:  rounding ⇒ 1.10001110

   1.10001110
 - .100000001
   1.000011011:  rounding ⇒ 1.00001110
```

Thus $x_{n+2} = x_{n+1}$

There is a change in the first step if x is odd. Then the sequence doesn't "drift".

You prove this in general by examining all the different cases.*

Question: How about other possible sequences and drift?

Answer: If multiply/divide are in your sequence, it will also settle down. Arguments are similar to those used by Dave Matula in papers on base conversion. (See his papers -- usually have 'base conversion' in title, look in ACM.)

In the multiply/divide case, you can verify the result by observing that the error can't exceed half a unit in the last place, in either multiply or divide. So in one step, the error can't exceed 1 in the last

*For $y \leq x$, you don't have to shift y. Then nothing interesting happens. If you have to shift y, some digits will fall to the right of x. Does addition cause x to overflow? Follow one branch. If you don't have to right shift, look at the digits to the right. Say $y > x$: right hand digits of x_0 get stripped off, then no rounding errors.

place. You need to show that if the error was 1/2 in both cases, something peculiar happens and show that that just doesn't happen.

Question: Why isn't it economical to build a machine that rounds your way?

Answer: I said it has not been thought worthwhile to do it this way. People who build machines don't see that there is much value in building machines that eliminate the bias. (Neither does Knuth as he doesn't discuss it at all.) I'm not sure people appreciate what would happen if you eliminated the bias. Certain iterations would work better, on the average. Certain identities would be preserved. It would make it easier to prove certain relations about iterations, such as ultimate convergence.

Example. $\sqrt{z}$, $z > 0$

```
      y0 = (1. + z)/2.
1     yN = (y0 + z/y0)/2.
      if (yN .EQ. y0) go out
      else y0 = yN
      go to 1
```

On a truncating machine, one thing can happen and on a rounding machine, another.

You try to prove that this algorithm will terminate (on some reasonable machine) with two successive equal values for y.

In principle: $y_1 > y_2 > y_3 > \cdots > y_n > \sqrt{z}$

In reality: one $>$ sign becomes an $=$ sign and you stop. You've come as close as you want to $\sqrt{z}$.

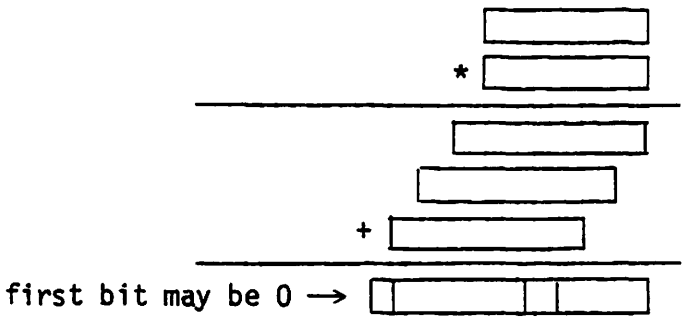
If drift does not exist, it is easy to prove that this terminates. You can also prove that it will terminate for rounding machines. But it might not terminate for machines which truncate.

Question: Then the purpose of sticky bit is to prevent drift and that's all?

Answer: Yes, it prevents drift and it makes things correctly rounded. If you want the machine to truncate correctly, you would still need a sticky bit. It is not possible to achieve the type of rounding desired with only a guard digit. You can add 1/2 or chop with only a guard digit. You cannot get correctly truncated arithmetic with only a guard digit.

Multiply and Divide

Now let us consider multiply and divide. Can we satisfy our axioms at a reasonable cost? Here is the picture:



When we multiply two single precision normalized numbers we may get at most one leading zero in the double precision result. Clearly we need at most only the leftmost word plus one digit ... except when we might be near the ambiguous case. If we don't care about that rule, we can eliminate about half the work. To follow our rules we must develop the entire double precision product precisely, even though, as on the IBM 360, only one guard digit need be maintained and un-needed digits of the product may be continually dropped off at the right. On the 360/91 many tricks are made to speed up the multiply and divide. See Kuki and Ascoly, "Fortran Extended-Precision Library," IBM Systems Journal, 10, p. 39, 1971, and Anderson et.al.,

"Floating-Point Execution Unit," IBM Journal of Research and Development, 11, p. 34, 1967.

Whatever is done about multiplication, adhering to our rules for division will cost us a factor of two in execution time. Perhaps we've been thinking of the usual division algorithm which gives a precise integer-style quotient and remainder, from which it is easy to implement our rules. The trouble is that our usual division algorithm is too slow. In the search for faster methods, our rules will go out of the window.

The fast division algorithms depend on fast multiplication techniques. We can divide this way almost as fast as we can multiply. We convert $\frac{A}{B}$ to $\frac{D}{1 \pm \varepsilon}$ by a number of multiplications on the numerator and denominator, on the general principle that $(1+\delta)(1-\delta) = 1-\delta^2$, to get the denominator to 1. But at the end we don't know whether to round up or down.

We can do one more multiplication to get a double precision quotient. For instance, a very troublesome division for getting a correctly chopped quotient is

$$\frac{.999\cdots98}{.999\cdots99} = .999\cdots98999\cdots\overset{\downarrow}{9}89\cdots$$

Clearly we must compute to full double precision plus one bit to get to the eight which tells us how to chop. But the extra hardware for a double precision divide might well un-justify the fast division algorithm!

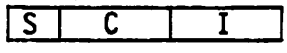
It is hardly surprising that most machines don't follow our rules. The B5500 does correctly in all but one instance. So there is hope! The next lecture will discuss the CDC 6400. For preparation read J.E. Thornton, Design of a Computer: The Control Data 6600, 1970.

4. ARITHMETIC ON THE CDC 6400

Number Representation

We now turn our attention to the capabilities of the local hardware unit, the Control Data 6400. First we need to consider the way the numbers are represented. The 6000 series uses ones-complement floating point representation, so that negatives may be obtained by complementation. For our convenience we will use signed-magnitude representation which is equivalent. That is, we could not tell if the results given by the 6400 were secretly computed in signed-magnitude and then converted to ones-complement for output.

If the number is negative, it is represented as the complement of the representation of its magnitude. Bit zero is the sign bit. Then a positive number is represented as



S = sign bit
C = characteristic (11 bits)
I = integer (48 bits)

C is interpreted by complementing the leading bit and regarding the result as an eleven-bit ones-complement binary integer, which is the exponent e. The reason for this complicated scheme is so that we can compare two floating point numbers by subtracting their entire 60-bit representations as integers. Then the sign of the result would indicate the proper relationship between the original operands. Unfortunately there are so many exceptions that this idea is unusable.

I is interpreted as a 48-bit integer with binary point at the right. Then the number represented is

$$(\pm)\ 2^{e} \cdot I \quad .$$

To make the representation unique we normally consider only normalized numbers.

A number is normalized if $e = -1023$ and $I = 0$, in which case it is a normalized zero, or if $-1023 \leq e \leq 1022$ and $2^{47} \leq I \leq 2^{48} - 1$. In binary, write

$$100\cdots00 \leq e \leq 011\cdots10$$

$$100\cdots00 \leq I \leq 111\cdots11 \quad .$$

With this restraint every number has a unique representation. We shall see that this is important in floating point units of the CDC variety. On, for instance, the 360, addition but not multiplication is affected by normalization. On the B5500 normalization makes no difference.

There are a few exceptions. If $e = 1023$, then the number is infinite and lies outside the magnitude range 2^{-976} to 2^{1070}. If a number is generated greater than 2^{1070} it is replaced by a characteristic of infinity. If the number you would have liked to generate had an exponent of precisely 1023, then the I part is correct. In general the I part is not related to the true result.

If the true result would have been less than 2^{-976} then it is replaced by zero with no indication to the user, except that he may notice that the product of non-zero numbers is zero. When an infinity is generated there is no indication except the infinity characteristic which may be tested.

The machine may be operated in two modes. In the most common, the subsequent <u>use</u> of an infinite operand aborts the job. The user is given the address of the word in which the instruction was located which tried to use the infinity. He may be able to determine which instruction in the word caused the interrupt

In the alternative mode no interrupt occurs and there may be generated a new kind of object called an indefinite θ, with $e = -0$, in accordance with certain rules, such as:

$$\infty * \infty = \infty \quad ,$$

$$\infty / \infty = \theta \quad ,$$

$$0 * \infty = \theta \quad ,$$

$$\text{number} / \infty = 0 \quad .$$

Note that an indefinite will never go away, but an infinity may disappear! These rules may seem safe but in fact they are not, as we shall see.

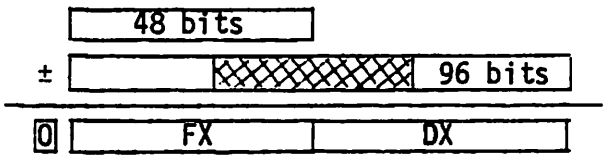
There is a question yet of what constitutes a zero. The multiply and divide units treat any quantity with $e = -1023$ as a zero. This was done to speed processing of sparse matrices by checking for zero factors in advance of multiplication. But the add-subtract logic checks all sixty bits and calls the number 0 only if it is precisely +0 or -0. So it is possible to use a branch on zero instruction to test an operand, which uses add-subtract logic, get a result of non-zero, divide, and get an infinity. This is a mistake in the design caused by the fact that the multiply-divide units test only the first twelve bits of the word. By adding one more logic element to test the thirteenth bit, the problem could be solved. Instead, the problem was given a name (partial underflow) and announced as a feature in the 7600.

Floating Point Instructions

Now we are ready to discuss the floating point operations. There are normal (chop) instructions such as FX, rounding operations called RX, and operations to get the second half of a double precision product, DX.

On an FX multiply you get the 48 most significant bits of the product. A DX multiply on the same operands yields the 48 least significant bits, with an exponent 48 less than the FX results. Note that underflow could happen in the DX result and not in the FX. Except in that case, the double precision product is the sum of the two numbers.

For addition things are not so handy. There is in effect a 96 bit register in which the smaller operand is placed, and then right-shifted, with digits off the end if necessary:



If an overflow occurs on a true add both registers are shifted right one and both exponents adjusted. On a true subtract, however, FX will give you an unnormalized result. Therefore we usually follow with an NX normalize instruction, which, unfortunately, only normalizes the left part. To see the problem here, consider this four-bit example, 1.000 - .1111:

	1.000	0000
-	.111	1000
	0.000	1000
	FX	DX

When the result of FX is normalized the answer is zero, yet clearly the operands are unequal. If we write $A = B - C$ we can't have $a = (b-c)(1+\alpha)$ with α small. In this case, in fact, $\alpha = -1$. The best we can say is that $a = b(1+\beta) - c(1+\gamma)$ with small β and γ, which is substantially less convenient for error analysis. Here $\beta = .001$ and $\gamma = 0$.

We would hope that, if a result could be represented precisely, then it would be. Fortunately in this case we could get around this by extra coding. For the sequence

```
FX2   X1-X0
NX2
```

,

insert instead

```
FX2   X1-X0
NX2
DX3   X1-X0
NX3
FX2   X2+X3
```

.

This gives very nearly the correctly chopped result. We need five instructions! It is hard to persuade compiler writers to generate this much code for such a simple operation.

Rounded Addition and Subtraction

We turn our attention now to the rounded arithmetic instructions which are in a unique form in these CDC machines. Rounding is normally thought of as adding one half in the last place after the operation has been completed. This may generate a carry chain which will slow things down. On addition the CDC units add one half in the last place to both operands before the operation. When the characteristics of the operands are equal the correct result is obtained. When the characteristics differ by Δ, then, in effect, the quantity $\frac{1}{2}+2^{-(\Delta+1)}$ is added to the result, if no overflow occurs. [If overflow occurs, then the quantity is $\frac{1}{2}(\frac{1}{2}+2^{-(\Delta+1)})$.] This quantity might be as large as $\frac{3}{4}$. The results are what we would expect when $\Delta = 0$ or Δ is large. But the overall arithmetic is very hard to

predict, and situations which may be bad in FX are worse in RX.

In particular, whenever we know that x+y = a+b, we want X+Y .EQ. A+B. Yet this is sometimes not the case on the 6400 if x and y are opposite in sign and large, but a and b are small. This can occur using either FX or RX arithmetic.

What Relations Does the 6000 Satisfy?

We have mentioned that we would like our hardware, which of necessity must approximate results, to preserve as many desirable relations as possible. For instance, if we know that the numbers represented in the cells A, B, X, and Y, satisfy the relation

$$a * b = x * y$$

then we would want this relation preserved by the machine operation *. The value stored for A * B should depend only on the value a * b, and not on *, a, or b. The purest kind of rule, such as Knuth proposes, whereby we perform the operation correctly and then round, is ideal. On the 6400 we can nearly achieve this goal on FX*, FX/, and FX± using the five-instruction sequence given in the previous lecture.

Exercise: Verify that the results of the operations indicated are very nearly independent of the operands.

There are, unfortunately, plenty of discrepant cases on the CDC machine. The ordinary FX+ and NX sequence provides several. Consider the following program:

```
      X = 0.5
      F = (X-0.5**48)+X
      DO 2 I=1,100
         X=X*2.0
         Y=X*F
         IF(X.EQ.Y.AND.(X-1.).NE.(Y-1.)) WRITE(I=...
2     CONTINUE
```

If this program is compiled on the standard RUN compiler and then executed, the message is printed for $I = 2,3,4,\ldots,48$ and for $I = 97$. Note that the value of X in the loop is 2^{I-1} and the value of Y is $2^{I-1}(1-2^{-48})$, both computed precisely. The problem here is the compiler's test for equality:

```
FX3   X1-X2
NX3   X3
ZR    X3,...   .
```

The problem is that the difference between X and Y is developed in the DX part of the sum. The FX part is zero, so X appears to be equal to Y. For any machine in which the result depends too strongly on the operands, such an example can be constructed.

One easy fix that suggests itself is to compile

```
IX3   X1-X2
ZR    X3,...   .
```

But now another type of program can get into trouble, namely one that includes a statement of the form

```
IF(X.NE.Y).....A/(X-Y)   .
```

That is, we have trouble with the uncertain definition of zero, since the

X.NE.Y will be done by an integer subtraction while the divide unit will receive the result of a floating point subtraction which may well be zero. To get by on a CDC machine we could write

$$\text{IF(X-Y.NE.0.0)...A/(X-Y)} \quad ,$$

but we have lost a degree of machine independence. The RUNW compiler has been modified by D. Lindsay to perform these tests in a reasonable way (see Appendix II).

We have an example of the CDC RX± instructions provided by Wirth in five bit arithmetic. In five bit arithmetic the number 33 is not representable so it should be represented by either 32 or 34. In an RX instruction to add 16 to 17 we get

```
                      Round bit
   1 0 0 0 0    1
 + 1 0 0 0 1    1
-----------------
 1 0 0 0 1 0---0 = 34
```

Now if we add 31 and 2:

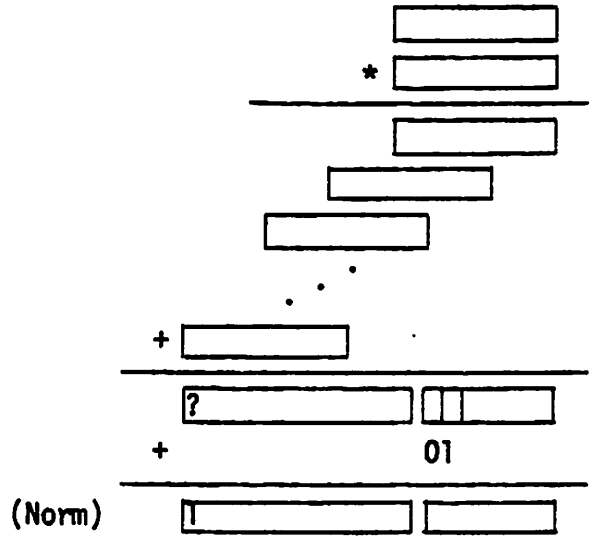
```
   1 1 1 1 1    1
 +      1 0    0 0 0 1
-----------------------
 1 0 0 0 0 1---1 0 0 1 = 32
```

So the CDC RX instructions have the same problems as the FX.

Rounded Multiplication

We now consider RX*. Recall that in FX* the result is independent of the operands. In RX, if both operands are normalized, then the round

is accomplished by adding a one to the result prior to final normalization:



(The 01 is present at the start of the multiplication in order to avoid a long carry at the end.) This scheme is reasonable if post-normalization occurs; we have added one half in the last place. But if no normalization occurs, we have only added a quarter. So the error in an RX* may be almost as large as $\frac{3}{4}$ in the last place, compared to 1 for FX*. But now the end result depends on the operands. Consider

$$A = 2^2*(2^{46}-1)$$
$$B = 5*2^{45}$$
$$X = 2^{24}(2^{23}+1)$$
$$Y = 5*2^{23}(2^{23}-1)$$

Then, although ab = xy, RX(A*B) < RX(X*Y)! Still, the difference is only one unit in the last place ... which is not serious, unless the difference is between zero and not zero.

An Example

Here's an example, in 2 decimal arithmetic, utilizing CDC's method of prerounding, in which

$$A*B \neq C*D$$

even though, in truncated arithmetic, the products are equal.

A = 45
B = 19

```
  45
× 19
----
0855
 05
----
0860 → 860 as the answer
```

C = 95
D = 9.0

```
  95
 9.0
----
8550
 05
----
8555 → 850 as the answer
```

Question: Is there a large range of numbers that will do this?

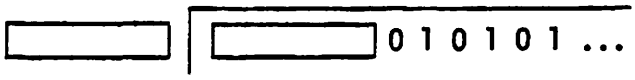
Answer: I used a table of factors, taking a number that had lots of factors and which had a leading digit that was large. I wanted it to be an eight, or the example doesn't work out so nicely.

The significance of this example is not that something is going to happen to you if you use rounded arithmetic but merely that rounded arithmetic on a CDC cannot be characterized by Knuth's rule that the result should be obtained from the true value by following a rounding prescription. This is true because the rounded result depends not only on the end value but also on intermediate values.

To get a correctly rounded product, we can do

```
FXO   X1*X2
DX3   X1*X2
RXO   X0+X3
```

For division, CDC adds $\frac{1}{3}$ in the last place to the dividend -- that is, the binary string 0101... .



On the average, one can expect the quotient obtained here to differ from the true quotient by zero. Curiously, on the 7600 the quantity $\frac{1}{4}$ is used instead of $\frac{1}{3}$. This could cause endless agonizing if we try to transfer a program.

Unsolved Problem: Is there an example of an RX division in which $x \div y = a \div b$ but X/Y.NE.A/B?

Unfortunately, there is no easy way of getting a quotient correctly rounded on the CDC 6400. The best that can be done is to take the given FX results, multiply it times the divisor to get a double precision product which is then subtracted from the dividend in double precision to get the remainder. Dividing the remainder by the divisor yields the correction which must be added to the first quotient.

Peculiarities of CDC Roundoff Error[†] (by Fred W. Dorr and Cleve B. Moler)

Kahan[††] proposed the following Fortran program as an indicator of computer roundoff error:

```
H = 1.0/2.0
X = 2.0/3.0-H
Y = 3.0/5.0-H
E = (X+X+X)-H
F = (Y+Y+Y+Y+Y)-H
Q = 2.0*F/E
PRINT, Q
```

[†] SIGNUM Newsletter, Vol. 8, No. 2, April 1973.

[††] W. Kahan, "A Problem," SIGNUM Newsletter, Vol. 6, No. 3, 1971, p. 6.

The problem is to find what possible values Q can assume on different computers. Kahan intended that ABS be used in the computation of Q, but we have found that the sign of Q is also interesting.

Thorough analysis of complicated numerical algorithms requires detailed understanding of computer arithmetic. Study of simple algorithms such as this helps in that understanding. We have run this program on the CDC 6600/7600 computers at the Los Alamos Scientific Laboratory. In describing our results we will call the above set of instructions Program I. We also consider a modification of this program in which the first three lines are replaced by

```
      ONE = 1.0
      TWO = 2.0
    THREE = 3.0
     FIVE = 5.0
        H = ONE/TWO
        X = TWO/THREE-H
        Y = THREE/FIVE-H
```

and we call this version Program II. On both computers it is possible to select either truncated or rounded arithmetic. The resulting values of Q are summarized in the following table:

		6600	7600
Program I	Truncated Arithmetic	-6.0	-6.0
	Rounded Arithmetic	-6.0	-6.0
Program II	Truncated Arithmetic	3.0	3.0
	Rounded Arithmetic	-6.0	4.0

There are two interesting features in this table: the differences between the two "identical" programs on a given machine, and the difference between the two machines on Program II with rounded arithmetic. The two programs produce different values on a given machine because the compiler computes values for the constants 1.0/2.0, 2.0/3.0 and 3.0/5.0 . The value for 1.0/2.0 is exact, but on both computers the value for 2.0/3.0 is rounded up while the value for 3.0/5.0 is rounded down. This occurs because the compiler computes the constants in two separate steps. The first is a rounded reciprocal divide (1.0/3.0) and the second is a truncated multiply (2.0*(1.0/3.0)).

The rounded arithmetic case for the 6600 is the same for both programs because the 6600 "rounded divide" computes exactly the same values for these constants as the compiler computes. Since this is also the reason for the difference between the two machines on Program II with rounded arithmetic, let us explain this phenomenon in detail. The rounded divide instruction, which is actually a form of "pre-rounding," is executed on both machines by the following algorithm: (1) take the 48 bit mantissa of the dividend and append a 48 bit number α after the least significant position, (2) divide this 96 bit number by the 48 bit mantissa of the divisor, and (3) truncate the result to 48 bits. For the 6600 $\alpha = \frac{1}{3}$ and for the 7600 $\alpha = \frac{1}{2}$. This algorithm produces the following rounding characteristics for the constants:

	6600		7600	
	2.0/3.0	3.0/5.0	2.0/3.0	3.0/5.0
Rounded reciprocal divide followed by a truncated multiply	up	down	up	down
Truncated divide	down	down	down	down
Rounded divide	up	down	up	up

The up-down combination leads to Q = -6, down-down leads to Q = 3, and up-up leads to Q = 4. The true values for the constants are:

$$\frac{2}{3} = 0.\ 52525252\ldots_8 \quad ,$$

$$\frac{3}{5} = 0.\ 46314631\ldots_8 \quad .$$

Overflow and Underflow

We now consider what happens on the 6400 when overflow or underflow occurs. The descriptions in the manual seem eminently reasonable, and some experience with the consequences is necessary for a proper appreciation. Suppose that we are operating in the mode which allows indeterminate forms to be used. We have the following program, along with the naive expected values and the actual computed values:

	Expect	Get
X = 2.0**1069	2^{1069}	2^{1069}
Y = 4.0*X	2^{1071} or ∞	∞
Z = Y-2.0*(X+X)	0 or $\pm\theta$	$\pm\theta$
T = (((Y-X)-X)-X)-X	0 or $\pm\theta$	∞ ?
U = 1.0/T	∞ or $\pm\theta$	0
V = X/Y	$\frac{1}{4}$ or $\pm\theta$	0

We expect to get a value that is either correct or indefinite. Unless we simulate each step performed by the 6400, however, we would be surprised by the last three results.

There is no consistent way to compute with infinities. After all, some infinities "really mean" infinity, as in $\frac{1}{0}$, but others mean a number that is just slightly too large to represent.

With underflow the situation is even worse because there is never any indication that it has occurred. Here is an example:

```
Z = 2.0**(2**10-48)
C = 1.0/Z
A = C+C
B = A*10.0**9
D = A+B
X = (B+D)/A
Y = ((AX+B)/(CX+D))/((A+B/X)/(C+D/X))
    (this ought to be 1)
IF (A>0 and B>0 and C>0 and D>0 and X>0 and Y>2.999) WRITE Y
```

All of these numbers are positive and not zero. No subtraction can occur so we don't worry about cancellation. Y might differ from 1 by a few units in the last place.

But this program prints out Y = 2.99999999875. What has happened is that an underflow has occurred without any warning. We find that

$$Z = 2^{976}$$

$$C = 2^{-976}$$

$$A = 2^{-975}$$

$$B = 2^{-975} * 10^9$$

$$D = 2^{-975}(10^9+1)$$

$$X = 2 \times 10^9 + 1$$

Considering the mechanisms for * and / we can actually predict the value of Y, taking into account the threshhold for underflow, as follows:

$AX+B = 2^{-975} * (3 \cdot 10^9+1)$ and the 6400 calculates this precisely.

When we try to compute CX, the multiply unit notices that the

exponent of C is -1023 and calls C a zero, when C is actually not zero but is "partially underflowed." So

$$CX+D = 2^{-975}*(10^9+1) \quad .$$

For comparison, on a cleaned-up 6400 with that 13^{th} bit wired into the multiply zero test

$$CX+D = 2^{-976}(4\cdot 10^9+3) \quad .$$

Then

$$\frac{AX+B}{CX+D} = \frac{3+10^{-9}}{1+10^{-9}} \doteq 3 - 2*10^{-9} \quad \text{on the 6400}$$

and

$$\frac{6+2*10^{-9}}{4+3*10^{-9}} \doteq \frac{3}{2} - \frac{5}{8}*10^{-9} \quad \text{on the cleaner version.}$$

Now $\frac{B}{X} = 2^{-975}(\frac{10^9}{2\cdot 10^9+1}) < 2^{-976}$ and is therefore set to zero as an underflow

$A+\frac{B}{X} = 2^{-975}$ on either system

$\frac{D}{X} = 2^{-975}(\frac{1+10^{-9}}{2+10^{-9}})$ so $2^{-976} < \frac{D}{X} < 2^{-975}$

which means it is partially underflowed. The add logic is not aware of such distinctions and adds it correctly to form

$$C + \frac{D}{X} = 2^{-976}(\frac{4+3*10^{-9}}{2+10^{-9}})$$

which is not partially underflowed. Then

$$\frac{A+B/X}{C+D/X} = \frac{4+2\cdot 10^{-9}}{4+3\cdot 10^{-9}} \doteq 1 - \frac{1}{4}\cdot 10^{-9} \quad .$$

Finally we compute a factor of two difference! $Y \doteq 3 - \frac{5}{4} \cdot 10^{-9}$ on the 6400 and $Y \doteq \frac{3}{2} - \frac{1}{4} \cdot 10^{-9}$ on the cleaner version. The system for underflow and overflow is clearly a serious problem on the 6400. We shall see that more rational schemes can be devised.

Integer Overflow on the CDC 6400

We study integer overflow to illustrate the principle that machines can't be designed piecemeal. There seems to be an inevitable interaction among various features of the machine so that poor design in one unit inhibits efficient action elsewhere.

Our CDC machine was basically designed to facilitate parallel processing of instructions to the greatest possible extent. At any moment it might not be possible to tell what order of execution was intended by the programmer for the instructions currently in various stages of execution. The CDC machines keep fairly well co-ordinated except in a few critical instances. It was decided not to interrupt on overflow or underflow because it was quite possible to get in the situation that, when overflow or underflow was detected, a later instruction had already been started which wiped out one of its input operands so that the program could not be restarted. Other machines with look-ahead features such as the 7094 were prepared to discard some decoded instructions if necessary in order to have over/underflow interrupts. This is reasonable for machines with small sets of registers. In contrast, when the 360/91 was designed it seemed unreasonable to do this, so there is a problem of "imprecise" interrupts. The interrupts occur, but the location of the error specified to the user is usually one or two words from the instruction which caused the error. Thus it was decided that the 6600 would not have such interrupts at all. Today we shall see

what consequences this entails in the case of integer overflow.

```
      I = 2**40
      DO 11 L=1,18
11    I = I+I
      J = I+3
      K = -I
      IF(J>0 AND K<0 AND J+K=3 AND J<K) WRITE...!
```

Why did this program output an exclamation? It was surprised that K<0<J and J<K. What has gone wrong has nothing to do with anything like rounding. Rather, the compiler test for J<K did not take into account the possibility of integer overflow. In this program $I = 2^{58}$, $J = 2^{58}+3$, $K = -2^{58}$. The J<K test is converted to J-K < 0. $J - K = 2^{59}+3$ which overflows, producing a negative number, with ones in bits 59, 1, and 0. With no overflow indication, the machine has erroneously concluded that J < K. It would have been complicated to include overflow indications for the eight X registers so it wasn't done.

The first example may be dismissed as the justifiable result of dealing with immorally large integers. Now suppose we are computing an infinite sum by the following formula:

$$\sum_{1}^{\infty} \frac{n}{1+n^3} = \sum_{1}^{L} \frac{n}{1+n^3} + \frac{1}{L} + R \quad ,$$

and we know that to get the residual $|R| < \varepsilon$ then $L \geq \frac{3}{\sqrt{\varepsilon}}$. Suppose we want $\varepsilon = 10^{-10}$ so L = 300,000. Consider the following innocuous program:

```
      EPS = 10^-10
      L = 3.0/SQRT(EPS)
      INCREM = 1
      WRITE L, The sum of L=... terms is ...
      SUM = 0.
1     DO 2 N=1,L,1
      EN = N
2     SUM = SUM+EN/(1.0+EN**3)
      WRITE SUM
      SUMINF = SUM+1.0/EN
      WRITE SUMINF, The infinite sum is ...
```

Now the output for the program as written was:

```
THE SUM OF 300,000 TERMS IS USER CPU ARITH ERROR
                              I: DETECTED BY MTR, FL=007455
```

As it turns out, we had a division by zero. Clearly this could only happen at line 2 if N=-1. But that can't be. N runs from 1 to 300,000.

After some detective work the user observed that by changing line 1 to read DO 2 N=1,L,INCREM the following result was printed:

```
THE SUM OF 300,000 TERMS IS  1.111640603830
THE INFINITE SUM IS          1.111643937163   .
```

What had happened? As originally compiled the incrementation of the DO loop was done with the SX instruction with an 18-bit adder. 300,000 is so large that the 18-bit adder went through its entire range of positive values, overflowed to its largest negative value, and continued to increment until it reached a value N=-1, causing an infinite value on division.

The same compiler uses the 60-bit IX instruction to increment the DO loop if a variable name is specified for the increment! So the modified program produced a moderately correct result. But how are we ever to discover bugs like this? The hardware designers did not make things easy for the compiler writers, but there is no excuse for this!

Consider the AINT function on our system, which produces the greatest floating point integer less than or equal to the floating input. The CDC compiler produces

```
UX2    X1,B2
LX2    B2
PX6    X2,0
NX6    X6
```

The left shift will normally cause a right shift because B2 is negative. But if the integer is larger than 2^{47}, it will perform as a true left shift, and the most significant digits will be lost, and the sign may be erroneous as well. Again, the machine designers did not allow for an overflow to inconveniently disturb the pipeline and the software perpetuated the folly.

This has been improved on in the RUNW compiler. An unnormalized zero (characteristic = 0) is produced in X0.

```
MX0    1
LX0    59    ,
```

then

```
FX2    X1+X0
NX2    X2
```

This takes care of the problem. If the integer is a small one it will be right shifted to align binary points with the zero. If it is large, the

zero will be shifted and the input unchanged, which is desired.

We have brought up the point of integer overflow to demonstrate how untidiness on the part of the machine design induces carelessness in the software and then by the users, because there is nothing to be gained by being careful. The best way to handle over/underflow is to not lose information, and this can only be done if the registers contain more bits than can be stored. Second best is to interrupt the machine if an overflow is going to occur which would cause information to be lost. See Kahan's SSD #159. We are going to see that there was less trouble on the 7094 from overflow than the 6400, even though it had only a tenth the magnitude range!

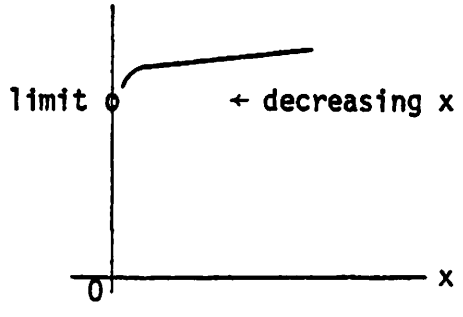
Exercise: What should be done on the CDC 6400 to compare two 60-bit integers to find the larger?

5. SOFTWARE CONSPIRACY AND THE COST OF ANOMALIES

We have seen what hardware flows alone can do. Let us now consider a case of inept hardware and software conspiring against the user. This particular case occurred on the IBM 7090, but something like it could happen on the 6400.

What Does LOG Have To Do With Differential Equations?

A graduate student had developed a marvelous idea for boundary layer control on wings of short take-off planes. He thought lift should be enhanced by his idea, and had set up the appropriate differential equations to check his ideas. Although he couldn't solve them analytically, he had some information about how they should behave and approach a limit as the independent variable went to zero. The limit was not calculable and the approach may not be smooth (like $1+\sqrt{x} \to 1$ as $x \to 0$ or $1+\frac{1}{\ln \frac{1}{x}} \to 1$ as $x \to 0$).



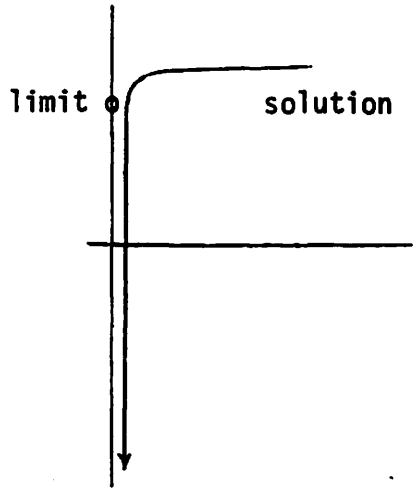
He couldn't show analytically that the limit existed so he turned to numerical methods.†

His coefficients misbehaved in a variety of ways so the standard methods were inapplicable, and besides, his job was wing design, not

†You solve a differential equation by breaking the space up into discrete little chunks, and consider functions with a slope in those intervals. The graph is replaced by a sequence of dots and this is justified if you can show that as the mesh gets smaller, the dots approach a continuous curve that is the solution.

numerical analysis.

So he did his work, but was unhappy because of the way that his solution was behaving. Rounding errors did play a role. The solution went toward $-\infty$ as the independent variable approached zero.



Clearly, something was going wrong as the vertical axis was approached.

Since he was not a numerical analyst, he did the obvious thing, and converted his program to double precision. For larger x, the solution matched perfectly, but there was still that odd behavior near zero. He decreased the mesh size, but the solution still went very far down. He knew this was not the physical solution -- to be of any use at all it had to stay positive.

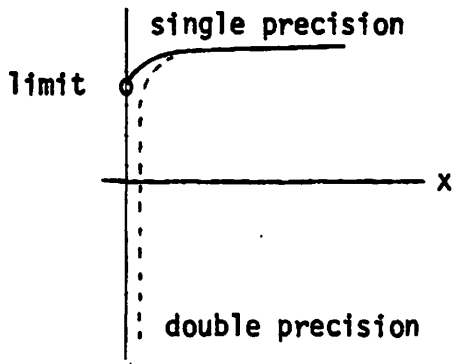
By now he had used up lots of machine time, and seemed at the end of what otherwise might have been a promising Ph.D. thesis.

At this time I [Kahan] was trying to debug a logarithm subroutine used to calculate A**B by taking exp(B*A log(A)). For some values of A and B the results were shockingly less accurate than others.

Looking over his shoulder, I saw he was using a logarithm routine and suggested he use mine, which was more accurate than the old. How much

more? My error was about .52 units in the last place and the old one's error was about 3 units and it had other interesting difficulties.

So he used my program and the single precision results reached a limit, but the double precision results continued to go down.

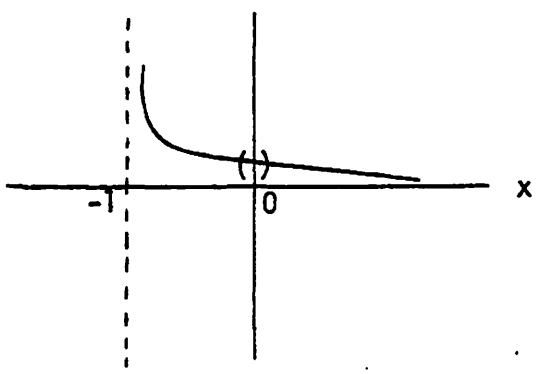


At this point I became interested. He was interested, of course; the single precision answer said he'd get his Ph.D., the double precision that he wouldn't.

He was trying to compute something like

$$f(x) = \frac{\ln(1+x)}{x}, \quad -1 < x < -10^4 \quad .$$

This function is really very well behaved, except near $x = -1$.



Strictly speaking, the point at $x = 0$ is missing. So use a power series $1 - \frac{1}{2}x + \frac{1}{3}x^2 + \cdots$ for $-1 < x-1$ in this range.

But this man had programmed so that he could transfer his program from the 7094 to a UNIVAC 1107, which he'd someday be using. They use different word lengths. So he wrote:

```
FUNCTION F(X)
IF(X .LE. -1) GO (OUT)
Y = 1.0 + X
Z = Y - 1.0
```

he's thinking

```
X =  .0000xxxxx
Y = 1.0000xxxx
Z =  .0000xx
```

so $y = 1 + z$, not $1 + x$. If x is small, I've made a rounding error and I won't take the log of $1+x$ but the log of $1 +$ (something else). Then in dividing by x, I get the wrong thing. So I'll take the log of $1 +$ something & divide by that something. And since my function is continuous and well behaved, I'll be near my point x and be on the graph of the function.

```
F = 1.0
IF(Z .EQ. 0) RETURN
F = ALOG(Y)/Z
RETURN
END
```

He settled for $F(z)$ instead of $F(x)$ since z is not far from x.

If everything had gone according to plan, he'd have only been off by half a dozen units in the last place, most attributable to rounding.

His reasoning should have been right but it wasn't. It wasn't right in single precision because of the way the log routine worked.

If you want ALOG(F), F is reduced to the range $1 \leq f \leq 2$. Then the following is computed:

$$\frac{f - \sqrt{2}}{f + \sqrt{2}}$$

But $\sqrt{2}$ is not representable by a machine number. So when $\sqrt{2}$ is off, you are in effect changing f from what it was to something else. And f is changed differently in numerator and denominator. You are calculating ALOG(F(1+e)) not ALOG(F). But he wanted to compute this for F close to 1. (1+e) is also close to 1 and so their logs are comparable. So in single precision things went wrong in a systematic way and drove his graph down. When he used my program, which didn't do this, the graph straightened out.

But he said that the double precision answer still went down, and after all, isn't double precision more accurate? In general that's true, but the double precision log function, which did logs differently and should have been more accurate, truncated in a funny way.

If x was tiny and negative, the wrong number got subtracted and z was off by about 50%. That was the hardware conspiring.

This story pretty well summarizes how things look to an engineer working on a thesis or building something, who doesn't want to understand the equipment.

How Should We Code To Prevent Conspiracy?

We see that it would be good if floating point units computed correct results and then rounded or chopped the result to fit in the word. Such schemes exact a penalty in time which is onerous to engineers who are designing to optimize a certain definition of performance. The consequence is that, from our point of view, the hardware is a set of compromises which are more difficult to describe than the simple model mentioned. If there is no guard digit of some sort, it becomes much more difficult to tell what we can compute economically. It seems likely that any computation that

can be done with a guard digit can be done without, but the designing and debugging of programs for certain computations becomes much more tedious.

Let us consider a calculation of a type common in engineering practice, that of the divided difference of a logarithm:

$$\Delta f(x_1,x_2) = \frac{\log(x_1) - \log(x_2)}{x_1 - x_2}, \quad x_1 \neq x_2$$

$$= \frac{1}{x}, \quad x = x_1 = x_2 \quad .$$

We've already seen how we can get very low significance in the computation of $\log(x_1) - \log(x_2)$ when $x_1 - x_2$ is small, even if $x_1 - x_2$ is known precisely. We would want to do this calculation differently if x_1 and x_2 agree to some number of figures, which is, however, a machine dependent computation we wish to avoid, as it may not work on the next machine it is run on.

We can rewrite the divided difference as

$$\frac{1}{x_2} \frac{\log(\frac{x_1}{x_2})}{\frac{x_1}{x_2} - 1} = \frac{1}{x_2} \phi(\frac{x_1}{x_2})$$

$$\phi(z) = \frac{\log z}{z - 1} \quad \text{if} \quad z \neq 1$$

$$= 1 \quad \text{if} \quad z = 1 \quad .$$

This code is now independent of references to significant figures. But we might wonder what happens when z is near 1. Now z will be a rounded quotient. We could have, using 4-decimal digit arithmetic,

$z = .9998$	$z - 1 = -.0002$	$\log z = -.0002$
$\frac{x}{y} = .99989998$	$\frac{x}{y} - 1 = -.0001002$	$\log \frac{x}{y} = -.0001002$

Perhaps surprisingly, we get $\phi = 1.000$ in either case! We can show rigorously that changing z by an ulp changes $\phi(z)$ by less than an ulp. So rounding $\frac{x}{y}$ will cause no problem. Here we have an example of a calculation where the intermediate results $z-1$ and $\log z$ are accurate to no significant figures yet the results are perfectly good, because the proper relation of the intermediate results was maintained.

On CDC-type machines we can run into troubles in various ways. Z may be compared to 1 and found unequal by an integer compare, and then $z-1$ will be computed to be zero by the hardware.

If $z < 1$ slightly the answer could be wrong by a factor of two, when $z-1$ is computed as $z(1+\zeta) - 1(1+\Omega)$.

However, these are merely hardware errors. Surely the software would be written to ameliorate some of the flaws, or at least not make them worse, or at least not introduce new ones! But such hopes are, alas, in vain. To compute logs, programmers often attempt to use mathematical identities such as

$$\log(z) = \log\left(\frac{1+y}{1-y}\right) + \log\sqrt{\frac{1}{2}}, \quad y = \frac{z-\sqrt{\frac{1}{2}}}{z+\sqrt{\frac{1}{2}}} .$$

When $\frac{1}{2} \le z \le 1$ the power series expansion of this form converges rapidly. Unfortunately $\sqrt{\frac{1}{2}}$ is not precisely representable, so that we actually compute

$$y = \frac{z-\sqrt{\frac{1}{2}}-\varepsilon}{z+\sqrt{\frac{1}{2}}+\varepsilon} = \frac{(z+\zeta)-\sqrt{\frac{1}{2}}}{(z+\zeta)+\sqrt{\frac{1}{2}}}$$

where

$$\zeta = \frac{-z}{\varepsilon + \sqrt{\frac{1}{2}}}\,\varepsilon \doteq -\sqrt{2}\,z\varepsilon \quad .$$

Instead of $(1+\lambda)\log z$, we get $(1+\lambda)\log(z+\zeta)$. When z is near one, $\phi(z)$ may not be computed accurately because the logarithm may not be very good. If, in addition, the hardware computes $z-1$ inaccurately, the results are totally unpredictable.

The sane thing to do is to compute $y = \frac{z-1}{z+1}$ if $z > \sqrt{\frac{1}{2}}$ and $y = \frac{z-\frac{1}{2}}{z+\frac{1}{2}}$ if $z < \sqrt{\frac{1}{2}}$. This takes a very slight additional effort on the part of the programmer but it saves the user from having to worry about details of the logarithm routine.

When designers optimize the speed without consideration of error, the user may sometimes get wrong results for no apparent reason or he may conclude that calculations such as extended summation can't be performed. The user is tempted to prove theorems such as Viten'ko's [see 10] which seem to be relevant to the hardware but really are not.

Regardless of how the hardware is designed, there will probably always be tricks such as the cubic equation algorithm [see 17]. However, if we design the hardware and software properly, it should not be necessary for ordinary users to get degrees in numerical analysis in order to find such tricks to solve their everyday problems.

Economic Cost of Anomalies

I don't object to the funny things machines do because they are so wrong that they make life not worth living. We can obviously code around them if we know about them.

I don't object because they violate mathematical aesthetics.

But I do object because these little flaws have an economic consequence out of all proportion to the cost of extirpating them. And the economic consequences are hard to uncover.

For example, the man above might have said that wing won't work and gotten a thesis in something else; or he might not have.

He was lucky, because the numerical calculation did not, in the end, deflect him from his project.

It is very unlikely that a rounding error in a floating point operation would cause a bridge to collapse, because people usually don't trust computer results. They build prototypes and throw in fudge factors.†

I don't know of any collapses caused by rounding errors, and I'd be as unlikely to know as the man who did it. How would you find out about it?

I don't know how often people have tried to simulate a difficult idea and because of rounding errors, given up on the idea. Probably very often. I've heard people discussing programs and methods used that wouldn't give correct answers; they wouldn't be off by orders of magnitude, just by factors like 1.325. For example, in differential equation solvers where they don't know what step size to use, they used a fixed size -- which is too big in one part and too small in another. But these solvers are imbedded in the program and are never noticed.

---

†There is one example of a bridge collapsing because of small (not rounding) errors, the Quebec bridge in Victorian times. The designer neglected the deflection of the sections under their own weight while the bridge was being constructed.

The side sections sagged before the center suspension section was added. It collapsed.

The only aircraft I know of that crashed because of a computer program is the Lockheed Electra. There it was not a rounding error but a mistake in the organization of appropriate subroutines.

Then there's a man, say a psychologist, who doesn't understand how that electronic stuff works, who is trying to debug a program, a fairly simple one of less than 100 statements. You give him a list of all the funny little things the computer does and he spends hours trying to find which one causes his bug. But of course his error was in a format statement.

So you see that the costs I'm enumerating are real economic costs. They are costs to people and to firms. And they have nothing to do with a rounding error committed in somebody's program, that he may not have anticipated. It might be that he now has extra things to look for. He has to fight the machine instead of getting help from it.

That's why I'm opposed to what happens on the 6400.

Question: I had worked on a numerical subroutine for solving differential equations. People using it would typically choose a step size and then one 10 times smaller to see if that changed the results. But the program kept blowing up mysteriously. It appeared that when funny things happened, they were really funny. The error propagated in rather impressive ways.

Answer: You're saying that errors will be accompanied by symptoms so obvious that one could hardly fail to notice them if he was at all conscientious. In response to the claim, made by the author of a matrix equation solver, that anyone's matrix that wasn't solved by his routine was so unlucky that he'd already been run over by a truck, I found a 2×2 matrix which, when put into the iterative solver gave what looked like a correct solution (all tests were satisfied), but not even the leading digits in that solution were correct. I don't think people know when an error is committed.

There are times when in treating continuous functions the intermediate results may be discontinuous. And these discontinuities may be important and you'd like to be told about them. So you depend on laws of arithmetic

that may not be honored by your machine. There are Fortran programs that run on a 7094, 7090, B5500 and GE 645, but they will not run on a 6600 and no one has found out why.

There may be a law of diminishing returns in hunting for these funny errors. If we can come up with a rationale for dealing with large classes of these errors and if this thinking isn't too devious or subtle, you'd hope others would come across that rationale and thereby avoid the errors.

The cost of weeding out these errors is negligible compared to the cost of the whole machine. You may end up with a machine that is better than it has to be, but not much better.

6. EXECUTION-TIME ERRORS

We leave the CDC 6400 now to discuss more reasonable methods of dealing with occurrences such as overflow. We will refer to the Toronto system for the 7094 described in Kahan's SHARE Secretarial Distribution #159 (1966).

We can distinguish between scheduled and unscheduled errors. A negative input to a square root routine which makes some provision for negative inputs is a scheduled error. Such errors are a matter to be decided between the user's program and the square root routine. An unscheduled error is one that occurs when no explicit provision has been made for dealing with it. Divisions by zero in many programs are unscheduled. A linear equation solver may set a flag if the system is too nearly singular. Scheduled errors happen to users who check this flag. Unscheduled errors happen to users who don't.

One would like to specify options for errors. In the square root case, the most useful output for a negative number might be $-\sqrt{|x|}$ for some users, $\sqrt{|x|}$, 0, or job abort for others. Some users want to know how the negative input came about. They would like to know the statement number and Fortran subroutine name where the negative square root was attempted, and the nest of calling routines, if any. Others expect an occasional negative from rounding errors, and don't care which small number is output as long as they aren't kicked off and their output is not blemished by an error trace. How many options should we offer? We could go to the extreme of a PL1 ON condition. This is generally too expensive. Error options are part of the environment in which subroutines are executed and would have to be saved and restored on every call and return. For instance, a user may specify that certain action is to be taken on an overflow, and later call a quadratic solver. The quadratic solver may generate overflows

in intermediate results with which it should not bother the user. But if the final answer deserves to be overflowed the user-specified action should occur. Therefore the options should be so simple that the subroutine should be able to determine what the user has specified and act accordingly.

Suppose then that, in the square root routine, the user may specify either $-\sqrt{|x|}$ for a negative x or be kicked off the system.

IF(KICKED(OFF))

Then we should arrange for kick-off to be less of a disaster than it commonly is. At Toronto there was the IF(KICKED(OFF)) statement. KICKED was a subroutine which returned a logical value FALSE. OFF was a parameter which was printed out on kick-off to identify the kick-off routine to the user.

During normal execution the action of the KICKED routine was to maintain a pointer to the conditional part of the last kicked-off statement executed. The conditional part of the statement was not executed because KICKED returned a value FALSE. When an error warranting a kick-off occurred, buffers were flushed, normal diagnostics were provided, and control was transferred via the pointer to the conditional part of the statement and a STOP was written over the next following statement:

```
        Before
3   IF(KICKED(3)) WRITE Save Tapes!
    NEXT STATEMENT
    X=SQRT(-3)

        After
3   IF(KICKED(3)) WRITE Save Tapes!
    STOP
    X=SQRT(-3)
```

Then execution could continue until another kick-off occurred, or the post-kick allowance of, say, 10 seconds and 300 lines, was exceeded. One could use the conditional part of the statement in any usual way, for printing out the values of key variables, issuing operator instructions, etc. Thus it should be possible to save what is necessary to restart a program that, for instance, had simply run out of time. The KICKED routine cost less execution time than a divide and could therefore be used quite freely. No change to the compiler was necessary.

What If They Don't Want to Kick Off?

With kick-off now less of a disaster, there remained the choice of the other options. 1./0. was treated as an overflow, but 1/0, 0.0/0.0, and 0/0 were always a kick-off. More elaborate options existed for overflow. The default silent option was to set the result to the number of the same sign largest in magnitude. A system flag would be set which could be turned off by testing. If no test occurred the system would print

LAST UNREQUITED OVERFLOW WAS (location) .

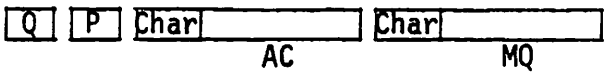
at the end of the job. A similar message was available for underflow. A logical extension to this system would be to include a message

FIRST UNREQUITED OVERFLOW WAS ... .

The cost of the system is negligible.

In the printing mode the same events occurred and, in addition, every overflow or underflow generated an immediate message with details as to cause and location. The user could also arrange for automatically switching to silent mode after printing a certain number of messages.

Underflow on the 7094 was complicated by "spurious" underflows. The accumulator AC and its extension MQ both had characteristics, and MQ could underflow. It would be reasonable to set MQ to zero only when software double precision was being done, which was a holdover from the 7090 made obsolete by the double precision hardware on the 7094. Consequently a warning was issued not to use software double precision and MQ underflows were ignored.



Another possibility of underflow was in the remainder following a single precision divide. There was, however, no way to use the remainder in Fortran. In other code the P and Q bits of AC could be thought of as a leftward extension of the characteristic of AC. Consequently this underflow was also ignored.

Most people don't want to be bothered with underflow messages and are satisfied to set the number to zero and continue. Rather than do something wrong without recording the fact, the unnormalized mode of treating underflow was developed. Unnormalized numbers have the smallest possible characteristic and unnormalized integer parts. Underflowed numbers are treated as unnormalized when possible, so that there is quite a range of very small numbers with gradually fewer significant digits. The assumption is that if it is all right to have underflows set to zero, then it is just as good to set them to small numbers. In the unnormalized mode no UNREQUITED UNDERFLOW message is produced. However, the unnormalized numbers were quite persistent, and, if the user attempted to divide by one of them, he would usually get kicked off. The existence of unnormalized numbers at the end of the

calculation was a signal to the user that he had not been correct in assuming that his underflows could be safely set to zero.

An application of this technique was in computing scalar products $\Sigma a_i b_i$. By doing the multiplication and addition with underflows treated in unnormalized mode the accuracy of the result could be easily ascertained by checking if it was normalized. If so, it is as accurate as it deserves to be; if not, it had underflowed, but the test need be made only once, at the end of the loop.

The effect of unnormalized mode was to soften the impact of underflow. The calculation discussed previously which yields about 3 on the 6400 would yield nearly the correct result of 1 on the 7094 in unnormalized mode, and the answer would probably have been unnormalized as a warning.

There are some calculations in which overflows occur inevitably, as in symbolic evaluation of large determinants. Counting mode was invented to allow a limited range of these computations. The user would designate a cell for counting overflows. Then every time certain operations occurred, that cell would be incremented if overflow occurred, and decremented if underflow occurred. For instance, to compute $\Pi \frac{(a_j+b_j)}{(x_j+y_j)}$, the denominator would be computed, and the cell incremented each time an overflow occurred on an add or multiply. The cell would be reversed in sign and the numerator computed. At the end of the calculation the counting cell would indicate the power of 2^{256} which should be applied to the number actually in storage. With no testing in inner loops, this technique costs the user only if an overflow actually occurs. Most of the computations in counting mode never actually overflowed or underflowed, but the counting mode made it possible to allow for the possibility in a rational manner without jeopardizing the entire calculation.

The 7094 hardware facilitated a reasonable treatment by interrupting before information was lost. The correct result could always be inferred because the extra information was always preserved in the left bits. Further, the overflow/underflow interrupt had priority over all others, so that this information was not lost. Because the system was written with a flexible set of options, no users ever found it necessary to supply their own overflow/underflow handling routines.

7. A PROOF OF A NUMERICAL PROGRAM

In a previous lecture [1] we described an algorithm for solving the quadratic equation $ax^2 - 2bx + c$. Today we shall write a program for such an algorithm and see what can be proved about the output of the program. We will assume that every arithmetic operation, including square root, is computed correctly and then chopped or rounded with at most an error of one unit in the last place, although slightly weaker assumptions would suffice. For our purposes today we shall ignore overflow and underflow.

Recall our algorithm:

$$d = b^2 - ac$$

$$\text{if } (d > 0) \quad x_{BIG} = \frac{b + \text{sign}(\sqrt{d}, b)}{a}$$

$$x_{LIT} = \frac{c}{a x_{BIG}}$$

$$\text{if } (d \leq 0) \quad x_{\pm} = \frac{b}{a} \pm i\frac{c}{a}\sqrt{-d}$$

We code it as follows:

```
      D = B**2-A*C
      IF (D.LE.0.0) GO TO 1
C     real distinct roots RP, RM
      S = B+ SIGN(SQRT(D),B)
      RP = S/A
      RM = C/S
      GO TO ...
C     complex or coincident RR ± √-TRI
1     RR = B/A
      RI = SQRT(-D)/A
```

For analysis purpose we introduce Greek letters after each operation. Each Greek letter is bounded by the maximum relative error due to chopping or rounding. In the example of four digit arithmetic the bound would be

$$-\frac{1000 - 1000.5}{1000} = 5 \times 10^{-3} \quad .$$

Lower-case Latin letters represent values stored in cells with corresponding upper-case names.

$$d = (b^2(1+\mu_1) - ac(1+\mu_2))(1+\sigma)$$

CDC arithmetic does not fit this mold. We would have to write $d = (b^2(1+\mu_1)(1+\sigma_1) - ac(1+\mu_2)(1+\sigma_2))$ but this does not affect the present analysis.

$d > 0$ real root

$$s = (|b| + (1+\rho)\sqrt{d})(1+\alpha)\cdot sgn(b)$$
$$r_+ = (1+\delta_1)s/a$$
$$r_- = (1+\delta_2)c/s$$

$d \leq 0$ complex or coincident

$$r_R = (1+\delta_3)b/a$$
$$r_I = (1+\delta_4)(1+\rho)\sqrt{-d}/a \quad .$$

Let us investigate how these errors affect the results. Suppose N is a large integer, say $2^{24}+1$ on the 6400. Then try to solve the equation

$$(N+1)x^2 - 2Nx + (N-1) = 0$$

whose roots are 1 and $\frac{N-1}{N+1}$. Suppose we make no other rounding errors than the following:

$$b^2 = N^2(1+\mu_1)$$
$$ac = (N^2-1)(1+\mu_2) \quad .$$

In this case it would be quite possible to obtain b^2 and ac rounded to the same value. $N^2 = 2^{48}+2^{25}+1$, and $N^2-1 = 2^{48}+2^{25}$. N^2 would probably be rounded to N^2-1, with an admittedly small error. However, now $d = 0$ and if no other errors are made the computed roots equal

$\frac{N}{N+1} = 1 - \frac{1}{N+1} \doteq 1 - 2^{-24}$. These "roots" are equidistant from the precise roots 1 and $1 - 2 \cdot 2^{-24}$. These computed roots differ from the true roots by about 2^{24} units in the last place! Clearly we can't make the claim that our program delivers the roots of the given quadratic correct to within a few units in the last place.

Now let us see what equation we did compute the roots of. This is

$$(N+1)x^2 - 2Nx + (\frac{N^2}{N+1}) \quad .$$

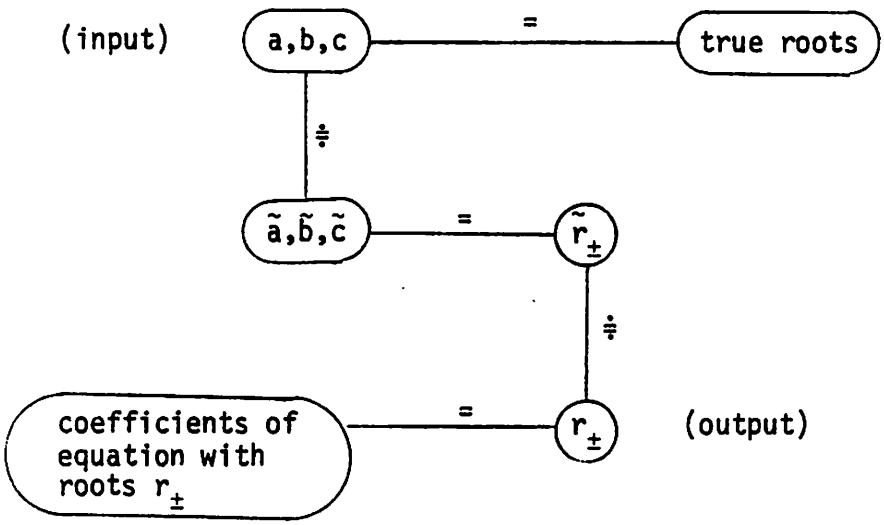
We see that the relative difference in the coefficient c is $1 + \frac{1}{N^2-1}$. That is, the computed roots were the correct roots of an equation whose coefficients differ from the original ones by less than one unit in the last place. Unfortunately, this statement also is not true for our program in general.

Consider any equation such as

$$x^2 - 2 \cdot 10^{-50} x - 1 = 0 \quad .$$

On any normal machine we compute the roots to be ± 1, because $10^{-100} + 1 \doteq 1$. These are the correct roots of the equation $x^2 - 1 = 0$. Clearly the coefficient b of this latter equation differs by a substantial relative amount from 10^{-50}!

Fortunately, we can say something definite. The roots given by our program differ by a few units in the last place from the true roots of a quadratic whose coefficients differ from those input by at most a few units in the last place. Let a, b, c represent the original coefficients and $r_\pm$ the roots delivered by the program. Then there are $\tilde{r}_\pm$ which are the precise roots of a quadratic with coefficients $\tilde{a}, \tilde{b}, \tilde{c}$, and in each case the $\tilde{}$ perturbation is a few units in the last place. In a picture:



Note that, in general, the choice of the intermediate quadratic is not unique.

Let us analyze in detail the simple case $\tilde{a} = a$, $\tilde{b} = b$, and $\tilde{c} = (\frac{1+\mu_2}{1+\mu_1})c$. Then if $d \le 0$, $\tilde{r}_\pm = \tilde{r}_R \pm i\tilde{r}_I$, $r_R = (1+\delta_3)\tilde{r}_R$, $r_I = (1+\delta_4)(1+\rho)\sqrt{(1+\mu_1)(1+\mu_2)}\tilde{r}_I$. Now when $d > 0$, define

$$\theta = \left(\frac{s}{(1+\alpha)\mathrm{sgn}(\tilde{b})}\right)/(|\tilde{b}| + \sqrt{\tilde{b}^2 - \tilde{a}\tilde{c}}) - 1$$

$$= \frac{\rho\sqrt{(1+\mu_1)(1+\sigma)} + (\mu_1+\sigma+\mu_1\sigma)/(1+\sqrt{(1+\mu_1)(1+\sigma)})}{1 + |b|\sqrt{(1+\mu_1)(1+\sigma)/d}}$$

$$\doteq (\rho + \tfrac{1}{2}\mu_1 + \tfrac{1}{2}\sigma)/(1 + |b|/\sqrt{d}) \quad .$$

Thus θ is of the order of a few units in the last place. With this definition

$$r_+ = (1+\theta)(1+\alpha)(1+\delta_1)\tilde{r}_+ \ , \quad r_- = \tilde{r}_-(1+\delta_2)(1+\mu_1)/((1+\theta)(1+\alpha)(1+\mu_2)) \quad .$$

That is, if we follow through the algorithm we do find roots that differ from the correct roots of the altered input by a few units in the last place.

We now know that the roots are approximately those of an altered quadratic. But how close are they to the roots of the original quadratic? Clearly it's the change in c that can make our results bad.

Let us now look at the problem from the point of view of perturbation analysis. How much could the roots possibly be expected to vary if we vary the input coefficient c? In particular, compare the equations

$$x^2 - 2bx + c \qquad \text{with roots } R_\pm \ ,$$
$$x^2 - 2bx + c(1+\gamma) \qquad \text{with roots } \tilde{r}_\pm \ .$$

Theorem. $\left|1 - \frac{\tilde{r}_\pm}{R_\pm}\right| \le \sqrt{|\gamma|}(\sqrt{|\gamma|} + \sqrt{1+|\gamma|})$.

For small γ, the relative difference is about $\sqrt{|\gamma|}$ at worst.

Proof. Let $\delta_\pm = 1 - \frac{\tilde{r}_\pm}{R_\pm}$. Then $\tilde{r}_\pm = (1-\delta_\pm)R_\pm$.

Using the facts $\tilde{r}_+ + \tilde{r}_- = 2b = R_+ + R_-$, $\tilde{r}_+ \cdot \tilde{r}_- = c(1+\gamma)$, and $R_+ \cdot R_- = c$, we deduce that $R_+\delta_+ + R_-\delta_- = 0$ and $(1-\delta_+)(1-\delta_-) = 1+\gamma$. Let $z = \frac{R_-}{R_+}$.

Without loss of generality let $|R_-| \le |R_+|$ so $|z| \le 1$. Then $|\delta_+| = |-z\delta_-| \le |\delta_-|$ so we only concern ourselves with δ_-. It satisfies

$$(1+z\delta_-)(1-\delta_-) = 1+\gamma$$
$$\text{or} \quad z\delta_-^2 - (z-1)\delta_- + \gamma = 0 \ .$$

The two roots for δ_- correspond to the roots $\tilde{r}_\pm$; we take that corresponding to the root δ_- smaller in magnitude.

Our problem may now be stated as follows:

Given $|\gamma| > 0$, $|z| \le 1$, $|\tau|^2 \le |\gamma|$ determine $\max|\delta(z,\tau)|$ where $\delta(z,\tau)$ is the smaller root of

$$z\delta^2 - (z-1)\delta + \tau^2 = 0 \ . \qquad (*)$$

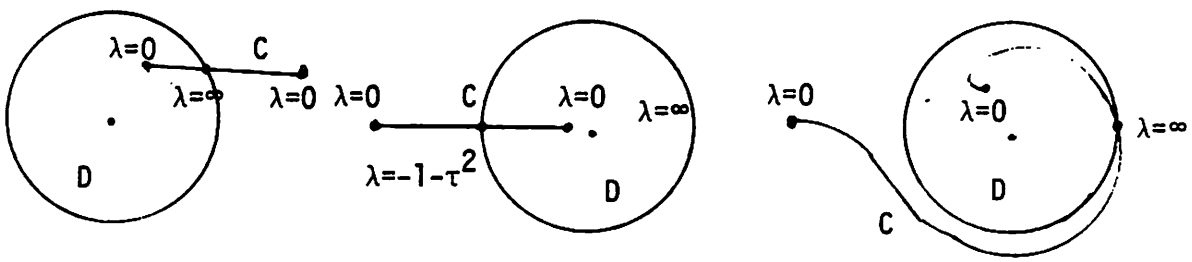
$\delta(z,\tau)$ is a holomorphic function of z except for those critical values where both roots of (*) are equal in magnitude. The critical values are those satisfying

$$\lambda \equiv \frac{4z\tau^2}{(z-1)^2} - 1 \geq 0 \quad .$$

These z lie on a curved arc C traced by

$$z_{\pm} = \frac{(\tau \pm (1+\lambda+\tau^2)^{\frac{1}{2}})^2}{1+\lambda}, \quad \lambda \geq 0 \quad .$$

Since $z_+ = \frac{1}{z_-}$, part of C must lie in the disk $|z| \leq 1$. Define D to be the domain obtained from the disk by cutting along C. Here are some examples.



C is the arc. $\tau^2 < -1$ C is the fallen question mark

C is the circumference plus the line segment

Generally, either $\{z: |z| = 1\} \subset C$ or C intersects the circumference $|z| = 1$ in just one point, at $z = 1$, $\lambda = +\infty$.

Certainly $\delta(z,\tau)$ is a holomorphic function of z inside D and continuous as z approaches the boundary of D. Therefore the maximum modulus theorem applies (Titchmarsh, Theory of Functions, pp. 166-168) so that the maximum of $|\delta|$ on D or on $|z| \leq 1$ occurs on the boundary of D.

If the maximum is achieved when $|z| = 1$ then $|\delta|^2 \le |\tau|^2$ with equality when $z = 1$. If achieved on C inside $|z| < 1$, $|\delta|^2 \le \frac{|\tau|^2}{|z|}$, because the product of the roots is $\frac{\tau^2}{z}$. Hence $|\delta|^2 \le |\tau|^2 \max_C |\frac{1}{z}|$.

But on C,

$$|\frac{1}{z}| = \frac{|\tau \pm (1+\lambda+\tau^2)^{1/2}|^2}{1+\lambda} \le \frac{(|\tau| + \sqrt{1+\lambda+|\tau|^2})^2}{1+\lambda} = (\xi + \sqrt{1+\xi^2})^2$$

$$\text{where } \xi \equiv \frac{|\tau|}{\sqrt{1+\lambda}} \le |\tau| \quad .$$

$$|\frac{1}{z}| \le (|\tau| + \sqrt{1+|\tau|^2})^2 \le (\sqrt{|\gamma|} + \sqrt{1+|\gamma|})^2 \quad ,$$

so $|\delta|^2 \le |\gamma|(\sqrt{|\gamma|} + \sqrt{1+|\gamma|})^2$, with equality when $\gamma \ge 0$ and $z = (\sqrt{\gamma} + \sqrt{1+\gamma})^{-2}$. (Then $\tilde{r}_+ = \tilde{r}_-$.) So the perturbation of γ is bounded by

$$\left|1 - \frac{\tilde{r}_\pm}{R_\pm}\right| \le |\delta| \le \sqrt{|\gamma|}(\sqrt{|\gamma|} + \sqrt{1+|\gamma|}) \quad . \qquad \text{Q.E.D.}$$

This bound is certainly the best possible, independent of the data, and it is achieved when the roots are nearly equal. In this case the discriminant is very small and inaccurate because cancellation has revealed previous rounding errors. We could get a much better bound in many cases through a more detailed analysis of the bound as a function of the coefficients a, b, c. To be useful we would have to incorporate a possibly lengthy computation of the bound into the quadratic routine. The user could then call upon this part of the routine if he wanted to know how good his roots are. Fortunately, as we shall see, there is a programming trick which we can exploit in this particular problem so that the user need not perform an error analysis, and we need not compute a complicated bound, because we will be able to show that an acceptable fixed bound now applies.

Analysis of Round-off for the Quadratic Equation Solver

We have seen a program to solve the quadratic equation that is good in the sense of delivering very nearly the correct answer to a problem that is very nearly that which we wished to solve. We have also seen that in the worst case a small relative perturbation γ in the coefficient c can cause a much larger relative perturbation $\sqrt{|\gamma|}$ in the roots. How can we get rid of this complication? The one critical computation is that of the discriminant:

$$d = (b^2(1+\mu_1) - ac(1+\mu_2))(1+\sigma) \quad .$$

The relative precision of the entire calculation can be as bad as the relative precision of the discriminant. When $b^2 \doteq ac$ we can't, in general, write

$$d = (b^2-ac)(1+\delta)$$

for small δ. What happens if we have double precision available? The product of single precision numbers is precisely representable in double precision. Further, if the double precision subtract rounds after normalization, then $\mu_1 = \mu_2 = 0$. Then we can write a program that will deliver nearly the roots of the given equation! However, along the way certain difficulties arise. For instance, we might try

```
DOUBLE PRECISION DD
DD = B*B
D = DD-A*C    .
```

We would hope then that DD would be a double precision number holding the product b^2 precisely, and D would be the double precision difference

rounded to single precision. For early IBM Fortran implementations this was actually done. The compilers checked the context before discarding the second half of the doubly precise product of single precision numbers. More recently the previous code would be compiled so that the second half of the doubly precise product is discarded without checking the context first. This procedure is now built into the syntax of the language. One way of dealing with such compilers is to write

```
DD = DBLE(B)**2
D = DD - DBLE(A)*DBLE(C)    .
```

Now the program appends zeros to A, B and C, and then does full double-precision multiplication to yield double precision products. Most of this work is not necessary for our purpose and in fact consumes a great deal of time: with software double precision, the cost of the three double precision operations will far outweigh all the other operations in the program, except the square root. A similar waste becomes critical in, for instance, scalar products of vectors. If single precision is used the error in the sum $\sum_{j=1}^{N} x_j y_j (1+\varepsilon_j)$ will be such that the computed sum will be the product of vectors, one of which may have perturbations as large as N units in the last place of each element. If we do double adds on the double products we would get $\sum_{j=1}^{N} x_j y_j (1+\varepsilon_j^2)$. The result then would be the product of vectors perturbed by N units in the double precision last place, which is ignorable. Consider the following results on inverting a 100×100 matrix on a 7094 with hardware double precision:

Arithmetic	Time	Backward Error Bound
Single Precision	7.5 seconds	100 units in last place per element
Double Precision only for accumulation of scalar products	11	2 ulps
Double Precision Throughout	15	800 units in last place of double precision

Note that using double precision throughout requires twice the storage space for the matrix, a matter of 10,000 cells on a 32K machine in this example. On the 360 the hardware is available but is not useable in Fortran. 360 short word arithmetic only carries six hexadecimal digits. Loss of 100 units in the last place means losing 2 of the 6 figures of accuracy. We would like to use short word arithmetic to conserve storage, but a mistaken principle in the compiler forces us to use double precision to get good single precision results.

Ideally the compiler should never lose information before consulting the context. We would like to have some means of specification such as

```
D = DSIC(B*B) - DSIC(A*C)
```

which means: treat the partial results as double precision until the assignment to D is made, when type conversion to single must occur. It doesn't matter so much which default rules the compiler may follow, as long as there is at least an option to do what we want. Explicit type conversions are done in many other contexts, why not this one?

We are almost to the point of writing

$$d = (b^2-ac)(1+\delta)$$

when a new problem is discovered. Most machines don't carry a guard digit

for double precision subtraction. The double precision instruction in the 7094 and CDC machines does not give the desired result but returns us to the situation

$$d = (b^2(1+\mu_1) - ac(1+\mu_2))(1+\sigma) \quad .$$

At least μ_1 and μ_2 are now a few units in double precision. This means that $\sqrt{|\gamma|}$ is a few units in single precision. We are now able to announce, perhaps, that our roots are correct to a few units in the last place (ulps).

When we make such an announcement it will be interpreted as meaning that the results are real if the roots are real, and complex if the roots are complex, and that each number printed is correct to within a few ulps. Recall, however, that our perturbation analysis was concerned only with the magnitude of complex numbers. Nothing was said about the real and complex components. Indeed, there is no way of showing, using our usual error analysis based on bounds on μ_1 and μ_2, that the complex part will be computed correctly to a few ulps, or even that the discriminant will not change signs due to errors.

Yet our program will run correctly on nearly every reasonable machine. The only way to understand this is by a rather devious line of reasoning. First we shall show that real roots will be computed essentially correctly.

Real Parts Remain Real

Suppose then that $b^2 = ac$. Then $b^2 - ac = 0$ because the subtraction should be performed precisely. Now suppose that $b^2 > ac$. Since we expect the arithmetic unit in a reasonable machine to be monotonic, we will find that computed $(b^2-ac) \geq$ computed $(ac-ac) = 0$. On such a machine there is, therefore, no possibility that a pair of real roots will be represented

as complex. Further, great relative error in b^2-ac occurs only when this quantity is nearly zero and therefore will not seriously affect the accuracy of $|b| + \sqrt{b^2-ac}$, so that the real roots will be nearly accurate.

Now consider complex roots, $b^2 < ac$. By monotonicity again, computed $(b^2-ac) \leq 0$. Therefore the real part will always be computed independent of the discriminant and will be computable essentially correctly.

Accuracy of Complex Parts

It is still unclear whether the complex parts are correct to a few ulps. Suppose $b^2 < ac$ but they have the same characteristic. Then the subtraction is performed precisely, and the complex part is OK. The worst situation is when $b^2 < ac$ but computed $(b^2-ac) = 0$, for then the complex part has vanished with great relative error. This situation would have to occur in a shift. Consider a right shift of one.

$$\text{ac: } 2^m \times \boxed{1000\cdots\cdots0000}$$
$$b^2\text{: } 2^{m-1} \times \boxed{1111\cdots\cdots1111}$$

b^2 can <u>not</u> be all ones. Remember, it was formed from a single precision number. A long string of ones is just less than a power of two. One way of getting it would be to square a number just less than a power of two, b = 111...11. Then we would have:

ac:	2^m	1.....00	0.....0	
b^2:		011.....11	0.....0	1 ← shifted off
$ac-b^2$		0.....01	0.....0	

The difference would <u>not</u> become zero, barring underflow, and in fact would be rather accurate. The other possibility is that b is just less than

the square roots of an odd power of two. Now a number just greater than $\sqrt{1/2}$ would produce a square containing a power of two plus some other things. A number just less than $\sqrt{1/2}$ would square to a number containing one less significant bit than the first, and would therefore be left shifted one bit in normalization, causing a zero to be inserted on the right. In the subtraction $ac-b^2$ the right shift would simply shift out a zero, so no information would be lost and the difference would be correct. Consequently we can conclude that if $b^2 \neq ac$ then the difference will be computed correctly to single precision, even without a guard digit. There are certain machines in which double precision is done in software. These machines sometimes lose two digits instead of one in the right shift, and the previous analysis may not be valid.

The peculiar analyses are necessary since one might be disinclined to believe that a quadratic solver could give the imaginary part of complex roots correct to a few ulps, based on a certain model of arithmetic. This model is not categorical, so in particular places we must invent special analyses to understand what is happening. We shall see in the next lecture that loopholes in our rules about rounding will allow us to perform calculations that are otherwise provably impossible. The reason poorly designed arithmetic is bad is not that the error is slightly larger, but that it is so often uncertain, in the sense that we must either expend substantial energy in detailed investigations of the type we did here or in tedious programming around the uncertainties, which increases the likelihood of an error and consequent cost of the final program.

8. MODIFYING THE QUADRATIC EQUATION SOLVER TO AVOID UNNECESSARY OVERFLOW AND UNDERFLOW

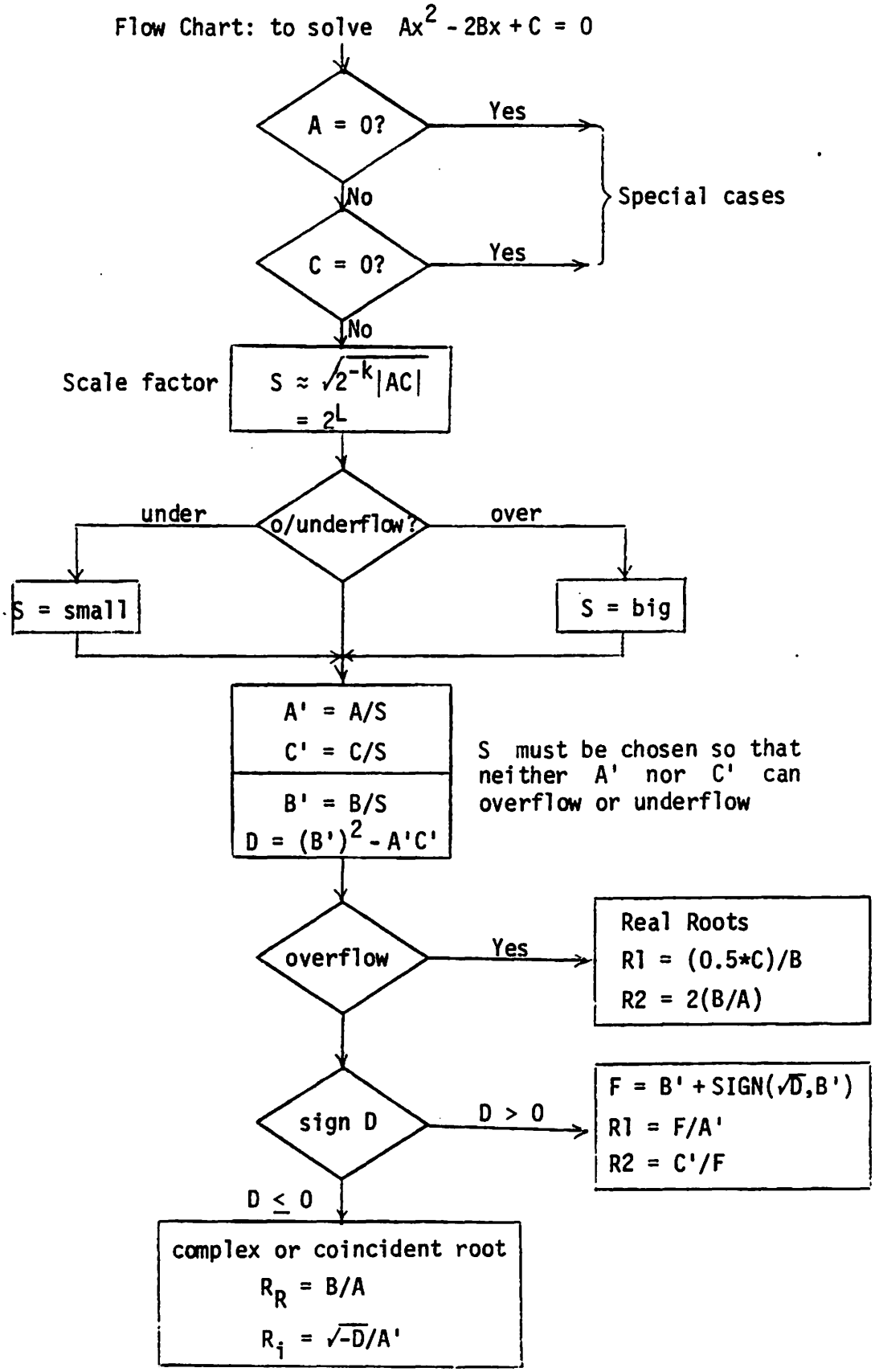
Now we will discuss how to cope with over/underflow in solving the quadratic equation

$$Ax^2 - 2Bx + C = 0 \quad .$$

The flow chart will not indicate where double precision is needed for the accuracy we want, as that aspect of the problem has been covered already.

The object here is to write relatively simple code to handle over/underflow, for most machines. Some choices indicated in the flow chart will have to be discussed with care, to see if choices can be made that follow the desired restraints. For example, can an appropriate scale factor S be found so that A/S and C/S will never overflow or underflow?

Can overflow and underflow at intermediate steps be handled adequately? On the CDC to suppress abortion upon using an infinite operand requires a control card. You cannot revert to the normal mode at some later stage in the computation. If you operate in the normal mode, then you could not take advantage of someone's program that handled over/underflow so nicely that you could almost imagine that it hadn't happened. In this quadratic solver, the writer might allow the program to handle overflow, but a user may want to be kicked off when that happens. Then you'd have to be careful never to use quantities which may have been set to infinity or indefinite; you'd have to do a large number of tests. Not all the tests will be indicated on the flow chart, but you'll be able to see where they should go.


Flow Chart: to solve $Ax^2 - 2Bx + C = 0$
A = 0?
Yes
No
Special cases
C = 0?
Yes
No
Scale factor
$S \approx \sqrt{2^{-k}|AC|}$
$= 2^L$
o/underflow?
under
over
S = small
S = big
A' = A/S
C' = C/S
B' = B/S
$D = (B')^2 - A'C'$
S must be chosen so that neither A' nor C' can overflow or underflow
overflow
Yes
Real Roots
R1 = (0.5*C)/B
R2 = 2(B/A)
sign D
D > 0
$F = B' + SIGN(\sqrt{D}, B')$
R1 = F/A'
R2 = C'/F
$D \leq 0$
complex or coincident root
$R_R = B/A$
$R_i = \sqrt{-D}/A'$

Special Cases A = 0 or C = 0

There is a difficult question in what is meant by A = 0 on the CDC. A might be one of those tiny numbers that looks like zero to the multiply/divide box but not to the add box. When you decide 'is A = 0', you're bound to disappoint somebody. You have to adopt a convention and stick to it.

Question: Wouldn't you look at the program and see if you were going to use A in multiplications or additions, to make your decision?

Answer: Yes, that is one rationale. But remember that that is of almost no consequence to the man who will use your program to solve a quadratic and who doesn't care how it is done. A coefficient that is zero to you may very well not be zero to him.†

My feeling now is that numbers that the multiply unit considers zero should be set to zero. So instead of writing IF(A .EQ. 0), I would write IF(1.0*A .EQ. 0).

Scale Factor S

Having ascertained that neither A nor C is zero to the multiply unit, we can compute S. We will see later what value K must have, but for now simply note that S is roughly the geometric mean of |AC|. The actual value chosen for S requires care. In the attempt to evaluate S, over/underflow may occur, but that actually is not serious. The object is to scale the whole equation by dividing through by S in order to insure that the new A*C is a modest number close enough to 1 so that if the new $(B)^2$ overflows, we know that AC is negligible compared to B^2. On our machine, AC could be $\sim 10^{\pm 50}$ and that would be close enough.

†It is a design flaw of the CDC that the multiply unit will allow you to generate a number with a zero characteristic and non-zero integer part (by successive divisions by 2), but then the unit will not accept that number as an operand.

An over/underflow may occur when computing S (to detect that requires tests on intermediate products). The final result for S must be a power of 2 so that dividing by S will not introduce round-off errors.

Over/Underflow in S

If we get an overflow, that means |AC| is a huge number and S could be taken as any big power of 2, say 2^{600}. Overflow means that both A and C are greater than 1; when we compute A/S and C/S, neither of these can underflow. A' and C' may still be large, but their product cannot overflow.† It must be far enough below the overflow threshhold that if $(B')^2$ overflows, A'C' is negligible. (Actually, $(B')^2$ could not overflow after such a huge scaling).

Question: It's not clear to me that a single S will do.

Answer: There is no single S. If S overflows or underflows, you don't choose S according to the formula.

Question: I meant a single S in any one situation.

Answer: That can be done. I'll indicate how to do it and leave the details to the students.

An underflow in computing S indicates A*C is very tiny. That means A and C are both less than 2^{40}. It is now enough to make S a small number like 2^{-500}. Then computing A/S and C/S will cause no serious problems. Computing B/S may overflow now, but that will be tested for further on in the program. If B' overflows, B'^2 will also and that will be caught later (if you are running in the mode that allows you to use infinite operands).

---

† Consider $A = 2^{1022+48}$, $C = 2^{1022+48}$, $AC = 2^{2044+96}$ overflow.
If $S = 2^{600}$, $A' = 2^{422+48}$, $C' = 2^{422+48}$, $A'C' = 2^{844+96}$ in range.

Overflow in B' = B/S

If B' overflows, then $(B')^2$ is so much larger than A'C' that we can neglect A'C' compared with B'. The roots then are relatively simple to compute:

$$R_1 = \frac{\frac{1}{2}C}{B}, \quad R_2 = \frac{2B}{A} \quad .$$

What If $(B')^2$ Overflows?

Using double precision to compute $D = (B')^2 - A'C'$ can lead to one problem.

You have computed B' and checked that it did not overflow. So you go into double precision to compute D and then check if it overflows.

However, if $(B')^2$ overflows, you will get kicked off. B' is now double precision. When you multiply the two upper halves of B' you generate an infinite operand. Then when you compute an (upper half)*(lower half) and try to add to the upper product, you pick up an infinite operand and get thrown off.

Solution: You must compute $(B')^2$ to single precision first and check for overflow. If it overflows, you know D will. If $(B')^2$ doesn't, D will not overflow either. Unfortunately, you will often compute $(B')^2$ to single precision and then to double precision as well. Or else you run in the mode that allows infinite operands.

Question: What if $(B')^2 - A'C'$ overflows but $(B')^2$ doesn't?

Answer: It will not happen that $(B')^2$ is so close to overflowing that adding a reasonable number -A'C' will push it over. There will be bounds on A'C' to insure this.

$$2^{-1024+47+96} < |A'C'| < 2^{1022+48-96} \quad .^{\dagger}$$

If $A'C' < 2^{1022+48-96}$, I cannot add it to a representable number and cause overflow. I don't want $(B')^2$ to underflow and still be significant, so set the lower bound on $|A'C'|$ to $2^{-1024+47+96}$.

The approximation when S over/underflows is crude because we cannot tell if AC overflowed by a little or a lot. A'C' could be $\sim 2^{900}$, but that is still in the acceptable range. If $(B')^2$ overflowed, A'C' can be thrown away without any more than a rounding error in double precision. Then the approximations R1 and R2 are correct to single precision.

There could be a problem in $R1 = \frac{1}{2}C/B$, if B is huge and C is tiny. It is important to form the product $\frac{1}{2}C$ first and then divide by B. If B is so large that underflow occurs, the root deserves to underflow. Divide C by 2. Then if underflow would have occurred in dividing C by B, it will occur in dividing $\frac{1}{2}C$ by B. You find out if $\frac{1}{2}C/B$ underflowed by testing if it is zero.

In computing R2, B/A cannot underflow, so you won't get a zero here. If B/A overflows, you may be kicked off the machine when you compute 2(B/A); you have to be careful. You cannot use the primed values to compute R2 because B' may have overflowed.

Computing S

I have to choose K (see flow chart) in such a way that if |AC| is in range, the intrusion of the scale factor will not cause difficulties. In getting to the point where D didn't overflow, I must be sure that A'C'

---

† overflow threshhold = $2^{1022+48}$
underflow threshhold = $2^{-1024+47}$ (smallest normalized operand for add box)

could not have overflowed or underflowed.

The problem is to get $|A'C'|$ into the range

$$2^{-1024+48+96} < |A'C'| < 2^{1022+48-96}$$

Suppose

$A = 2^{1022+48}(1 - 2^{-47})$ largest operand

$C = 2^{-1024+48}$ smallest operand ($2^{-1024+47}$ is zero to multiply box)

In this extreme case, I dare not divide A by a number less than 1, nor divide C by a number greater than 1. Hence S must be 1 here. This puts a condition on K.

$$AC \approx 2^{96-2}$$

$$2^{-K}AC \approx 2^{96-2-K}$$ to within a unit in the last place

Now I'll take the square root and do something to it and I'd better get 1.

I want a number bigger than 1 ($= 2^0$), so that when I take its SQRT and throw digits away, it will be 1. I don't want a number bigger than 2 after taking the square root, so the original number must be less than 4, or less than 2^2. In exponents of 2:

$$0 < 96 - 2 - K < 2 \quad .$$

So we have $96 - 2 - K = 1$

or $K = 93$.

That's the only value of K that will work on the CDC.

Question: You pulled the numbers A and C out of the machine and got one particular value for K.

Answer: If that one case is to work properly, K must be 93. Now the question is, will that value work for all other numbers?

Does K = 93 Work For All Other Numbers?

The approximation for S means I compute $2^{-K}|AC|$, take its square root and truncate it down to the next lower power of 2; that is, throw away the last 47 bits of the word. That is 2^L. We have just verified that if A and C are at opposite extremes of the range, S = 1.

Question: You're making some assumptions about the SQRT routine, that for numbers near 1 you don't end up too far down.

Answer: Let's see what's happening. $A = 2^{1022+48}(1-2^{-47})$, $C = 2^{-1024+48}$.

$$2^{-93}|AC| = 2^{96-2-93}(1-2^{-47})$$

$$\sqrt{2(1-2^{-47})}(1+e) \approx \sqrt{2} \approx 1.4$$

It is hard to see how any machine could be so far wrong on $\sqrt{2}$ that when you chop you get a number other than 1.

Now it is necessary to see that nothing goes wrong when A and C move from these extreme values. Let $A = 2^{1022+47}$. It has the same characteristic as before but is now a string of 0's instead of 1's after the high order 1. Then AC is reduced and the initial approximation of S is reduced. But S itself must not be reduced; if S < 1, A/S will overflow.

$$2^{-93}|AC| = 2^{95-2-93} = 2^0 = 1 \quad \text{exactly.}$$

When I take $\sqrt{T}$ and throw away digits, nothing bad will happen. By monotonicity, as long as $2^{1022+47} \leq A \leq 2^{1022+48}(1-2^{-47})$, nothing goes wrong.

We have to do the same check for C in its appropriate interval. As C increases, S cannot decrease, but we cannot allow S to increase such as to make C/S underflow. An argument similar to that for A will do.

The cases for A and C not at the extremes work out more easily.

The point of this odd argument is that by an artful choice of constants, which have to be verified for each machine individually, you can manage to have relatively few tests for over/underflow. We've discussed most of the tests except for the last ones to see if the roots over/underflowed.

Test For Sign of D, $D \leq 0$

You have complex or coincident roots and compute them in the obvious way.

$$R_R = B/A \quad \text{or} \quad B'/A'$$

If B/A over/underflows, you deserve it.

$$R_i = \sqrt{-D}/A' \quad .$$

D is representable without over/underflow, so the same is true of $\sqrt{-D}$, unless D is one of those numbers that is zero to the multiply box. Then the result depends on the SQRT routine. But that cannot happen since A'C' has been scaled to be nowhere near the underflow threshhold. Even cancellation from $(B')^2$ cannot take you near enough to the threshhold to bother the SQRT. You could get exact cancellation, but that is alright.

Notice that if you had some decent way of turning off the spurious over/underflow responses, you could run in that mode until the test on D had been made. Then you could restore the mode wanted by the user before computing the actual roots and if he wanted to be kicked off he would be.

The only over/underflows that occur now are those that deserve to happen because the roots over/underflow.

D > 0

The roots are real and distinct.

First you have to compute

$$F = B' + SIGN(SQRT(D),B') \quad .$$

Observe that F cannot overflow or underflow. We know $(B')^2$ didn't overflow. Therefore 2B' cannot ($\sqrt{D} \approx B'$), or $B' + \sqrt{A'C'}$ cannot ($\sqrt{D} \approx \sqrt{A'C'}$, remember the range for $|A'C'|$).

Now we compute the roots and they could over/underflow.

$$R1 = F/A'$$

$$R2 = C'/F$$

If either of these over/underflows, it deserves to.

About the Program

Observe that this program has a relatively simple flow chart, in that the tests are to some extent minimal. It is also getting close to being machine independent. It is my assertion that the scaling trick can be carried out on any machine that I know about. It would be possible to design a machine so that this trick would not work, because the numbers are represented in some peculiar way.

Once the scale factor has been chosen, there is nothing to indicate if the machine is binary or hexadecimal.

Another property of the program is that we haven't spent much more time than the minimum to solve a quadratic. The minimum is our program after the scaling has been done. We haven't more than doubled the minimum amount of time.

What About Automatic Theorem Proving?

Question: Some people at Stanford try to prove validity of programs by putting balloons around decisions. In proving the validity of your SQRT you took an analytic approach. But in this quadratic solver with its tests and decisions, you were reduced to looking at cases. Balloons wouldn't help. Is there some theory that says when you've exhausted the cases?

Answer: The approches used by the people at Stanford to prove validity make techniques which reduce in the end to an examination of cases seem abstract and impressive. There is no systematic way I know of to minimize the number of cases. In general, it is better to break the cases up in any way that makes sense to you, even if the number is then larger than necessary and tackle them. You'll find that arguments used in one case will work for another; maybe those two cases should have been one, but separating them won't have cost you very much.

The difficulty in their approach arises when you try to prove anything about a machine like ours which is capricious. If the program was reasonably simple and the number of rules was reasonably small, their formalization would appear to be quite successful. What I have done on the quadratic is essentially what they would do, stripped of abstractions. It is possible to write down comments that enable you, at any point, to tell what the state of the machine is, subject to certain parameters. The parameters depend on the data. Every time you pass through a decision you can give the new

parameters in terms of the old. You could verify that certain relations remain satisfied by those parameters. But there is no systematic way to generate those relations, which depend on your objective. The men at Stanford have yet to prove the validity of any program half as complicated as the quadratic solver.

Question: Their problems are typically logical ones, like sorts. They don't come into contact with the machine.

Answer: They use induction. They do not have inequalities, for which in critical cases you have to examine a finite number of integer variables and let them run through their values. That is not because their method is incapable of doing so. It can if you tell it to but that is where the work is and it is not part of their scheme.

Working out what to do with a proof involves cleverness in deciding which statements to test for validity, not in doing the actual technical manipulations which go into proving the validity of statements you've decided to test. The best I would expect from mechanical program verifiers would be that if you could reduce the verification of a program to a set of verifications of formal statements, which only required a certain amount of exhaustion of cases generated in a routine way which you'd rather not do yourself, then you'd let the machine do it.

The example of the 29 incorrect [19] square roots was a time when I had the machine do some verification. But deciding what routine to use required ingenuity. I don't think you can escape that for non-trivial programs. I think all you'd usually get from theorem proving was really just proof checking. But proof checking could be tedious and you may have trouble explaining to the machine that certain things are true (like properties of continuous functions).

The point is not that the machine cannot know everything. Rather, it is that in my attempt to explain to the machine what I know, I may be building in a misconception without realizing it.

Example. Find the maximum value of a continuous function on a closed interval. Everybody knows that the function achieves its maximum. But there is no algorithm which when given the program that generates the function and the endpoints of the interval can guarantee the maximum to an arbitrary preassigned precision. If such a program existed, it could solve mathematical problems like Fermat's last theorem, or the Riemann hypothesis. So what do we mean when we write down MAX(f(x),a,b)? We aren't sure we should write that down perfectly freely. But we do it anyway. There could be a mistake in our concept of a maximum which we may infuse into a proof, which was intended to be constructive. By introducing this non-constructive idea we may have clobbered the proof without realizing it.

The proof checker has then checked the validity of a certain argument following from certain assumptions without really proving the theorem. I'm afraid people will assume that anything checked by a proof checker is true. It is only true if the assumptions were, but they could be true in a non-constructive sense and not true in a constructive sense.

9. HOW CAN WE ADD UP A LONG STRING OF NUMBERS? - STANDARD ALGORITHM

Today we shall demonstrate the difficulties that arise from poor design of floating point hardware when we try to add up a sequence of numbers. This simple problem has been chosen because the analysis is relatively simple. We shall see that the analysis becomes more difficult as we weaken our demands on the arithmetic unit.

The usual program for evaluating $\sum_1^n x_j$ would be

```
      S = 0
      DO 1 J=1,N
    1 S = S+X(J)
```

The final result of this program is

$$s_n = ((\cdots((x_1+x_2)(1+\sigma_2)+x_3)(1+\sigma_3)+\cdots+x_{n-1})(1+\sigma_{n-1})+x_n)(1+\sigma_n) \quad .$$

We could also write

$$s_n = \sum_{j=1}^{n} x_j(1+\xi_j) \quad , \quad 1+\xi_j = (1+\sigma_j)(1+\sigma_{j+1})\cdots(1+\sigma_n) \quad , \quad \sigma_1 \equiv 0 \quad .$$

If we assume $|\sigma_j| \leq \varepsilon$ then we find that

$$|\xi_j| = |(1+\xi_j)-1| \leq (1+\varepsilon)^{n+1-j} - 1 \sim (n+1-j)\varepsilon + O((n\varepsilon)^2) \quad .$$

Thus our computed value is actually the sum of slightly altered numbers. The alteration is at most a few units in the last place times the number of summands. This does not seem intolerable, at least for small values of n.

Let us see what happens when we try to solve an ordinary differential equation by summing this way. A typical problem would be

$$dy = f(t,y)dt \quad ,$$

which we would try to solve by an algorithm such as

$$y(t_0) = \text{given}$$
$$y(t_{n+1}) = y(t_n) + F(t_n, y(t_n))(t_{n+1} - t_n) \quad .$$

If we take small steps, we will sum many small increments. Most of the increments might be much smaller than $y(t_0)$, the given initial value. We could have the situation

$$\begin{array}{r} y(t_0) = \text{XXXXXX.} \\ \Delta y(t_0) = \quad \text{XX.XXXX} \\ \hline y(t_1) = \text{XXXXXX.////} \end{array}$$

The digits to the right of the decimal point are lost in rounding. One way of looking at this digit loss is as a perturbation of $y(t_0)$. That is, the rounded result $y(t_1)$ is the correct result of addition of $\Delta y(t_0)$ to a slightly smaller $y(t_0)$. This perturbation is in the seventh place of $y(t_0)$ and thus is fairly negligible, since it is, after all, less than the uncertainty in $y(t_0)$ caused by rounding to six figures. Unfortunately, if there are a million steps in the computation, transferring each rounding error to the initial value might well change it beyond recognition. Then we would have the right answer -- to a completely wrong problem.

There is a more fruitful way of looking at the computation. That is to imagine that instead of computing F, an average of f, at each step we are actually computing another function that yields XX.0 instead of XX.XXXX so that the addition is always performed correctly. So we get the correct result for a problem with a somewhat different function which agrees with f only to about two figures.

Use Double Precision

In general we would like to solve a problem closer to the given one. There are several tricks which we can employ to do this. Suppose, for instance, that we only do a double precision add at each step:

$$y(t_0) = \text{XXXXXX.000000}$$
$$\Delta y(t_0) = \text{XX.XXXX00} \quad .$$

If double precision hardware is available the extra cost of this is small. We can see that as long as $\Delta y(t_0)$ affects the last (single-precision) place of $y(t_0)$, then none of $\Delta y(t_0)$ will be lost in the addition and the sum will remain accurate. Generally, as long as Δy is large enough to alter the last single precision place of y, any error introduced by summation will be small and the sum will be nearly accurate in single precision. We could trace this small error introduced by summation back to being a single-precision perturbation in the integrated functions. (The worst that could happen would be that the steps would be so small that Δy would not affect the left half of y, but this could not happen often or else the numerical process would be regarded as impractically slow.)

The value of this technique is that the bound on $|\xi_j|$ is changed from proportional to ε to proportional to ε^2. Then we can sum as many as $\sim \frac{1}{\varepsilon}$ terms before worrying about rounding error showing up in our single precision result. As a concrete example, consider a million steps on the 360. Short word arithmetic has six hexadecimal digits, so the perturbation on the input data could be as large as 1. Long word arithmetic carries fourteen hexadecimal digits, so about 10^{10} terms could be added before the perturbations become serious.

Therefore all we need do is add the statement DOUBLE S to our previous

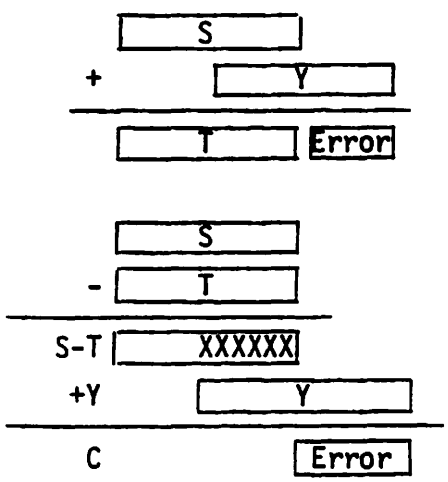
program. To avoid having the double precision propagate into other parts of the program where it is not necessary, truncate S to another single precision variable and use that variable elsewhere.

Suppose There's No Higher Precision

Suppose, however, that double precision is not available, or that we were in double precision to begin with! Fortunately, we can, in effect, simulate those parts of double precision that interest us by programming one of a number of well known tricks. One of these is as follows:

```
      S = 0
      C = 0
      DO 1 J=1,N
      Y = C+X(J)
      T = S+Y
      C = (S-T)+Y
1     S = T
      SUM = S+C  (rounded)
```

C represents the rounding error computed in the previous step. Y is a slightly perturbed summand which is added to the sum S via the temporary sum T. In pictures:



We don't expect any error in S-T <u>if our machine has a guard digit</u>. The characteristics of S and T are either equal or differ by one, so we expect their difference to be computed correctly. The difference will be about the size of Y and will have a number of leading zeros, causing a left shift, so that we are sure that the difference will be accurate.

We need to apply our model to get a credible proof of the effectiveness of this program. We see that the values in storage are

$$
\begin{aligned}
s_0 &= 0 \\
c_0 &= 0 \\
y_j &= (c_{j-1}+x_j)(1+\eta_j) \\
s_j &= t_j = (s_{j-1}+y_j)(1+\tau_j) \\
c_j &= ((s_{j-1}-s_j)(1+\sigma_j)+y_j)(1+\gamma_j) \quad .
\end{aligned}
$$

For each Greek letter, $|\text{Greek}| \leq \varepsilon$. By induction the result is

$$
\begin{aligned}
s_n + c_n &= \sum_1^n (1+\xi_j)x_j \\
1+\xi_j &= (1+\eta_j)(1-\sigma_j) + O((n+1-j)\varepsilon^2) \quad .
\end{aligned}
$$

That is, we've perturbed the input by a few ulps in single precision <u>independent of</u> n and a number of ulps in double precision proportional to n. This routine gives an answer about as good as we deserve without invoking the double precision package. There are cases when the algorithm will not work on machines that chop before rounding. On our CDC machine, if s and t differ slightly, with different characteristics, their difference might be zero. Then $\sigma_j = -1$ which ruins everything. Now we wouldn't <u>expect</u> such s and t to occur very often -- they would be numbers just slightly on different sides of a power of two. But such a claim is hard to prove.

In 1968 van Reeken "discovered" that the algorithm worked correctly on every machine and input he could think of, for the purpose of computing running averages:

$$M_n = \frac{\sum_1^n v_j}{n} = M_{n-1} + \frac{v_n - M_{n-1}}{n} \quad .$$

The purpose of computing the running mean by means of the recurrence was actually to use it in computing a running standard deviation by means of a similar recurrence. Such a recurrence requires a square root of a sum that might become negative due to rounding errors if the v_n's are all nearly the same. Therefore we would want to compute that sum using the single-precision algorithm described above.

Kahan has since discovered a counter-example that shows that the algorithm <u>will</u> fail on the running average problem on machines with no guard digit. The average is over three million values satisfying

$$\frac{1}{2} \leq v_j \leq \frac{3}{2} \quad .$$

The last six digits in the single precision average are wrong, because s-t is computed incorrectly about $\frac{1}{3}$ of the time, on the 6400. s and t are nearly always just on opposite sides of 1. Clearly the erroneous result depends rather strongly on the careful choice of the input. Nevertheless, it is hard to understand <u>a priori</u> why a computer should average reasonable numbers so poorly. But this is just a specific case of the general principle that it is very hard to understand computers that do not follow simple rules. After all, the trick in the algorithm is so easy to discover that at least half a dozen persons have done so independently. It is much more difficult, however, to determine on which machines it will work...or fail.

If this trick were needed only to solve differential equations, it

would not be worth crying over its loss. You then would write a double precision subroutine, in assembly language if necessary and call that to add your numbers in double precision.

Why You Want Exact Differences[††]

But this reaches into many other areas. It affects our ability to code higher precision arithmetic out of single precision by subroutines that are partially machine independent. This may not appear important to you, but when people produce numerical algorithms they would like them to work on any reasonable computer. In the middle they will do calculations to essentially higher precision by some trick. The writer could insist that your compiler provide double precision. But Algol usually doesn't (except on the B5500).

But there is a nontheorem that tells you there is no theorem to tell you that if you want to solve this problem to single precision you must carry n-tuple precision. There cannot be such a theorem to specify n, since n-tuple precision can be simulated by single precision.[†]

People who talk about coding multiple precision with single generally miss a couple of points. One is that they insist upon being able to compute Y+Z exactly, for all combinations of Y and Z as the sum of two other floating point numbers, say Y+Z = S1+S2, where S1 and S2 almost constitute a double precision number, with S1 the leading and S2 the trailing parts. But then the difference between Y+Z as computed and Y+Z as it ought to be might not be a machine representable number in single / precisior It also assumes that S1 and S2 must have the same sign, and that isn't

[†]A report by T.J. Dekker shows how to do this on "clean" machines, that is, on machines on which the preceding trick will work. There is a book in manuscript by Patrick Sterbenz in which he also shows how to code double precision arithmetic from single, provided the machine is reasonable, like 360 equipment. Knuth, Section 4.2, also talks about this and even has the trick enshrined as a theorem.

[††]Exact differences are important for sums with good error bounds.

true.†

The issue is not what can you do exactly, even though if you have a solution for that you can do everything else.

The issue is that if you have a "dirty" type of single precision arithmetic, can you make up a double precision arithmetic that is also dirty, but not unreasonably so. The answer is yes, but it is harder.

If you can get a difference exactly (if it is representable exactly), then for all the kinds of arithmetic we've been studying you have the equipment to do double precision arithmetic, coded in FORTRAN or ALGOL, using only the ordinary floating point arithmetic. Then you can pyramid. And the code is transportable to another machine. You need only verify the most rudimentary aspects of the machine, like its number base, number of digits carried in single precision. If you work at it long enough you can get operations to be performed exactly.

Question: In trying to simulate the double precision, wouldn't it be better to unpack the numbers and work on them as integers?

Answer: Yes, but I am trying to write a FORTRAN (or ALGOL) program that will compute a difference exactly.

Question: What's the good of a FORTRAN program, if you have to rethink and reprove that the program will work when you go to a different machine?

Answer: If this program is done correctly, it will work for any machine whose arithmetic is somewhat messy. Then you pyramid this operation, using single precision to get double, then double to get quadruple, and so on, until the messiness catches up with you, somewhere around 128-length precision.

Question: With that length, isn't it still better to work with your own number representation?

---

†This red herring is raised by Knuth, Dekker and Sterbenz.

Answer: More efficient, yes. But the idea was to show that you could write in an essentially machine independent language.

Question: But why not use integer arithmetic in FORTRAN, if you're going up to 100-length words?

Answer: Then you have to take into account machines like the 7094, where integers are limited to 15 bits (FORTRAN II) and overflow is not detectable in an intelligent way. Could you code things there? In FORTRAN IV, you have the 36 bits, but overflow is even less detectable. You would have to clear the overflow bits before and test after each operation. In fact, you could not only do arbitrary precision but be infinitely precise; it's rational arithmetic.

Question: Your code is transportable only with some assumptions about the machine. And you haven't stated what those assumptions are.

Answer: The assumption would be that whenever they do an arithmetic operation the result is no worse than what you would have gotten had you changed both the operands by a unit in the last place.

Question: You've drawn pictures of how the numbers appear in the machine. What if they don't appear that way, but involve lots of funny shifts?

Answer: The algorithms would work, but the proof gets harder.

Question: It seems you're assuming more than that the result you get is no worse than making a slight modification in the operands because you keep making statements that "this calculation can be done exactly", but that wouldn't follow from your original assumption.

Answer: No, the original assumption is that if two operands have sufficiently few digits, the operation is done exactly (the digits line up). There are some numbers for which you get what you should, in the absence of rounding errors. Another assumption is the one about modification of the

operands. The tricks are designed to work toward the numbers with few digits, where you can do something exactly and then work your way back to the original problem.

Question: In the time you spent working out this trick in FORTRAN, it could have been done in machine language for five different machines.

Answer: You could, but the trick, in FORTRAN, can be imbedded in code and carry the code wherever you want to any machine as long as it isn't too weird.

Question: Are the properties you require (for the trick) going to be easily determinable for any machine?

Answer: Yes, they will be for single precision and once they are determined, the scheme will work for double precision, etc.

Question: If what you say is true, about being able to devise a trick for even dirty machines, then why should Knuth, Dekker and Sterbenz continue to lay out red herrings?

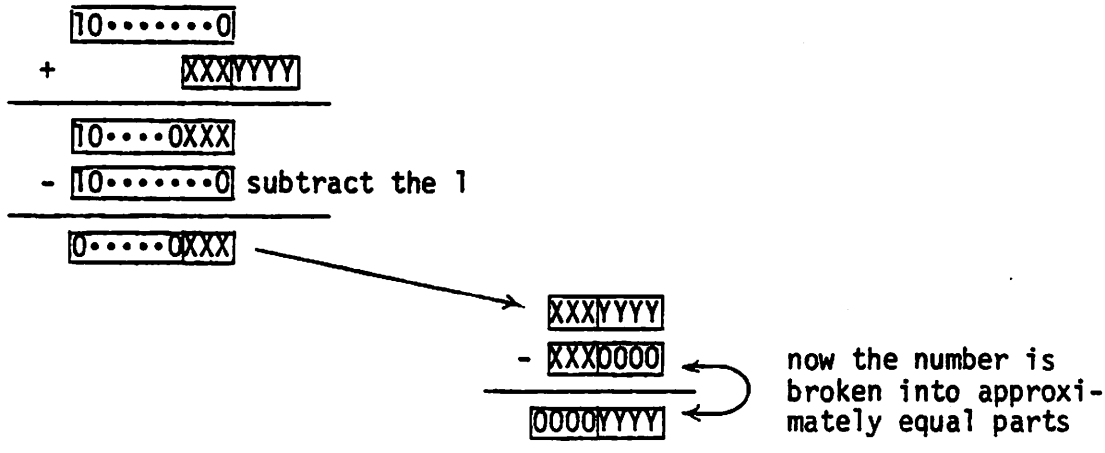
Answer: They started out from a paper by Møller that appeared in BIT (about the same time as my note in the CACM), which stated in a theorem that floating point numbers are related in a certain way, provided the arithmetic is such and such, and he set a pattern for the others. Knuth is not a numerical analyst and doesn't care about these things and he just pursued that rather interesting mathematical pattern. Dekker works mostly with "clean" machines, so he worked out his scheme for them. Sterbenz worked with the group in SHARE that got IBM to change its hardware.

Question: By the time you find someone who knows enough about the machine to answer your questions, you could have coded in machine language.

Answer: That I dispute. Consider the B5500. We know what the characteristics are: 13 octal characters with such and such arithmetic. We know

that now, but not the order code. It would be easier now to code the FORTRAN rather than learn the order code. Or say, in a hurry, I want you to produce roughly quadruple precision add. You'd find it faster probably to take the double precision add and trick it, even on a machine whose assembly language you are utterly familiar with.

If worse comes to worst, to clear out some digits (so you can do exact arithmetic), do the following:



If you can do this, the rest is easy.

Proof for the Pseudo-Double Precision Accumulation

We write the algorithm here with Greek letters for each error committed. $|\text{Greek}| \leq \varepsilon$.

```
      S = 0.                 s_0 = 0
      C = 0.                 c_0 = 0
      DO 9 I=1,N             for j = 1,n
           Y=C+X(I)             y_j = (x_j+c_{j-1})(1+η_j)
           T=S+Y                s_j = (s_{j-1}+y_j)(1+τ_j)
           C=(S-T)+Y            c_j = ((s_{j-1}-s_j)(1+σ_j)+y_j)(1+γ_j)
    9      S=T
      SUM=S+T
```

We will see that this program works on machines that normalize and then round. The equations with Greek letters in them are all based on the assumption that the sum of a and b is computed as $(a+b)(1+\gamma)$.

We intend to demonstrate the following facts by induction:

$$s_n + c_n = \sum_{i=1}^{n} \chi_{n,i} x_i \qquad \chi_{n,i} = (1+\eta_i)(1-\sigma_i+0(\epsilon^2))$$

$$c_n = \sum_{i=1}^{n} \Gamma_{n,i} x_i \quad (n > i) \qquad \Gamma_{n,i} = -\tau_n + 0(\epsilon^2)$$

$$\Gamma_{n,n} = (-\tau_n-\sigma_n) + 0(\epsilon^2)$$

Exercise. Determine the coefficient of ϵ^2 and show that it is $0(n)$.

Note that these insertions imply that the rounding error we attach to x_i is of order ϵ and is independent of n to single precision. The first three steps of the computation provide the basis for the induction.

$$y_1 = x_1 \quad \eta_1 = 0$$

$$s = x_1 \quad \tau_1 = 0 \qquad \Omega_{1,1} = 1$$

$$c_1 = 0 \quad \sigma_1 = 0 \quad \gamma_1 = 0 \qquad \Gamma_{1,1} = 0 = -\tau_1$$

$$y_2 = x_2 \quad \eta_2 = 0 \qquad \chi_{1,1} = 1 = (1+\eta_1)(1-\sigma_1)$$

$$s_2 = (1+\tau_2)x_1 + (1+\tau_2)x_2 \qquad \Omega_{2,1} = 1+\tau_2 \quad \Omega_{2,2} = 1+\tau_2$$

$$c_2 \doteq (-\tau_2)x_1 + (-\tau_2-\sigma_2)x_2 \qquad \Gamma_{2,1} \doteq -\tau_2 \quad \Gamma_{2,2} \doteq -\tau_2-\sigma_2$$

$$\chi_{2,1} \doteq 1 = (1+\eta_1)(1-\sigma_1)$$

$$\chi_{2,2} \doteq 1-\sigma_2 = (1+\eta_2)(1-\sigma_2)$$

$$y_3 \doteq (1+\eta_3)(-\tau_2)x_1 + (1+\eta_3)(-\tau_2-\sigma_2)x_2 + (1+\eta_3)x_3$$

$$s_3 \doteq (1+\tau_3)x_1 + (1+\tau_3)(1-\sigma_2)x_2 + (1+\tau_3)(1+\eta_3)x_3$$

$$\Omega_{3,1} \doteq 1+\tau_3 \quad \Omega_{3,2} \doteq (1+\tau_3)(1-\sigma_2)$$

$$\Omega_{3,3} \doteq (1+\tau_3)(1+\eta_3)$$

$$c_3 \doteq (-\tau_3)x_1 + (-\tau_3)x_2 + (-\tau_3-\sigma_3)x_3$$

$$\Gamma_{3,1} \doteq -\tau_3 \qquad \Gamma_{3,2} \doteq -\tau_3 \qquad \Gamma_{3,3} \doteq -\tau_3 - \sigma_3$$

$$\chi_{3,1} \doteq 1 \qquad \chi_{3,2} \doteq 1 - \sigma_2 \qquad \chi_{3,3} \doteq 1 + \eta_3 - \sigma_3 \doteq (1+\eta_3)(1-\sigma_3)$$

Here we have written $\Omega_{n,i} = \chi_{n,i} - \Gamma_{n,i}$ so that $s_n = \sum_{i=1}^{n} \Omega_{n,i} x_i$, and we dropped all terms of order ε^2 or ε^3, so that "$+O(\varepsilon^2)$" should be appended to each approximate equality to make it exact. These steps provide a basis for the induction and indicate how the induction hypothesis was chosen. Now let us drop subscripts of n and write -1 for $n-1$ in what is to follow. Then we find

$$y = \chi(1+\eta) + (1+\eta)\sum_{i=1}^{n-1} \Gamma_{-1,i}\chi_i \quad ,$$

$$s = \sum_{i=1}^{n} \Omega_{n,i}\chi_i = (1+\tau)\{\sum_{i=1}^{n-1} \Omega_{-1,i}\chi_i + \chi(1+\eta) + (1+\eta)\sum_{i=1}^{n-1} \Gamma_{-1,i}\chi_i\} \quad .$$

Assuming with the induction hypothesis that Γ and χ are independent of x, we find formally that

$$\text{for } i < n, \quad \Omega_{n,i} = (1+\tau)\Omega_{-1,i} + (1+\tau)(1+\eta)\Gamma_{-1,i}$$

$$\Omega_{n,n} = (1+\eta)(1+\tau) \quad .$$

Also

$$c_n = \sum_{i=1}^{n} \Gamma_i x_i = (1+\gamma)\{\chi(1+\eta) + (1+\eta)\sum_{i=1}^{n-1} \Gamma_{-1,i}\chi_i\}$$

$$+ (1+\gamma)(1+\sigma)\{\sum_{i=1}^{n-1} (\Omega_{-1,i} - (1+\tau)\{\Omega_{-1,i} + (1+\eta)\Gamma_{-1,i}\})\chi_i$$

$$- (1+\eta)(1+\tau)x\} \quad .$$

Then for $i < n$

$$\Gamma_{n,i} = (1+\gamma)(1+\sigma)(-\tau)\Omega_{-1,i} + (1+\gamma)(-\sigma-\tau-\sigma\tau)\Gamma_{-1,i}$$

and

$$\Gamma_{n,n} = (1+\gamma)(1+\eta)(-\sigma-\tau-\sigma\tau) = -\sigma_n - \tau_n + 0(\varepsilon^2) \quad .$$

Note that

$$\begin{aligned} \chi_{n,n} &= \Gamma_{n,n} + \Omega_{n,n} \\ &= (1+\gamma)(1+\eta)(-\sigma-\tau-\sigma\tau) + (1+\eta)(1+\tau) \\ &= (1+\eta_n)(1-\sigma_n+0(\varepsilon^2)) \\ &= (1+\eta_i)(1-\sigma_i+0(\varepsilon^2)) \quad . \end{aligned}$$

Therefore the "induction" hypothesis is verified directly for $\chi_{n,n}$ and $\Gamma_{n,n}$ without any induction.

For the case where $i < n$ we have deduced the relation

$$\begin{pmatrix} \Omega_i \\ \Gamma_i \end{pmatrix} = \begin{pmatrix} 1+\tau & (1+\eta)(1+\tau) \\ (1+\gamma)(1+\sigma)(-\tau) & (1+\gamma)(1+\eta)(-\sigma-\tau-\sigma\tau) \end{pmatrix} \begin{pmatrix} \Omega_{-1,i} \\ \Gamma_{-1,i} \end{pmatrix} \quad .$$

We are interested in χ rather than Ω so we note that

$$\begin{pmatrix} \chi_i \\ \Gamma_i \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \Omega_i \\ \Gamma_i \end{pmatrix}, \quad \begin{pmatrix} \Omega_{-1,i} \\ \Gamma_{-1,i} \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \chi_{-1,i} \\ \Gamma_{-1,i} \end{pmatrix} \quad .$$

When we perform the indicated matrix multiplications we find that

$$\begin{pmatrix} \chi_i \\ \Gamma_i \end{pmatrix} = \begin{pmatrix} 1+0(\varepsilon^2) & \eta-\sigma+0(\varepsilon^2) \\ -\tau+0(\varepsilon^2) & -\sigma+0(\varepsilon^2) \end{pmatrix} \begin{pmatrix} \chi_{-1,i} \\ \Gamma_{-1,i} \end{pmatrix} \quad .$$

We are now ready to apply the induction hypothesis for $1,2,\ldots,n-1$ to see if it holds for n. First there is the case $i < n-1$.

Then

$$\begin{pmatrix} \chi_{n,i} \\ \Gamma_{n,i} \end{pmatrix} = \begin{pmatrix} 1+0(\varepsilon^2) & \eta-\sigma+0(\varepsilon^2) \\ -\tau+0(\varepsilon^2) & -\sigma+0(\varepsilon^2) \end{pmatrix} \begin{pmatrix} (1+\eta_i)(1-\sigma_i+0(\varepsilon^2)) \\ -\tau_{-1}+0(\varepsilon^2) \end{pmatrix}$$

$$= \begin{pmatrix} (1+\eta_i)(1-\sigma_i+0(\varepsilon^2)) \\ -\tau_n+0(\varepsilon^2) \end{pmatrix} .$$

The final possibility, that $i = n-1$ so $\Gamma_{n-1,n-1} = -\tau_{n-1} - \sigma_{n-1} + 0(\varepsilon^2)$, leads to the same result. Note that a variety of induction hypotheses would satisfy the induction step and the computation of the first few values in the basis is essential to find the correct hypothesis.

We must step back from the blizzard of subscripts and understand that the important step was realizing, from the picture, that a useful induction hypothesis could be formulated. Note that our picture is based on the standard way of doing floating point addition, yet the proof derived from the picture is completely valid for any machine, such as a properly designed logarithmic machine, in which the arithmetic is done by rounding the correct result.

This type of algorithm has been developed independently by

Babuška, "Numerical Stability in Mathematical Analysis," Proceedings of IFIP Congress 1968, Vol. 1.

Møller, "Quasi-Double Precision in Floating Point Addition," BIT 5, 37-50 and 251-255.

(Møller's algorithm was designed for bad machines and is not optimal for good ones!)

Knuth, Seminumerical Algorithms, pp. 201-204, 1969.

10. ADDING UP A LONG STRING OF NUMBERS -- A MYSTERIOUS ALGORITHM WITH A MAGIC CONSTANT

The previous algorithm will not work on a CDC 6000 machine. However, we can make a somewhat similar algorithm work, even in the face of apparent theoretical difficulties.

Theoretical Difficulties (?)

The difference in machines is in the model of arithmetic that may be applied to them. The good ones compute $A+B$ as $(a+b)(1+\gamma)$, while the others yield $(a(1+\alpha)+b(1+\beta))$. Perhaps, though, a more clever algorithm could be devised that would yield a result in single precision almost as good as if double precision had been used, even on machines that compute according to the second model. The Russian Viten'ko (U.S.S.R. Computational Math. and Math. Physics 8 (5) pp. 183-195) has shown that for any algorithm for a sum $\sum_1^n x_j(1+\xi_j)$, on a machine where the second model must be applied, the bound $\xi \sim \varepsilon \log_2 n$ is the best possible. Basically, addition on a binary tree structure is best possible. For example,

$$\sum_{j=1}^{8} x_j = (((x_1+x_2)+(x_3+x_4)) + ((x_5+x_6)+(x_7+x_8))) \quad .$$

Clearly each x will have $3 = \log_2 8$ rounding errors attached to it. Any algorithm for computing the sum of eight numbers will have at least three Greek letters attached to at least one of the operands.

DIF1 - The Algorithm That Defies Viten'ko's Theorem

Viten'ko's theorem is true, yet misleading. Even on machines such as the CDC we can sum many numbers without explicit double precision, with a backward error ξ of a few units independent on n in single precision,

and $O(n^2)$ units in double precision. For instance, the following mysterious algorithm DIF1 will work:

```
      S = 0.0
      C = 0.0
      DO 1 J=1,N
           Y = C+X(J)
           T = S+Y
           F = 0.0
           IF(SIGN(1.0,Y).EQ.SIGN(1.0,S))  F = (0.46*T-T)+T
           C = ((S-F)-(T-F))+Y
    1      S = T
```

This code is machine independent on all North American machines with floating hardware. However, the proof that it will work is very difficult. Yet nothing about it contradicts Viten'ko's work. The model $(a(1+\alpha)+b(1+\beta))$ for addition is not categorical and even bad machines often behave better than this model indicates. The mysterious algorithm is based on a careful exploitation of some of these loopholes where the model is overly pessimistic.

Proof of DIF1, the Magic Constant Algorithm

If we compare the proof to follow with that in section [9], we see that an entirely new line of reasoning is necessary. First we must see how the special fiddling works. In the previous program we wrote

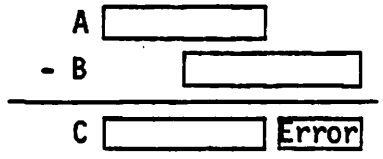
$$C = (S-T)+Y$$

in order to get

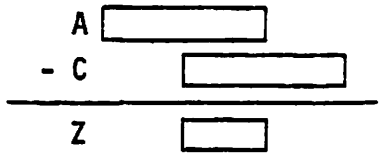
$$c = ((s-t)(1+\sigma)+y)(1+\gamma) \quad .$$

This just won't work on machines such as the CDC. Nonetheless, there is a kind of arithmetic that can always be done precisely. By exploiting this

kind of arithmetic, we can compute a c satisfying such an equation, with σ and γ perhaps somewhat larger. Suppose we want to determine the error in A-B as follows:



A > B > 0



The important thing to notice is that Z = A - C is computed precisely. C was formed as a number with the characteristic of A and then was normalized. When it is subtracted from A the part shifted out and lost is just the previous normalization's zeroes.

Consequently we can conclude that $z \doteq b$ and that their difference could be computed precisely, except in the case that their characteristics differ by one. The fact that B - Z may not be computed precisely would be considered disastrous by Møller and Knuth. We will find that a small perturbation in B is no worse than a small perturbation in the numbers we are trying to sum.

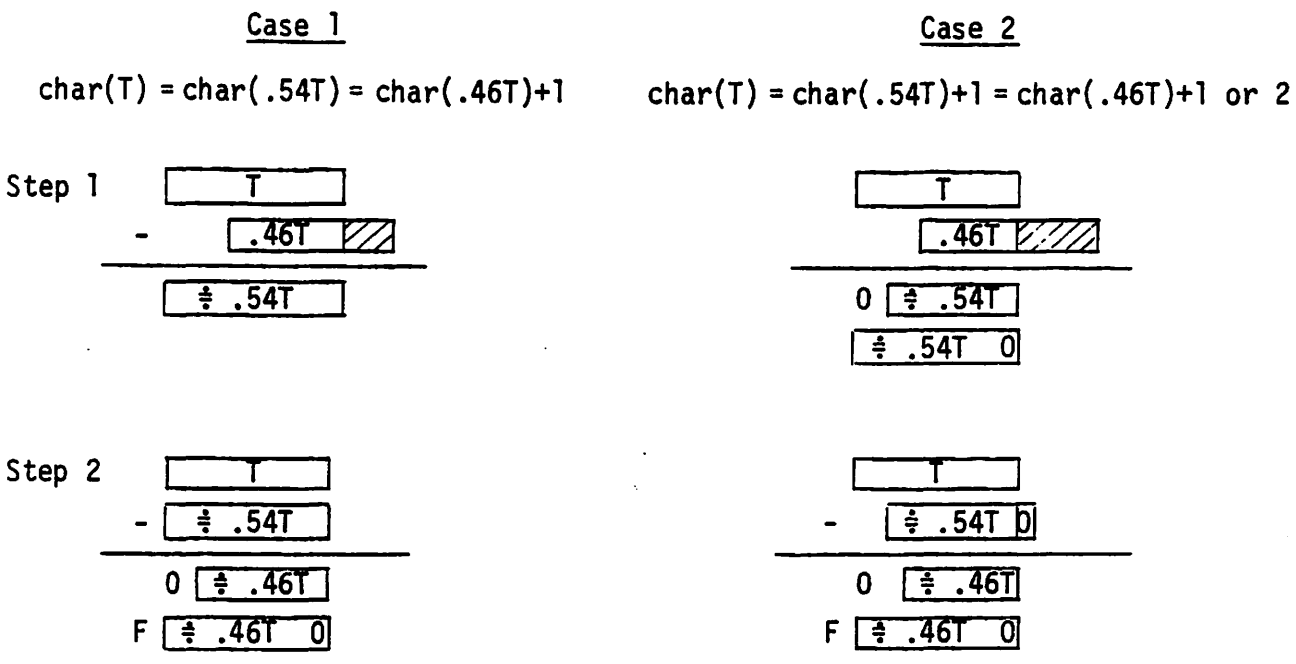
We will replace the statement

$$C = (S-T) + Y$$

by

```
F = 0.0
IF(SIGN(Y).EQ.SIGN(S))  F = (0.46*T-T)+T
C = ((S-F)-(T-F))+Y
```

We will need a picture to understand this:

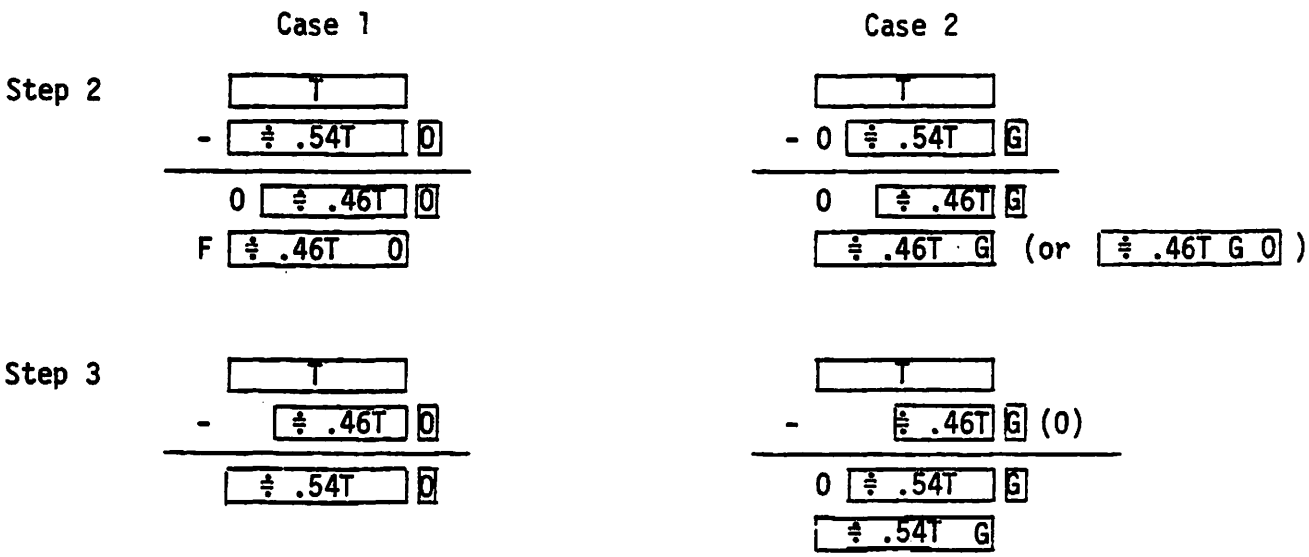


F has been fabricated in such a way that it can be subtracted precisely from T. The proof of this statement depends on the machine. First, consider a machine like the 650 which does arithmetic by dropping right-shifted bits. Such arithmetic is illustrated in the picture. F is formed from a number with the characteristic of T, by left-normalizing and inserting zeros at the right. Then when T-F is computed, F is right shifted so that those same zeros chop off, with no loss of accuracy, so that T-F is exact.

A similar argument applies if a whole guard word is used and then discarded as on the CDC 6400. The contents of the guard words are zero when T-F is computed.

Suppose there is a guard digit as on the IBM 360 series. Then we must distinguish two cases, according to char(.54T).

Guard digit



In the first case, char(.54T) = char(T) = char(.46T) + 1. Then the guard digit is always a zero and the answer T-F is precise. In the second case, the guard digit may not be zero. But F is always left shifted at least once after step 2, so the guard digit is saved. Then when T-F is computed the guard digit is shifted back to the proper position. Therefore it may be non-zero. But char(T-F) < char(T) so T-F is right shifted at least once, so the guard digit is saved and no error is made.

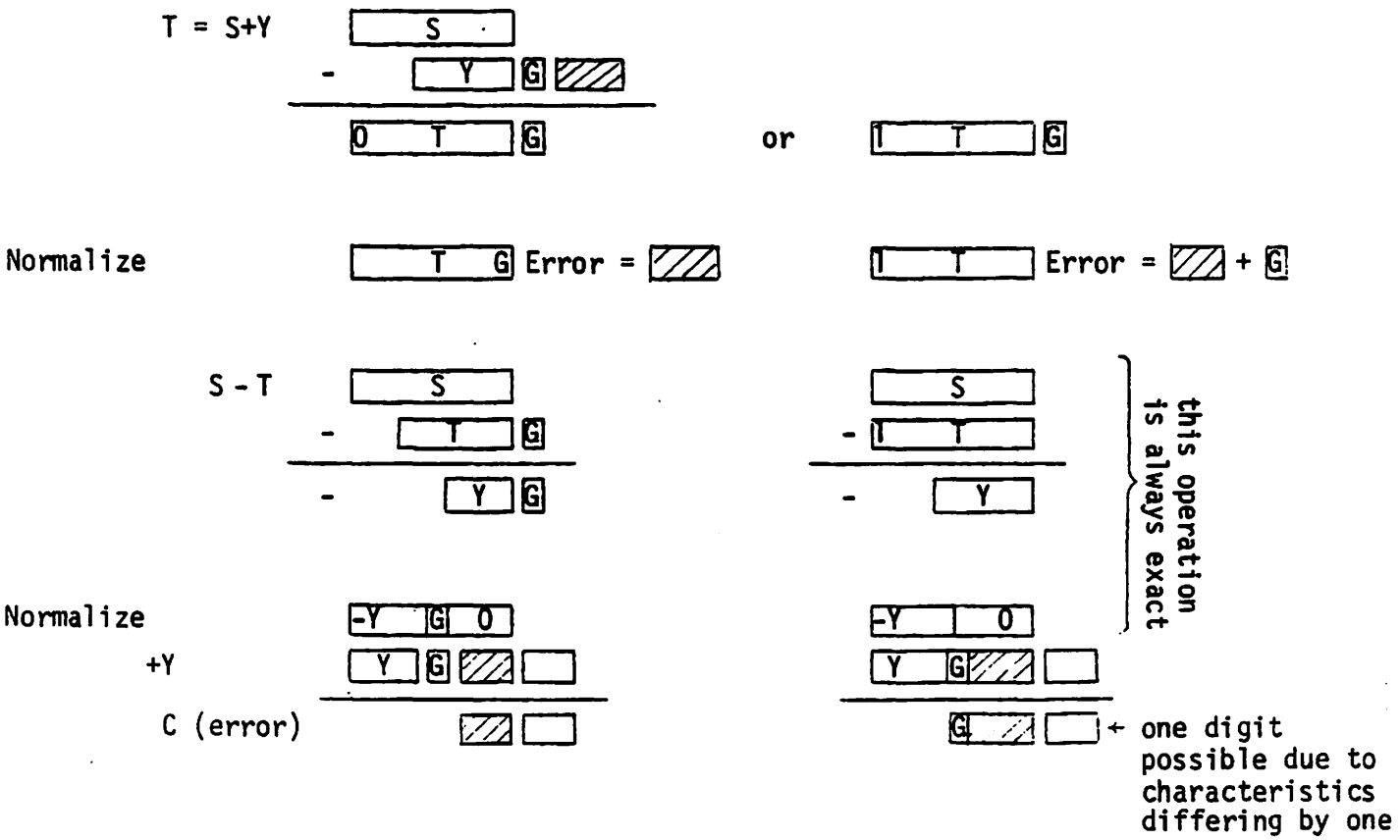
Suppose finally that a whole guard word is kept in the subtraction and used in the subsequent normalization as is done by the IBM 7094. Case 1 is precisely as in the case where digits are discarded. The guard words are entirely zero. Case 2 is precisely as in the case of the guard digit. The only extra digit that might be saved in the guard word is always a zero, so it makes no difference.

We are satisfied that T-F has no error. Now we must examine S-F to determine what happens when we compute

$$T = S + Y$$
$$\vdots$$
$$C = ((S-F)-(T-F)) + Y \quad .$$

Suppose first that $|Y|$ is substantial, say at least $\frac{1}{2}|T|$. Then $((S-F)-(T-F))$ is essentially $(S-T)$ to a few ulps, and is approximately $-Y$, to within a few ulps of Y, so that C is computed to be a few ulps of Y. Then we can safely say $c = ((s-t)(1+\sigma)+y)$, with a small σ, perhaps 2ε. Then we can map the error back to a small perturbation in X, as before, unless X is so small we don't care about it.

Suppose next that $|Y|$ is so small that $S \doteq T$, and sign(S) $\neq$ sign(Y). Then $|S| > |T|$, $F = 0$, and T was formed by a magnitude subtract from S. $S-T$ is always correct, so that $\sigma = 0$. Then $(s-t)+y$ will also be done correctly, except for possibly an ulp due to different characteristics. We illustrate with the case of a guard digit:

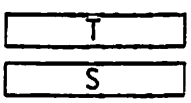


All arithmetic units will work properly in this case to give C almost exactly equal to (s-t)+y.

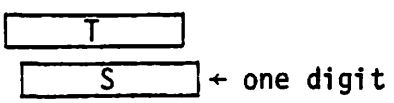
Suppose, however, that |Y| is small so that S ≒ T, and the signs of S and Y agree. Then |S| < |T|. In fact, |T| > |S| > |T-F| > |S-F| > |F|. Remember that T-F is always precise. What about S-F? The only way accuracy is lost is if F is right shifted so far with respect to S that non-zero digits in F extend to the right of S-F so that they can't be included in the result. But S is between T and F. Consequently F will be right shifted with respect to S no farther than with respect to T, and perhaps less. Since T-F is precise, S-F must also be precise.

We only need to know if (S-F) - (T-F) is precise. This will certainly be the case if char(S-F) = char(T-F). Therefore we only need to locate and investigate the cases where char(S-F) < char(T-F).

First we need to establish what the possible alignments relative to T might be, for each operand. S ≒ T so it might have relative alignment 0 or 1 in the addition unit:



relative alignment 0



relative alignment 1

Now F ≒ .46T, so it could have alignment 0 or 1 with respect to T, or even 2. The case of relative alignment 2 occurs only on binary machines, and requires that a characteristic jump occur between .46T and .50T, and again between .92T and 1.0T. Since S-F and T-F are about .54T, they must have the same characteristic and hence the same alignment relative to T, which could be 0 or 1.

With these facts and monotonicity,

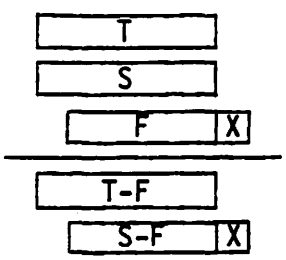
$$\text{char}(T) \geq \text{char}(S) \geq \text{char}(T\text{-}F) \geq \text{char}(S\text{-}F) \geq \text{char}(F) \quad ,$$

we can construct a table of all the possible alignments of operands:

Case	(0 or 1) Align S	(0 or 1) Align T-F	(0 or 1) Align S-F	(0, 1, or 2) Align F	
1	0	0	0	0	✓
2	0	0	0	1	✓
3	0	0	0	2	✓
4	0	0	1	1	?
5	0	0	1	2	Excluded
6	0	1	1	1	✓
7	0	1	1	2	✓
8	1	1	1	1	✓
9	1	1	1	2	✓

Case 5 is excluded because it is impossible by the preceding paragraph's argument. In all the other cases except 4 the alignments of T-F and S-F agree, so that (S-F) - (T-F) can be computed precisely.

Case 4 requires a finer analysis. It corresponds to the picture



On a machine which discards digits (650) or the guard word (6400), we know the digit X in F will be zero. Otherwise T-F could not be computed precisely, and we know that it is. Therefore the X in S-F will also be zero. When S-F is aligned for subtraction from T-F, the X will be discarded, with no loss of accuracy.

On a machine with a guard word (7090) or digit (360), the same analysis holds! The digit X is not discarded but is in the guard digit. But because we know T-F was computed precisely, and it has no room for the guard digit X, the guard digit must have been 0.

Therefore (S-F) -(T-F) is always precisely computed and we can be sure it is precisely s - t. It will then be added to Y to give an answer correct to a few ulps of Y.

The important thing to remember about the workings of this mystery algorithm in this case is that the expression S-T might not always have a small relative error or be precise, but F has been rigged in such a way that the expression (S-F) -(T-F) is always precisely S-T.

The mysterious constant 0.46, which could perhaps be any number between 0.25 and 0.50, and the fact that the proof requires a consideration of known machine designs, indicate that this algorithm is not an advance in computer science. This sort of devious reasoning is not desirable and should not be necessary. But we can expect to have to do more of the same until hardware designers understand the true costs of their decisions.

Can Using DIF1[†] Make the Answer Worse?

Is it possible, by using the function DIF1, to get an answer that is worse than not having used it at all? This needs to be checked for machines like 7094 and B5500, which have guard words. We must check that if the difference is representable exactly, that's what you get. This code will obviously not work on machines which represent numbers by their logs.

[†]DIF1 is the"magic constant" algorithm.

The essence of working out DIF1 for machines on which it is not needed is that in using this code to make it machine independent the results may be worse than before.*

Question: How did you happen to write this code?

Answer: It was written when we were switching from a 650 to a 7090 using code that had been previously written for a Ferranti-Manchester Mark I, when we were wondering how we'd ever live through all these transitions. So an attempt was made to write code that was machine independent.

Question: I can't see in my mind the sequence that led to your having written such code.

Answer: In my mind was a picture of digits, of course, digit strings as might come from a desk calculator, octal calculator, or a binary machine. The Ferranti-Manchester had to be coded in teletype code -- there was no assembly language so everything had to be visualized quite clearly -- digits had to be input as characters like /, @, Y. It was the practice of visualizing what was going on in the F-M that made it possible for me to see what was going to happen for the various implementations. The tricky part is the .46*T -- who would think to do that?

Question: That was the part I was looking at. I more or less convinced myself that it would work on the 6400 and probably on the 360, for entirely different reasons. I don't call that machine independent.

Answer: The trouble is that it is machine independent in the sense that it is independent of the machines currently on the market.

Question: By a proof that is different for each one?

Answer: Right, and that is very unsatisfactory.

---

*In looking at this code, you will learn a new way to look at numbers, namely as strings of digits.

Question: I don't see any underlying principles.

Answer: The underlying principles are that the digits get pushed off the right hand side of the register but it is not exactly certain what they do as they go.

Unfortunately, the proof for DIF1 is different for each machine. What is annoying is this: The commercial, economic value of machine independent code is so great that people have tried for a long time to find it. And they will continue to try. The time spent on floating point calculations is very small, in most cases. However, that code is normally written by people who, if they write it successfully are a little more educated than many of the other programmers. Therefore, when an error occurs in that code you have a great deal of trouble debugging it unless you find just such a person as wrote it. But such people don't stay around. So the expense of obtaining code like this and transferring it from one machine to another is horrendous.

Question: Isn't it true on that precise argument, that code like this, that looks like FORTRAN, is actually going to be more expensive to transfer to a new machine than code that is explicitly machine code where somebody knows he'll have to go in and rewrite the code?

Answer: What you are saying is, shall we write the code in assembly language with careful documentation to explain exactly what we are doing hoping then, that when we switch to another machine, anyone who reads and understands the documentation will be able to translate into the new assembly language, or should we try to write in machine independent code with some sort of theorem, even an ugly one, that tells us it'll work on almost any machine you can think of. It seems to you that the first

rationale is more sensible. I used to think so to. But I have some code I wrote in 1965 that I can no longer understand, even though it is richly commented. It was written in assembly language and uses every bit of the machine to squeeze every ounce of performance out of its code. Now, even though it was perfectly reasonable and transparent at the time, it would take me several days to once again understand it. That's very expensive.

Question: That is still less expensive than taking this program to a new machine, say that is just being built, and finding out in 6 months that it doesn't work.

Answer: Yes, so I guess my contention would be that machines ought to be better designed. That there ought to be some uniformity in the arithmetic units so arguments of the kind we're having are unnecessary.

Question: Why are manufacturers so unresponsive to your pleas?

Answer: Manufacturers are individuals who are sometimes on the ascendancy politically, and sometimes not. The sales organization is generally on the ascendancy, when the company is on the make. The salesmen have their own particular way of finding out what their customers want. But customers only know part of what they want. So salesmen make rather shallow estimates of what is wanted and present misbegotten specifications to the engineers, who are happy to implement anything. CDC salesmen collected the specs for the 6000 and 7600. One salesman tried to tell me his customers didn't want the machine to round, because he hadn't heard the appendage "the way they were doing it." And you know why. So the salesmen said, they don't have to use the round instruction.

DIF1 should also work in double precision, though there it is harder to see what's going on. Double precision code on the CDC is not a fixed thing but varies from compiler to compiler.

Question: Not only does it vary from compiler to compiler, but one of my friends on the system staff tells me it is incorrect. The guy who wrote the algorithm for the compiler changed certain things in the way it did its double precision multiples, for instance, so that it would go faster.

Answer: That's another reason, for example, for wanting to use a pyramid, since you don't know what kind of double precision has been provided. The pyramid has the property that you know what happens in double precision because it is what happens in single precision.

Why Are Exact Differences Important?

I don't think being able to code double precision is a very desirable thing in itself, though people have worried about it.†

The problem will arise when somebody has used a trick of this sort unknowingly. He has used this trick with a picture in his mind of how machines work. He thinks he has a machine independent code, and he needn't know the details of each machine that he runs his code on.

That's only one problem. That problem collides with another problem -- really a different approach to the same ultimate desideratums: Is numerical analysis a science? Or is it just an art? It was taught to me as an art. My professors did not think of it as mathematics. To think of

†Dekker and Sterbenz show that if the single precision arithmetic satisfied certain rules -- the essential rule is that if you compute a difference you get it to within a unit in the last place, or thereabouts, and precisely if it can be represented precisely -- then you could get double precision. The double precision they get doesn't satisfy that rule, but with a little extra work you can clean it up. Then the double precision would look like what you would have on a certain kind of machine, in which its single precision was what you had just coded to be double precision. Then you could pyramid this.

it that way is really a step backwards, since all the old classical mathematicians, Euler, the Bernoulli brothers, Lagrange, LaGuerre, thought of mathematics and numerical analysis as practically synonymous. They didn't distinguish the two. But subsequent mathematicians did. Mathematics was regarded as a simplification of real life and numerical analysis was a compromise.

Numerical Analysis vs. Mathematics

If we cannot prove anything about numerical analysis, then we run the risk of never being able to prove anything worth knowing about computers. You must distinguish between numerical analysis, and mathematics, in which there are infinite processes and in which things are alleged to converge. Until you start to discuss rounding errors as they really are, and under/overflow as they really are, you don't have numerical analysis. van Wijngaarden thinks this way too; in 1966 he published a paper "Numerical Analysis as an Independent Science," BIT 6, 66-81. Knuth has condensed van Wijngaarden's many pages into about a page.† Knuth's approach is to say let's discuss what

†van Wijngaarden has numerous rules for entities which would be represented in the machine and would be intended to represent real numbers. These are to supplant the rules we learned about real numbers. But since most people don't understand this much smaller set of rules, what are the chances of re-educating people with van Wijngaarden's? He tries to skirt around another problem, that of using one symbol to mean different things. He would like his rules to hold if you replace each number by a set of numbers that differ only by a few units in the last place. You should make only those statements that will remain valid if the operands are perturbed before the calculation is done. Questions like, are two numbers equal, he thought you ought not to ask. You ask if they are equal to within a tolerance, which is tantamount to saying that the equal sign represents an operation; you perform this operation upon two operands and the result must be independent of what you would get had you perturbed the operands by at most a few units in the last place. Two numbers may be equal, to within a tolerance, or definitely different to within that tolerance, with some borderline areas in between. Knuth discusses these notions. He has $a = b$, $a \approx b$ (a almost equal to b), and $a \sim b$ (a not quite as equal to b as that). Philosophers and other people would sum this up by saying -- if you do that you will be unable to say that you mean or to mean what you say. See Knuth, Seminumerical Algorithms, 199, 1969.

is a reasonable model of what is done in a computer (or could be) as merely a small distortion of what would happen in the world of real numbers.

$$A + B = a + b \quad \text{which gets rounded}$$

That is a very simple rule. Unfortunately computers don't obey it. But Knuth does have the beginnings of a science. Knuth has compromised a bit, as we could go further and describe all operations in terms of integers. Knuth has done this, by writing programs for MIX, an integer machine. He says these will be the definitions of the floating point operations.†

So there you see the two problems converging. One is -- can you write machine independent code. The other is -- can you think of numerical analysis as a science. If you claim to have machine independent code it means you have proved something about it which is tantamount to proving a theorem about an algorithm, and that's the type of thing we want to do in numerical analysis.

A Third Consideration: How Much Precision?

There is really a third leg on this stool. If you lose any one of the three, the stool will fall over. The third leg is this: can you prove theorems about numerical analysis comparable to theorems in computational complexity, but bearing instead on how much precision you have to carry to do a certain job. There are theorems about how long it takes to do

---

† Anything that is not implied fully by the rule above will have to be ferreted out by looking at the integer manipulation. You have to look at exactly what is that rounding rule and exactly what is the base of your machine. Knuth says he doesn't care what the base is -- a byte can hold 64 or 100 possibilities. In my experience that is a disaster. All sorts of ugly things happen to non-binary machines.

operations -- say multiply two numbers together.†

There's a theory due to Belaga-Pan (also in Knuth) which tells you that if you have an $n^{\underline{th}}$ degree polynomial, you can, by rearranging it in various subtle ways, reduce the number of multiplications by a factor near two. The theory doesn't tell you how many digits you will need to carry, however. How long does it take to multiply two matrices together? The conventional way requires n^3 (for $n \times n$ matrices) multiplications. Winograd showed that roughly half that many will do. Strassen has shown that actually it is $n^{2.xxx}$ where the exponent is $\log_2 7$ instead of $\log_2 8$. Other theorems say how much storage you need to do something. But there is a shocking lack of theorems that tell you how accurately you have to do something. These theorems don't exist because in principle you could code multiple precision using only single precision arithmetic. And that's the explanation for wanting to do some of this, just to demonstrate that if you had to do it by brute force, you could do this coding, and it would be to some extent machine independent. In the absence of definitive rules on how machines work, it is hard to say just what that code should be like.

---

†To add two numbers of length n in a machine whose components only have a certain complexity takes on the order of $\log n$. Machines currently do run close to the optimum, which is nice. A lower bounded for multiplying is similar, but there are no algorithms that really come close; usually $n \log n$ is more realistic. So there is room to improve multipliers. Or the lower bound.

11. HOW MUCH PRECISION DO YOU NEED IN GENERAL?

Beyond the problems of hardware and software flaws looms the larger question of how little precision can be carried and still yield a desired accuracy in the result. Current thinking is that this question is likely to be refractory in the foreseeable future.

T.J. Dekker (in Numerische Mathematik 18, #3, 1971) demonstrates how to use single precision floating point hardware to compute double precision addition, multiplication, division, and square root. His code is weakly machine dependent in so far as it requires the base and word length. But we know that we can write machine independent code to determine these parameters. His double precision addition is similar to our algorithm. Multiplication is based on splitting each operand into two parts. He must assume, however, that the floating point units give correctly rounded results or something close to them. His algorithm will work on a GE 635 with proper software but not on most machines in general use, such as the 360.

A more serious problem is that the hardware commonly built for double precision does not satisfy as good a model of arithmetic as that for single precision on the same machine. The GE 635 comes close because extra digits beyond the double word are included in the arithmetic registers. On most other machines, such as the B5500 or IBM 7094, the hardware is basically double precision. The good model satisfied by the single precision instructions is a consequence of the double registers being already present in the arithmetic units (or we could look on the double precision facility as a cheap bonus for doing the single precision properly). But there is no guard digit readily available for double precision arithmetic. It would have to be built in, and it generally is not.

We would like to be able to program quadruple precision on double

precision machines by the same trick Dekker uses to get double precision on single precision machines. Unfortunately no machine comes close enough in double precision to the model "round the precise result" for this to be possible. The problems caused by this fact are discussed in Kuki and Ascoly, "Fortran Extended-Precision Library," IBM Systems Journal 10, 1971, p. 39. After the design of the 360/85, it was desired to simulate the double-long word arithmetic of the 85 on other 360 models which had only long word arithmetic registers. Because of the shortcomings of the long word arithmetic, it was extremely difficult to simulate double-long, particularly in division. It seems, however, that a rational design of double precision hardware, to give always the correctly rounded result or something very close to it, could be achieved.

Students' Report: Higher Precision Out of Single Precision

After a number of false starts we concluded we could reasonably go about producing double precision from single precision by the following strategy:

6400 floating point	$\xrightarrow{\text{DIFF}^{\dagger}}$	dirty single precision
(start here; it is about the worst around)	(exact subtraction routine)	(much like 6400 floating point but without cancellation problems; has error of < 1 ulp of the result)

We thought this dirty single precision was a good place to start. The technique published by Dekker which looked most promising for constructing double precision out of single precision led to a double precision that was off by a few ulps of double precision. Then we'd have to have a technique to turn dirty double precision into clean almost-double precision.

†DIFF uses logic like that in DIF1 [10] to determine the difference between two single precision numbers with minimal error.

Given this technique, we'd first apply it to the dirty single precision to get clean single precision, since the Dekker method started with clean single precision.

Continuing from above:

dirty single precision → clean single precision (correctly rounded)

$\xrightarrow[\text{method}]{\text{Dekker}}$ dirty double precision $\xrightarrow[\text{as above}]{\text{same technique}}$ clean, almost double precision

(The best to clean it up is to throw away a few bits and get say a 93 bit result to get arithmetic with same characteristics as the machine.)

(Then you start all over again.)

Perhaps by another technique we can produce clean, truly double precision, but we didn't think of it until too late to try it; it would be a nicer result.

Machine Single Precision → Dirty Single Precision

This is done using the DIFF subroutine. It is this step that has to be machine independent, in that it has to work if the machine rounds, chops, normalizes or not in its arithmetic. Once you have the dirty single precision, only the base of the machine and the number of digits carried is important.

To do this, we compare the sign of two numbers to be added. If the signs were the same, we simply used the machine's arithmetic. If the signs were opposite, we ordered them by magnitude and used the DIFF routine.

Question: But if you use 6400 arithmetic to compare two magnitudes, they can come out equal when they are not.

Answer: Yes, we forgot about that.

Question: Can't you just feed the arguments to DIFF and see if you get a positive or negative answer?

Answer: No, if you reverse the arguments sent to DIFF, you don't get the right result.

So we still have this problem of the comparison.

Kahan: If you have two numbers that machine arithmetic says are equal but you suspect are not, you could send them in both orders to DIFF. If they are equal, both results will be equal. If they are not equal, I think you'll get zero in one case and the correct result in the other.

This is the crucial problem. If when you feed two numbers to the arithmetic unit it has the privilege of muddying them by as much as an ulp before it does anything, then you can't make delicate comparisons. You can't even talk about a number because the arithmetic unit is talking about a different one, and it won't tell you which one.

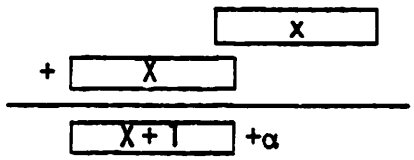
Dirty → Clean Single Precision

This is accomplished by a trick which works, regretably, only for binary machines.

We use a procedure that, we think, takes two single precision numbers into their double precision sum.

you get two halves

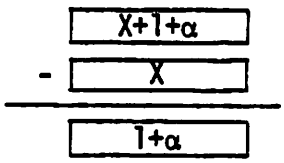
You look at the least significant half and see if it is more than half a unit in the last place of the more significant half. If it is, you correct the more significant half. The easiest way seemed to be to have a way to construct a number that is a unit in the last place of a given number. The scheme goes like this:



$\alpha = 1, 0$

X is x shifted n-1 bits left, where n is the number of bits in the word. α depends on how the machine feels about rounding this operation.

α can't be more than 1, since we assume the error is not more than 1 ulp. Since the numbers have the same sign, the number cannot come out too low. Now subtract off X:



using DIFF (in case there was a carry and the exponents are different)

$1+\alpha$ lines up with the top of the original number x. Because the machine is binary $1+\alpha$ can only have two possible values, one of which is bigger than the original number (the 1 digit is the leading position of x, so we either have 1 in the leading digit ($\alpha = 0$) or 2 in the leading digit ($\alpha = 1$)). On non-binary machines, you would know that that first digit was a 1, only that it was nonzero in the leading digit of x. For an arbitrary base machine, you'd get $b+\alpha$, where $1 \leq b <$ base of machine.

Once you've determined $\alpha = 1$, you want to get rid of it. You can't just subtract 1, because you don't know where to put it (if you did you'd have solved the problem). But in binary, it is very easy; you just divide by two if $\alpha = 1$.

So we have a unit in the last place of a number.

Question: What if the division algorithm of the machine is wrong?

Answer: Actually, we multiply by 0.5.

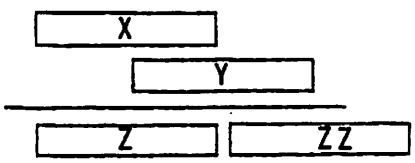
Question: But what if multiplication is funny? The old 360 way of multiplying lost the bottom digit of single precision, if a normalizing

shift was necessary, because there was no guard digit.

Answer: We are assuming that if there are only a couple bits in each number, the multiplication will be exact. In constructing X, we are multiplying by a power of 2 and it is reasonable to expect the multiplication to work if one operand only has one bit. But even if X is in error, it doesn't matter because you subtract it off again. All that really matters is that X have the correct characteristic.

Single + Single → Double Sum

|X| > |Y|; X and Y are single precision and the arithmetic is dirty but within an ulp of what you want. We are adding X and Y and want to come out with two single precision numbers Z and ZZ, such that if you do an infinite precision add of Z and ZZ, you get exactly the infinite precision sum X+Y. We'll go through the argument assuming clean arithmetic, then consider the cases where dirty arithmetic makes a difference.



The Algorithm To Do This

```
SSDADD(X,Y,Z)
DIMENSION Z(2)
Z(1) = X+Y          (may be rounded, so correction may need to be negative)
Z(2) = Y-(Z(1)-X)   |X| > |Y|
```

Z(1)-X will be done precisely. Since the error is less than 1 ulp and because both Z(1) and Z(2) are representable, they must both be exact.

In the case of dirty arithmetic, you get in trouble if sign(Y) ≠ sign(X), and $|Y| < \frac{1}{2}$ ulp of X. So we test in the routine for this condition (using the bit extraction routine), and if it holds, return X as Z(1) and Y as Z(2), even if Y is very much smaller than X.

We use this to get clean single precision and to know how to round correctly we do need the exact answer. This routine is also used in the dirty double precision add routine and there we have good arithmetic already.

Kahan: I'm not convinced you've got this working, or that you're so close that I'd accept it as feasible. There are still lots of holes.

If you could work out not just the program but the way of understanding arithmetic so that you could write programs like this without great agony, it would then be possible to take a program that worked on any machine and if it had been written in such a way that, somewhere, the code on that machine had been designed to simulate first a standard machine using rounded arithmetic, you would then be able to use that code, if you too could simulate the same standard arithmetic. In principle, code conversion would be accomplished by considering the algorithm that converts one kind of code into another and imbedding it in your conversion procedure. People can't do this yet without making the conversion routines very machine dependent. You're trying to solve the problem of designing the conversion routines to change any machine's arithmetic into some standard arithmetic on which you base transportable programs.

How Much Precision Do We Need?

We might come to the conclusion that multiple precision is something everyone wants but no one wants to pay for. The sales of the 360/85 were poor, possibly because of other factors such as I/O mismatch with the CPU.

We rarely see exactly what the benefits and costs of double precision hardware are, because other improvements are usually made in the machine at the same time. So what follows is in the nature of opinion.

Whatever precision is supported as standard, and this may be the so-called double precision at most 360 installations, there should be a cheap way of getting double the standard precision, preferably in hardware. A little judicious use of double precision makes it possible to guarantee good results in single precision, and more importantly to dispense with much of the error analysis that must be done when the calculation is carried out in single precision. In the quadratic, double precision enables us to guarantee answers correct to a few units in the last place of single precision and we need endure little error analysis. In matrix calculations, we can expect that double precision accumulation of scalar products will reduce the effect of rounding error to well below the uncertainty in the data.

So double precision is useful, but is it worth what it costs? We can practically say that the cost of double precision hardware is so small in the total system that we can disregard it. But if double is good, is quadruple precision better? It turns out that demands on precision taper off very rapidly for almost all technological users. Most orbit computations when done rather crudely, require at most 100 bits to give satisfactory results during the lifetime of most artificial satellites. Indeed, no physical constants are known to more than about 18 significant figures (60 bits).

The situation for mathematical calculations is rather different. For any precision a calculation can be specified which requires that precision to yield an answer with a single significant digit. Where cancellation is very severe, as in the evaluation of integrals of oscillatory functions,

arbitrarily large precision is required, and how large can't be predicted in advance. We can safely assume that if we do these computations and the result is inadequate, we can increase precision and do them over, and the cost of the preceding runs with lesser precision will be utterly negligible compared to the cost of the current run. Since the amount of precision is unpredictable, software seems to be required.

Actually, really high precision spends so little time on exponent manipulation that it is really more in the realm of integer arithmetic and is outside the scope of this course.

In contrast to the preceding, it is very common to want just a little bit more precision. Let's examine a subroutine to compute a transcendental function. These functions are commonly approximated by rational functions such as

$$\frac{a_0x^3 + a_1x^2 + a_2x + a_3}{x^3 + b_1x^2 + b_3x + b_4}$$

Evaluation of this function requires five multiplies and one divide. An equally close approximation can be had from a function of the form

$$\alpha_1 + \cfrac{\beta_1}{x + \alpha_2 + \cfrac{\beta_2}{x + \alpha_3 + \cfrac{\beta_3}{x + \alpha_4}}}$$

which only requires three divides.

Although the latter expression can be evaluated more quickly, we have more trouble from cancellation. Therefore we would like to have a few more bits in order to use the second method. If we don't have them we may have to go to a good deal of trouble to get our function accurate to our working

precision.

Almost all the elementary Fortran functions get into trouble for precisely this reason. Consider A**B, which is usually computed as exp(B log A). Suppose $A \doteq 1$ and B is huge so A^B is also huge. Then log A and B will, say, at best be precise in single precision, so their product is accurate to 1 ulp in single precision. Now the logs and exps are generally with respect to the base of the machine. The integer part of B log A will be removed and used as the characteristic of A^B. If B log A is large, the mantissa may have few significant figures. But it is solely the mantissa which determines the significant figures in the final result. Consequently the final result may be accurate to much less than single precision. Clearly we need extra digits, equal in number to the number of digits that could be occupied by the integer part of a logarithm of the largest number representable on the machine. Then we can guarantee that A**B will be correct to a few ulps in single precision.

What is the meaning of all of these considerations for the hardware designer? He must understand the level of accuracy his users will want. In order to get single precision accuracy the user will need, if not complete double precision, something close to it.

Double precise products and sums and differences of single precision operands have to be developed anyway. They might as well be convenient for the user to access. Quotients are more difficult but at least a remainder should be supplied, as on an old mechanical desk calculator!

The organization of the 7094 was similar to what we have sketched. We could even ask for a bit more than double precision in the accumulator, as in the GE 635. Unfortunately there was no single instruction for storing the extra bits.

The machine designer who has put the extra bits in may now be amused to discover that the language designer has made it difficult to use the extra bits in higher languages. In most theories of types, "real" and "double" are completely distinct entities which happen to have rules for converting between them. The concept of using a few bits of double precision in a single precision operation has yet to be incorporated into such theories.

Seemingly we must design the hardware, the language, the compiler, and the operating system (to handle overflow, etc.) together from the ground up! We will have to leave the reader confronted by this grim prospect.

12. INTERVAL ARITHMETIC

We have by now seen enough to be ready to avoid error analysis whenever possible. Certainly users ought not to have to do error analysis. As computer scientists it behooves us to investigate whether it can be done automatically to avoid the staggering cost of manual error analysis.

The first step in this direction was called significance arithmetic. Basically, machine numbers were allowed to be unnormalized, the number of zeros on the left indicating the uncertainty in the number:

$$\boxed{2^0 \mid .100000} \sim \frac{1}{2} \pm 2^{-t-1}$$

$$\boxed{2^3 \mid .000100} \sim \frac{1}{2} \pm 2^3 \cdot 2^{-t-1}$$

Then each word actually represents an interval.

We have to construct rules for dealing with such intervals. The rules were worked out by N. Metropolis and R. Ashenhurst. There is a choice between intervals that are possibly wider than the desired interval, and intervals that are possibly narrower, because most intervals can't be represented precisely in significance arithmetic, e.g. in 3 significant decimal arithmetic, $(02.0 \pm .05) \times (05.0 \pm .05) = 010.\ ^{+.3525}_{-.3875}$; should we substitute $010. \pm .5$ which is too wide, or $10.0 \pm .05$ which is too narrow? The optimistic point of view is to choose an interval that is sometimes slightly narrower than the most appropriate interval. Examples can be constructed where this policy will give no hint that dreadful errors have occurred. The pessimistic approach is to take an interval slightly wider than the most appropriate one. Then you can get error bounds so unrealistically bad that they are ignored.

Interval arithmetic contains all the intervals of significance arithmetic plus many more, and so is more powerful and flexible. We are familiar with intervals from mathematics:

$$\text{if } a \leq b, \quad [a,b] = \{x: a \leq x \leq b\}$$
$$\text{if } a < b, \quad (a,b) = \{x: a < x < b\}$$

Arithmetic on intervals is based on the idea of a Minkowski sum:

$$[a,b] + [c,d] = \{x+y: x \in [a,b],\ y \in [c,d]\} = [a+c,b+d] \quad ,$$

and in general

$$A \text{ op } B \equiv \{a \text{ op } b: a \in A \text{ and } b \in B\}$$

where we use capital letters for intervals. In all cases we only need to know the endpoints of the operand intervals to get the endpoints of the result intervals.

Naturally the first question to arise is how to deal with round-off. When an endpoint of the result interval is not precisely representable we widen the interval as little as possible to the next machine number, so that we take a pessimistic point of view, but less pessimistic in general than for significance arithmetic.

We can see that the following operations are going to cause problems (using 5 sig. dec. arithmetic):

Good	Bad
$[2,2]\cdot[2,2] = [4,4]$	$[2,2]\cdot[2,2] = [3.9999,4.0001]$
$[1,1] + [-10^{-30},10^{-30}] = [.99999,1.0001]$	$[1,1] + [-10^{-30},10^{-30}] = [1,1]$
	or $[.99999.1]$
	or $[1,1.0001]$

It's pretty hard to get software that will get the good result in both computations. We would seem to need hardware that offers the choice of rounding to the left or to the right, under program control. Such hardware is not usually available so most interval arithmetic generates an unnecessary spread on numbers that should be exact.

Triplex arithmetic was invented to ameliorate this problem and to economize on storage. Intervals are represented by midpoint and a spread, as the following correspondence shows:

$$[a,b] \sim \{\frac{a+b}{2}, \frac{a-b}{2}\}$$

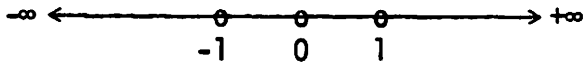
$$[x-\delta, x+\delta] \sim \{x, \delta\}$$

This system works well on small intervals but poorly on large intervals. If $x \doteq \delta$ the small end of the interval can't be represented very well because of cancellation. Then if we take the reciprocal of such an interval it becomes rather uncertain. Generally interval arithmetic is more effective. The computation is the same because the endpoints of result intervals must be computed the same way in either case. The advantage of triplex is if the intervals are all small, then less storage may be required for δ than for x.

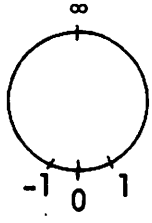
To secure this storage advantage you must give up something important of a practical nature. We would like to use the built-in two word double or complex handling facilities of standard Fortran compilers to implement interval arithmetic. Then we could avoid rewriting much of the Fortran compiler.

There is one more problem with intervals in general, and that involves reciprocals. Certainly $1/[1,2] = [\frac{1}{2},1]$. But what about $1/[-1,1]$? This would be $(\infty,-1] \cup [1,\infty)$. We can handily write this as $[1,-1]$. That is,

we interpret $[a,b]$ as $(\infty,b] \cup [a,\infty)$ when $a > b$. Then the system of intervals is closed under rational operations. This amounts to discarding our conventional view of the real numbers in favor of a circle with one



infinity (a subset of the projection of the complex plane onto a sphere):



We will need some new symbols. For the interval containing all the extended real numbers we have

$$\Omega = [-\infty,+\infty] \quad .$$

For the point ∞ we have

$$\infty = [\infty,\infty] \quad .$$

Then we have an indefinite situation for $\Phi = [+\infty,-\infty]$, which is not a valid interval.

Clearly we will need a machine representation for infinity. These conventions will enable us to avoid the nuisance of most of the usual implementations of interval arithmetic which lack the complementary intervals containing ∞.

We see that our definition of interval operations makes a <u>closed</u> system when exterior intervals such as $[1,-1]$ are included, provided we make proper

conventions concerning ∞ and Ω.

Some of the axioms of normal arithmetic are not preserved in interval arithmetic. For instance, only a sub-distributive law holds:

$$A \cdot (B+C) \subseteq A \cdot B + A \cdot C \quad .$$

Likewise

$$\frac{B+C}{A} \subseteq \frac{B}{A} + \frac{C}{A} \quad .$$

Equality is occasionally achieved in these laws. In the first case, when

(1) A is a real number $[a,a]$

or (2) $B \cdot C \not\subseteq [\infty,0]$ and $\infty \notin A$

then $A \cdot (B+C) = A \cdot B + A \cdot C$ (exercises for student).

There is also a kind of sub-cancellation. That is,

$$\frac{(A\ B)}{(C\ B)} \supseteq \frac{A}{C} \quad \text{and} \quad (A-B) - (C-B) \supseteq A - C \quad .$$

In both cases equality is achieved when B is a real number $[b,b]$.

Different theorems have to be discovered and applied to interval arithmetic. There is, for instance, an inclusion monotonicity theorem:

> If $A \subseteq X$ and $B \subseteq Y$ then $A \circledast B \subseteq X \circledast Y$ for any operation $\circledast \in \{+,-,/,*\}$.

We also have to replace the total ordering of real numbers by a partial ordering of intervals. It is difficult to formulate a satisfactory ordering of overlapping intervals, or with any exterior interval.

More about interval arithmetic may be found in:

E.R. Hansen, ed., Topics in Interval Analysis, Oxford University Press, 1969.

W. Kahan, Notes for University of Michigan Summer Course, 1968.

R. Moore, Interval Analysis, Prentice Hall, 1966.

K. Nickel, "Error Bounds and Computer Arithmetic," IFIP 68 Proceedings, pp. 54-62.

Interval Arithmetic Functions

We have seen that interval arithmetic possesses a number of peculiar properties. One systematic approach would be to invent a new kind of algebraic structure having these properties, and then deducing theorems about such structures in general which of course would apply to interval arithmetic. Such an approach would, indeed, be looked upon with favor in many mathematical circles, but our purposes will be better served now by turning to an examination of the problems involved in defining functions whose domain and range are intervals.

Let us start with the simple set of functions $f(A) = A^n$ for n a positive integer. We could define $A^n = A \cdot A \cdot A \cdots A$ n times. This causes an unrealistic large spread in the size of the interval. If we define instead

$$A^n \equiv \{a^n : a \in A\}$$

we find that

$$A^n \subseteq A \cdot A \cdot A \cdots A \quad .$$

If $A = [-1,1]$ and $n = 2$ we find $A^2 = [0,1]$ and $A \cdot A = [-1,1]$.

We are going to have to differentiate between functions and the expressions for computing them. For real numbers the difference is rarely significant mathematically. For intervals the situation is totally different. Consider the three expressions

$$E_1 = \frac{x \cdot (x-y)(x+y)}{x^2 + y^2} \qquad E_2 = \frac{x(x-y)(x+y)}{x \cdot x + y \cdot y} \qquad E_3 = x\left(1 - \frac{2}{1 + (\frac{x}{y})^2}\right)$$

These expressions are all representation for the same rational function of scalar variables. The last is best in the case $x = y = 0$ because, as $x \to 0$, $E_3 \to 0$ regardless of how y behaves.

But now suppose $x \leftarrow [0,0]$ and $y \leftarrow [0,0]$. Then $E_1 = E_2 = \Omega$. If we evaluate $(\frac{x}{y})^2$ as $(\frac{x}{y})(\frac{x}{y})$ we get $E_3 = \Omega$. But if we do it correctly, we compute

$$\frac{x}{y} = \frac{[0,0]}{[0,0]} = \Omega$$

$$(\frac{x}{y})^2 = \{z^2 : z \in (\frac{x}{y})\} = [0,+\infty]$$

$$1 + (\frac{x}{y})^2 = [1,\infty]$$

$$\frac{2}{1+(\frac{x}{y})^2} = [0,2]$$

$$1 - \frac{2}{1+(\frac{x}{y})^2} = [-1,1]$$

$$E_3 = [0,0]\cdot[-1,1] = [0,0]$$

The single point $[0,0]$ is certainly an improvement over Ω.

This example demonstrates that it is pretty tricky to define interval functions except by stating their interval expressions. Then interval expressions which would seem equivalent in scalar arithmetic will often define different functions in interval arithmetic, and moreover it is often difficult to determine whether two interval expressions define the same function. To discuss these issues systematically we will write scalar functions as

$$f(x_1;x_2;\ldots;x_n) \quad .$$

We identify the rational expression easily with the function. Then we define

$$F(X_1;X_2;\ldots;X_n)$$

as what we get when we substitute in the expression for f the intervals X_i for the variables x_i. We also need to define the range of the function f as

$$Rf(X_1;X_2;\ldots;X_n) \equiv \bigcup_{x_i \in X_i} f(x_1;x_2;\ldots;x_n) \quad .$$

Now we have a theorem generalizing our previous monotonicity result:

$$Rf(X_1;X_2;\ldots;X_n) \subseteq F(X_1;X_2;\ldots;X_n) \quad .$$

The strength of this theorem is that it is independent of the expression F used for f.

For rational functions we can prove the theorem by induction. Then other functions of intervals are defined to satisfy this theorem.

The Independence Phenomenon

What we would like to do is always compute with the expression that is equal to the range of the function for every argument. Then our intervals will not expand unnecessarily. Such an expression sometimes exists, but sometimes does not.

An example where a solution exists is in the case where

$$\phi(\xi) = \frac{\xi}{\xi+2} = \frac{1}{1+\frac{2}{\xi}} \quad .$$

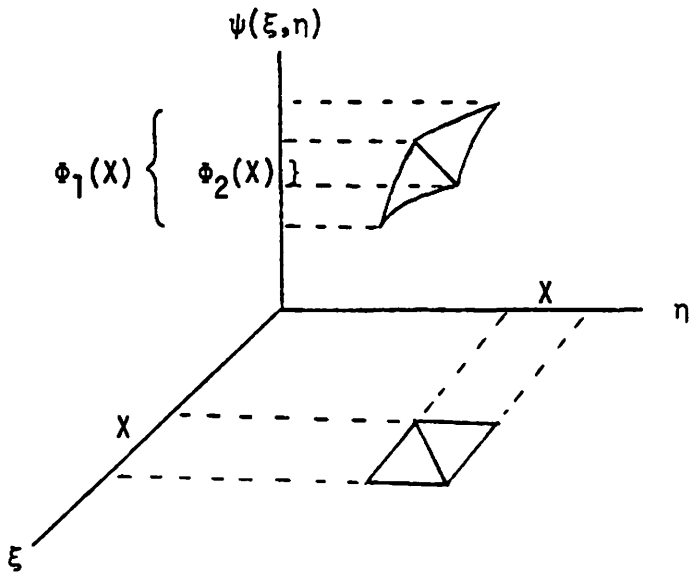
Then $\Phi_1(X) = \frac{X}{X+2}$, $\Phi_2(X) = \frac{1}{1+\frac{2}{X}}$. These interval functions are distinctly different and

$$R\phi(X) = \Phi_2(X) \subseteq \Phi_1(X) \quad .$$

The variable ξ occurs in the second expression only once. When we evaluate

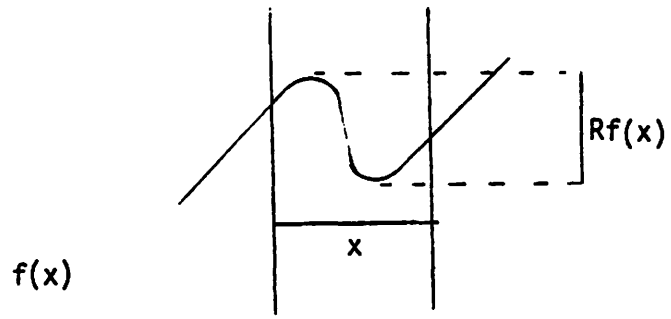
$\Phi_1(X)$ we actually evaluate $\psi(X,X)$ where $\psi(X,Y) = \frac{X}{Y+2}$. That is, the two occurrences of ξ are independent, so that $\psi(\xi,\eta) = \frac{\xi}{\eta+2}$. This is called the Independence Phenomenon.

We can visualize what is happening as something like:



The square domain in the $\xi\eta$ plane represents all the values that could be used to compute $\psi(\xi,\eta)$ with $\xi \in X$, $\eta \in X$. The values of the function ψ as it ranges over this domain are represented by the floating curved surface. The projection of this surface on the ψ axis is $\Phi_1(X)$. The line segment subset of the square domain is the set of values that ξ and η could actually obtain -- namely those where $\xi = \eta$. Their set of values $\psi(\xi,\xi)$ is a line segment subset of the curved surface. The projection $\Phi_2(X)$ of $\psi(\xi,\xi)$ is a subset of the projection $\Phi_1(X)$ of $\psi(\xi,\eta)$.

There are other problems besides the independence phenomenon. Remember we are trying to find a rational expression which gives the same result as the range of the function we are trying to compute. Suppose, for instance, we have a cubic polynomial on an interval in which the maximum and minimum are defined by the derivative vanishing rather than the endpoints:



Let us suppose this cubic has integer coefficients. The co-ordinates x, such that $f'(x) = 0$, are defined by solving a quadratic equation with two real roots. In general these roots involve surds and are irrational. Therefore $f(x)$ at these points will also be irrational numbers. That is, $Rf(X)$ has irrational endpoints. In general no rational expression F can yield the correct interval.

That is not to say that we can't come arbitrarily close. We can chop the interval X into a number of parts. Let

$$X = \bigcup_1^N X_j \quad .$$

Then

$$Rf(X) = \bigcup_j Rf(X_j) \subseteq \bigcup_j F(X_j) \quad .$$

We claim that if the function is reasonable and the X_j are chosen small enough, say a few ulps wide, then F can be chosen so that $\cup F(X_j)$ is just a few ulps wider than $Rf(\cup X_j)$.

We will outline a justification for such a claim. In general, a function $f(x_1;x_2;\ldots;x_n)$ has many corresponding expressions $F(x_1;x_2;\ldots;x_n)$. We want to find an expression where each x_i appears only once. If it appears more than once, give it several different names and increase n. We will consider the ramifications of this later.

Now we would like to assume that f is differentiable and that the

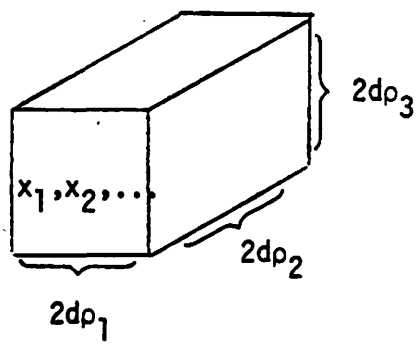
intervals X_j are so small that they are infinitesimal, which means

$$X_j = \{x_j + dx_j : |dx_j| \leq d\rho_j\} \quad .$$

Then we find that

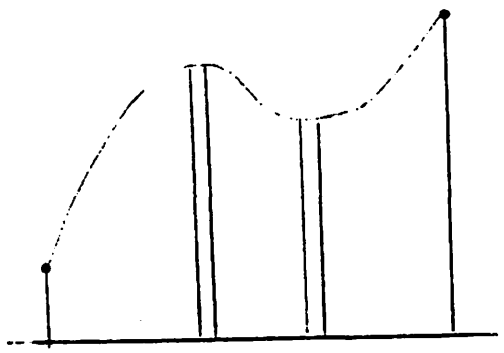
$$Rf(X_1;\ldots;X_n) = f(x_1;x_2;\ldots;x_n) + \bigcup_{|dx_j| \leq d\rho_j} \sum_j \frac{\partial f}{\partial x_j} dx_j \quad .$$

We have in mind a box as the domain of f:



The point $x_1;x_2;\ldots;x_n$ is centered in the box, a rectangular parallelopiped. Then Rf consists of all the values f can take over the box domain. Clearly, then, for any large rectangular parallelopiped domain, we can break it up into many infinitesimal boxes. Each box determines an infinitesimal interval of the range of f. The union of all these infinitesimal ranges is infinitesimally close to the range of f.

Clearly, part of the technique of interval arithmetic is the division of intervals into smaller parts for analysis. We don't want to do this any more than necessary. We will take advantage of monotonicity of functions wherever possible so we can just take the values at their endpoints:



In this example we have three monotonic intervals and two uncertain ones.

Since we replaced multiple appearances of a variable with several independent variables ranging over the same interval in an uncorrelated fashion, we should investigate what widening effect this has upon the intervals we compute. As an example we could have

$$\phi(x) = \frac{x}{2+x} = f(x,x) \ , \quad \text{where} \quad f(x,y) = \frac{x}{2+y} \quad .$$

Then the range

$$R\phi(X) \subseteq \Phi(X) = \frac{X}{2+X} \quad .$$

In general the interval on the right is wider than the interval on the left.

To start to see why this is so, recall that the statement

$$Rf(X,Y) = F(X,Y) = \frac{X}{2+Y}$$

is true if X and Y are actually independent variables. But

$$Rf(X,X) \subseteq F(X,X) \quad .$$

We have seen that in special cases we can rewrite ϕ so that equality holds.

In general a rewriting is not practical or possible so we need to see how much wider the intervals can become. Suppose X is an infinitesimal interval:

$$X = x + [-d\rho, d\rho] \quad .$$

Then when we can perform a Taylor expansion of $f(x,y)$ in each variable at the point (x,x), we get

$$F(X,X) = f(x,x) + \left(\left|\frac{\partial f}{\partial x}(x,x)\right| + \left|\frac{\partial f}{\partial y}(x,x)\right|\right)[-d\rho, d\rho] \quad .$$

Note that the expansion is performed before we substitute x for y.

We want to compare this with

$$R\phi(X) = \phi(x) + \left|\frac{d\phi}{dx}\right|[-d\rho, d\rho] \quad .$$

Since $\frac{d\phi}{dx} = \left(\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\right)(x,x)$ we can see that

$$\frac{d\phi}{dx} \leq \left|\frac{\partial f}{\partial x}(x,x)\right| + \left|\frac{\partial f}{\partial y}(x,x)\right|$$

so that $R\phi(X) \subseteq F(X,X)$. In the particular example $f(x,y) = \frac{x}{2+y}$, $\frac{\partial f}{\partial x} \geq 0$ and $\frac{\partial f}{\partial y} \leq 0$ so we would expect some cancellation if x and y were properly correlated. This analysis shows why distributive laws and cancellation laws fail: certain interval variables appear twice and are not properly correlated.

To cure the problem requires symbolic analysis which can't be made routine. Sometimes monotonicity properties help, so that we can evaluate the interval function by evaluating the scalar function at the endpoints. Sometimes the extrema are useful. One suggestion has been to transform the function with a midpoint expansion, as follows.

Suppose, then, that we want to evaluate $R\phi(X)$ for some $\phi(x)$. We could let

$$X = x_o + [-\Delta, \Delta] \quad .$$

Then we could consider the Taylor series or the divided difference expression,

In the later case

$$\phi(x) = \phi(x_0) + \Delta\phi(x,x_0)(x-x_0) \quad .$$

Suppose we can do the division in the divided difference explicitly [1]. Then we can write

$$\Phi(X) = \phi(x_0) + \Delta\Phi(x_0 + [-\Delta,\Delta],x_0)\cdot[-\Delta,\Delta] \quad .$$

Then the width of the interval in the range is some multiple of the width of the interval in the domain. We hopefully can find an expression for $\Delta\Phi$ which computes a narrow interval for a narrow interval argument. There are theorems which indicate when the interval obtained from $\Phi(X)$ above is tighter than that obtained from

$$\Phi(X) = \phi(X) \quad .$$

If ϕ is, for instance, already a divided difference we can expect trouble from this scheme if we can't do the division symbolically; then we might divide by an interval containing zero. Therefore we need a sophisticated symbolic manipulator at execution time. One of the better implementations of interval arithmetic was at the University of Wisconsin by Moore, and it contained a symbolic manipulator. He wanted to get error bounds for systems of differential equations. To get the advantages of interval arithmetic he had to limit himself to differential equations that can be expressed in terms of rational functions. Then the computer would symbolically differentiate the rational functions, not to get better interval arithmetic but because the partial derivatives were needed to compute the error bounds. His differential equation solver never worked properly, however, as it gave utterly pessimistic bounds.

What Can You Do With Interval Arithmetic?

No one else has gotten good results from interval arithmetic. There have been some results by the group at Karlsruhe reported by Nickel. They are rather handicapped by lack of a symbolic manipulator.

Let us see what kind of theorem they can prove. In particular, consider the solution of some equation $f(x) = 0$ by Newton's method. We want to find ξ such that $f(\xi) = 0$, and we assume that such a $\xi \in X_0$, our initial interval. Then, with $x_0 \in X_0$ as a start, let

$$X_1 = x_0 - \frac{F(x_0)}{F'(X_0)}, \quad x_1 = x_0 - \frac{f(x_0)}{F'(x_0)} \in X_1 \quad .$$

Here F is the expression one gets by simple substitution of the degenerate interval for x_0 in the expression for $f(x)$, $F(x_0)$ would be a scalar except that rounding errors will be incorporated into the computed intervals, widening them into non-degenerate intervals. $F'(X_0)$ is obtained by substituting the interval X_0 into the expression for $f'(x)$.

What could we say about such an algorithm? The usual Newton method yields $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$. Then $x_1 \in X_1$, even with rounding errors taken into account. We could take this as the definition of the x_1 to use in the next step of the interval scheme.

We can prove that if $\xi \in X_0$, then $\xi \in X_1$. First, notice that for purposes of this proof, we can replace $F(x_0)$ by the scalar $f(x_0)$. This only shrinks the size of the interval we compute. Our second observation is that

$$f(\xi) = 0 = f(x_0) + (\xi - x_0)f'(\eta) \quad \text{where } \eta \text{ is between } \xi \text{ and } x_0$$

and is therefore in X_0. Then

$$\xi = x_0 - \frac{f(x_0)}{f'(\eta)} \subseteq x_0 - \frac{f(x_0)}{Rf'(X_0)} \subseteq x_0 - \frac{f(x_0)}{F'(X_0)} \subseteq X_1 \quad , \qquad \text{Q.E.D.}$$

We had better inquire as to what happens when $F'(X_0)$ includes zero. The Karlsruhe solution is to kick off the user, since their system lacks exterior intervals.

The alternative seems to be to replace X_1 by

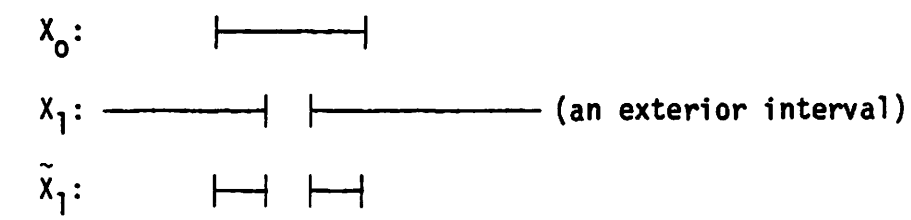
$$\tilde{X}_1 = X_1 \cap X_0 \quad .$$

Now every root in X_0 also lies in X_1, by repeated application of the previous argument. Therefore every root in X_0 lies in $X_1 \cap X_0$.

If the intersection is empty, we can be sure there was a mistake in the hypothesis $\xi \in X_0$. Otherwise we just continue our computation with $\tilde{X}_1$. If $\tilde{X}_1 = X_0$ then we have squeezed all the information out of this scheme that we can, and we should perhaps divide up the interval into smaller parts and work on them separately. Otherwise $X_1 \subset X_0$ so we have made some progress.

Unsolved Problem. If $\tilde{X}_1 \neq \emptyset$, and $\xi \notin X_0$, is there some other root in $\tilde{X}_1$?

In any case, we would expect the intervals X_i to get smaller until an interval representing the accumulated rounding error was reached. However, if there are two roots the final interval may contain them both. In this case we might have



Then the question for the programmer is to decide how to handle the pieces.

He should investigate both parts. If any part leads to a null set, he can conclude that no root was in that interval.

Nickel's scheme lacks exterior intervals and therefore does not converge in some cases. We examine such an example now.

Let $f(x) = x - \frac{1-x^2}{3+x^2}$ which also defines $F(X)$ by substitution.

He used the iteration

$$X_{n+1} = x_n - \frac{F(x_n)}{F'(X_n)}$$

$$x_{n+1} = \text{center of } X_{n+1} \quad .$$

He neglected to mention what expression he used for $F'(X)$. We can think of at least three:

$$f'(x) = 1 + g(x)$$

$$g_0(x) = \frac{1}{2} - \frac{1}{2}\frac{(x^2-1)^2 + 8(x-1)^2}{(x^2+3)^2}$$

$$g_1(x) = \frac{2}{\left(\frac{8+(x-1)^2}{1-(1-\frac{2}{x+1})^2} - 4\right)}$$

$$g_2(x) = \frac{8x}{(x^2+3)^2}$$

All three expressions $g_i(x)$ represent the same rational function of a scalar x, though they lead to different interval expressions $G_i(X)$.

Now, how much wider is $G_i(X)$ than $Rg(X)$? The answer depends on the interval X. If $X = [-1,1]$, then

$$G_0(X) = [-\tfrac{4}{3}, \tfrac{1}{2}]$$

$$G_1(X) = [-\tfrac{1}{2}, \tfrac{1}{2}] = Rg(X)$$

$$G_2(X) = [-\tfrac{8}{9}, \tfrac{8}{9}]$$

In general these expressions are optimal over different intervals. We find that

$$Rg(X) = G_0(X) \quad \text{if} \quad X \subseteq [0,1]$$

$$= G_1(X) \quad \text{if} \quad X \subseteq [0,1] \quad \text{or} \quad X \subseteq [\infty,-1]$$

$$\subset G_i(X) \quad \text{for almost all other } X \text{ except degenerate intervals and } [-1,1], \quad i = 0,1,2.$$

It is interesting that the expression g_1 was manufactured to work on $[-1,1]$ but it is no longer equal to $Rg(X)$ if $[-1,1]$ is perturbed by any amount, no matter how small.

Nickel actually used G_2 so his scheme was

$$X_{n+1} = x_{n+1} - \frac{F(x_n)}{1 + G_2(X_n)} \quad .$$

He claimed that his scheme converged quadratically with respect to the end-points of the intervals, for any starting point, until stopped by rounding error in $F(x_n)$. Kahan discovered that the scheme blew up on $X = [-1.2,1.2]$ because of a zero divide. If their interval arithmetic had been closed under division they would have been able to get quadratic convergence from any starting interval.

By mixing interval arithmetic and ordinary arithmetic the Karlsruhe group is able to get guaranteed error bounds on results. As we have seen, the techniques are not mathematically deep, the only heavily-used theorem being inclusion monotonicity. The one other important useful technique would be symbol manipulation.

It seems probable that technological users -- scientists and engineers -- would benefit more from a good implementation of interval arithmetic

within Fortran or Algol than from any other change in those programming languages that anyone would consider plausible. The few installations where interval arithmetic is available at all usually offer it only as subroutines, but it ought to be built right into the Fortran compiler as a standard data type. Perhaps the persons responsible for compilers are too busy producing new languages!

Probably the biggest problem in a convenient implementation is that a guard digit and a sticky bit are really vital to keep the intervals from growing unnecessarily.

13. WHAT CLAIMS SHOULD WE MAKE ABOUT THE PROGRAMS WE WRITE?[†]

We now imagine ourselves to be writing library programs for the use of others who may not be adept in numerical analysis. We would like to know what claims we might be able to make about the programs we write, and what claims we should make. After all, when we studied hardware we found that the difference between the claims $(a+b)(1+\gamma)$ and $a(1+\alpha)+b(1+\beta)$ had substantial implications.

The most straightforward situation is illustrated by the SQRT routine. We would like to claim that SQRT(X) = $\sqrt{x}$ if $x \geq 0$. (The case $x < 0$ is discussed in section [6] on execution time diagnostics.) Clearly this is hopeless because certain representable numbers have non-representable square roots. Perhaps we can state, instead,

$$\text{SQRT}(x) = \sqrt{x} \pm \frac{1}{2}\,\text{ulp} \quad \text{if} \quad x \geq 0 \quad .$$

Exercise: Show that an algorithm could be devised to deliver this accuracy.

In real computers there are always numbers whose square roots are extremely close to just halfway between representable numbers. We would have to compute many more digits than we wish to keep to decide between cases like

XXXX.49999997 which rounds to XXXX, and

XXXX.50000001 which rounds to XXXX+1 .

We shall see how these problems are handled in the Toronto 7094 routines written by Kahan. To keep the cost of computation reasonable, they guarantee

[†]See also W. Kahan, "The Error-Analyst's Quandary", Computer Science Technical Report #8, University of California, Berkeley, 1972.

$$SQRT(x) = \sqrt{x} \pm .50000163 \text{ ulps} \quad .$$

The 7094 had only 27 bits of precision so the subroutine clearly had to compute SQRT to about 100 ulps in double precision to make such a claim.

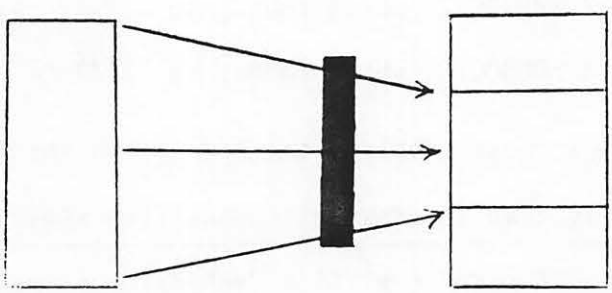
In the 7094 there are eight characteristic bits and 27 integer bits. Counting only positive normalized numbers there are 2^{34} different numbers. Of these many just differ by a power of four and are therefore essentially similar from the point of view of the square root routine. Of these 2^{34} numbers, for $29 \cdot 2^7$ the error exceeds $\frac{1}{2}$ ulp.

Having lost the attribute of "$\pm\frac{1}{2}$ ulp" we should see if certain other valuable properties of the square root function remain true. For this particular implementation, monotonicity is preserved. Also, the square root of the square of a number, whose square fits in single precision, is the original number.

Further, SQRT(X**2) = ABS(X), provided overflow or underflow didn't occur. A similar test would be

$$(SQRT(X))**2 = ABS(X)$$

but this is an example of an impossible demand to make on a square root routine. It is, after all, in the nature of a square root function to map the set of representable positive numbers onto a much smaller subset:



This means that at least two distinct x have identical computed square roots. Consequently the square of this computed square root could be one or the other but not both.

Concerning the claim SQRT(X^2) = ABS(X), one could proceed by direct calculation, checking all the relevant inputs, which numbered about 2^{27} in this case. Instead a mathematical proof was worked out. In general we would hope that comparable claims could be proven for other subroutines, or at least that comparably rigorous proofs could be given for lesser claims.

Other Functions

It gets more interesting when you consider what to do with functions other than the SQRT. You cannot always say that you will compute such

and such a function to within a unit in the last place. Half a unit in the last place is not achievable, because that is the table maker's dilemma. To be able to compute a function to within 1/2 ulp, it may first be necessary to compute it precisely and that may require infinitely many digits, if the value is exactly half way between two machine representable numbers. To make the correct decision you have to discover whether or not that is true. For the SQRT, any number whose square root was half way between 2 machine numbers would not be representable to single precision; so the problem doesn't arise in SQRT. We saw that we had trouble only when $4X$ is very near an odd square; but it couldn't be equal to that odd square because $4X$ is even.

It isn't clear why the dilemma cannot occur for, say, the exponential routine. It is possible, although we have every reason to doubt it. Could you construct an argument X, such that e^X was exactly half way between 2 machine numbers?*

Question: What is the significance of a number being transcendental?

Answer: A transcendental number or an irrational number cannot lie exactly half way between 2 machine representable numbers and the table maker's dilemma will not arise. But the dilemma can be arbitrarily closely approximated.

Question: Using the infinite series you can get to within a half ulp?

Answer: Yes, you can compute them as accurately as you like. And you know if you compute them accurately enough you can decide. But if you didn't know that the result couldn't be half way between 2 machine numbers, you might have to compute to infinite precision, because no error, however small, would enable you to render a decision.

---

*Actually, e^X is a bad example. It is known, I think, that for all rational X, e^X is transcendental (not rational), except for $e^0 = 1$. A similar result then follows for the logarithm. And similarly for the sine and cosine. But there could be other values where the issue is in doubt.

Reasonable Bounds for Other Functions

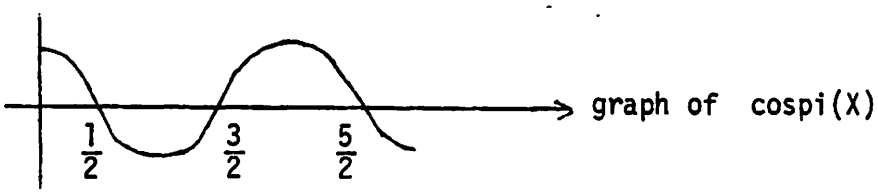
Let us consider some reasonable and plausible bounds for some other functions. These numbers are in ulps for routines on the 7094.

SQRT	< .50000163
LØG,QBRT	< .52 *
EXP	< .77
CABS	< .854
COSPI,DSQRT SINPI,DQBRT	< 1.0

COSPI and SINPI

COSPI(X) is what you ask for when you want to compute $\cos(\pi x)$.

$\cos(\pi x)$ vanishes when x is half an odd integer; the routine does vanish exactly at those points.



The claim that the error is at most 1 ulp is reasonable, even near a zero, because we know exactly what the function looks like near its zeros.

$$\cos \pi(\frac{2k+1}{2}+\xi) \approx \pm\xi \pm \xi^3/6 + \cdots$$

You find out what ξ is by subtracting half an odd integer (representable precisely for integers of decent size), and compute as accurately

---

*The .52 for QBRT is an acknowledgement that it is not a sufficiently important function to bother getting a better bound on the error. For this error, however, I was able to compute all the arguments for which the error was approximately that big. We'll see how that was done later.

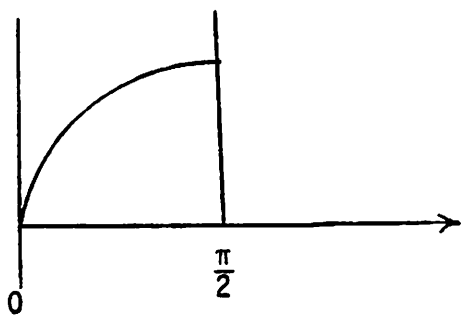
as you want using the power series.

The reason I'm pointing all this out is because for functions like cos and sin, even though we know where the zeroes are and how the functions behave near them, the roots are half integer or integer multiples of π, and we don't know π exactly. We know it to a large number of decimal digits, but we can't even represent it in the machine to as many digits as we know. Thus we are unable to say exactly where the functions vanish.

Computing Trig Functions When π Is Uncertain

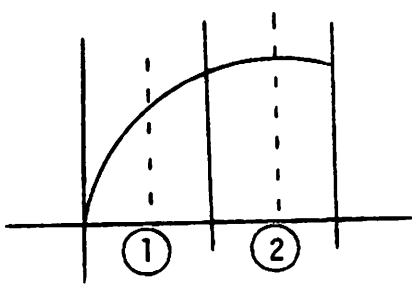
Let's see how this uncertainty in π contaminates our ability to compute the trig functions.

Suppose I wish to compute sin x. Since sine is periodic, the approximating function need not be repeated.



Need only consider this arc in computing sin x

What is conventional to do is to have 4 intervals:



In region 1, approximate sin x by an odd function; in region 2 approximate what amounts to cos x by an even function. Each arc is really only half as long by symmetry arguments; you build up the whole function by piecing together these arcs. In order to use these approximations, you have to reduce the given argument to 1 of those 4 intervals. That means dividing by some integer or half-integer multiple of π.

You have to compute and represent:

$$\frac{x}{\pi} \text{ or } \frac{x}{\pi/2} \text{ or } \frac{x}{\pi/4} = \text{integer} + \text{fraction}$$

The integer tells you which interval and which sign to use (that's called quadraproduction); the fraction says how far to move in that interval.

Then someone may say, why not use a representation that does not involve this argument reduction. There are infinite series after all. But look what happens in sin x for a moderately large argument.

$$\sin x = x - \frac{x^3}{6} + \frac{x^5}{120} - \cdots$$

Say x = 100. How many terms will you have to carry? What if x = 10,000? The series very quickly becomes useless, no matter how much work you were willing to perform. It is not because you have to compute a large number of terms; the problem becomes acute when you realize that most of the digits you compute are going to cancel off.

Computing SIN 100

What happens for x = 100? You know sin x cannot exceed 1, but the first term is 100. The leading two digits of 100 have to cancel off. $x^3/6 = 10^6/6$; that gives 5 digits to be cancelled off. $x^5/120 = 10^{10}/120$; 8 digits that must be cancelled.

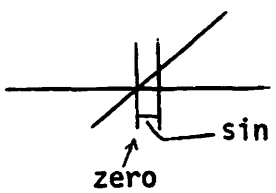
After this the terms get smaller; but you see you'll have to carry 8 decimal digits over and above the 14 you wanted in your answer. That is a more serious fact than having to compute many terms; they just use a do loop and take a few microseconds. But the extra digits require double precision and that is not done with a do loop. That takes a D.P. declaration and means the whole D.P. package sits in core.

Thus, while it is not impossible to do things this way, it is impractical.

Accuracy for Trig Functions

To make things more interesting, suppose I want to say my result is accurate to within a few units in the last place.

How close to a zero of $\sin x$ can we come? When you are close to a zero, $\sin x \doteqdot x$ (the slope of the graph is nearly ± 1).



How small can that distance be for numbers representable in the machine? If you represent numbers to 48 bits, you can approximate a root to within 96 bits, by a dodge.

You want to represent $m\pi$ by a number that for all practical purposes is an integer (it has to be rational in the machine).

$$m\pi \approx P/q \qquad q \text{ is a power of 2}$$

You are representing π by:

$$\pi \approx p/m \times 2^{-t} \qquad p, m \text{ are each 48 bits}$$

There is a theory that says if you allow yourself integers of a certain number of digits, you can approximate irrational numbers to at least twice as many digits as in either numerator or denominator. That's reasonable since you have twice as many digits to play with.

For certain, abnormal numbers, that is the best you can do. For most numbers, you do better.*

We'll just assume we can match the zero to 96 significant bits. Then you'll need another 48 bits. It looks like you'll have to carry 150 bits after the binary point, to get 47 or 48 that are correct. Not to mention the digits before the binary point that are going to cancel. Now you see the utter impracticality of it all.

Question: It appears there are several reasons for wanting to reduce x. One is the fact that you'll get overflows. That seems even more important than questions of precision.

Answer: Of course, if x is enormous, x^3 would overflow before you got anywhere. But that situation could be coped with, by whatever means you used to cope with multiple precision. If you have to assign extra words to the right, you wouldn't mind assigning a word for the exponent.

More to the point is if x is small; then x^3 may underflow and you may get all kinds of messages that have no significance at all. In cases like this, x is already a very good approximation to $\sin x$. If $x = 10^{-100}$, $\sin x = x$ is correct to something like 100 decimal digits.

Question: Can't you tell people something who want to compute things like $\sin 10^{-100}$? Like maybe to rephrase it?

Answer: I really haven't explained how I'm going to do $\sin x$. I was only showing why the obvious ways won't work. You need 150 digits to the

*See Hardy and Wright's book on the theory of numbers.

right of the point if x is close to a zero, and some interesting number to the left if x is large. The conspiracy is getting worse; that's why we don't do it that way.

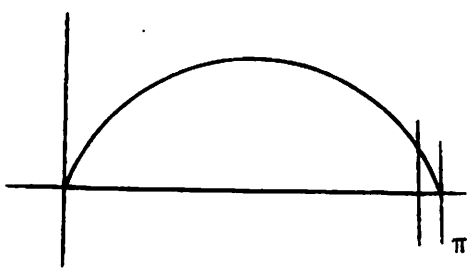
Quadraproduction (Argument Reduction)

So we use quadraproduction. That gets rid of needing digits to the left of the point. But that has not gotten rid of the problem of carrying digits to the right. In some respects we have made that problem worse.

Remember, we don't know the value for π. And no matter how many digits we put in for π, we can't do the division, $\frac{x}{(\pi/2)}$. You compute: $\frac{x}{(\pi/2)}(1+\xi)$. ξ is at most 1 ulp of the precision you are using for π and the division. You are going to commit at least 1 rounding error in the division. And you have already made an error in π.

You have effected quadraproduction not on x but on $x(1+\xi)$. Already, you are computing the sin of the wrong angle. You can imagine what that will do if x is close to a zero. You just moved the argument. You are computing for the wrong angle, so you can't possibly get sin x correct to a unit in the last place for any sort of moderately large argument.

Say you do division to double precision and $x \doteqdot \pi$.



You have moved x by a unit in the last place of double precision; the closest x can come to a zero is about 1 ulp of single precision;

you still have roughly a single precision word to play with.

The situation isn't too bad at π, 2π, or 3π. But how about $10^5\pi$? You'll have lost 5 decimal digits.

Error in SIN(X) and COS(X)

How you actually compute the sin is not pertinent to this class. What is important is that you have to think about what you can compute in a rather different way than you might be accustomed to. Namely, that whenever you ask the machine to compute SIN(X), you can be fairly confident that that is not what it is going to do.

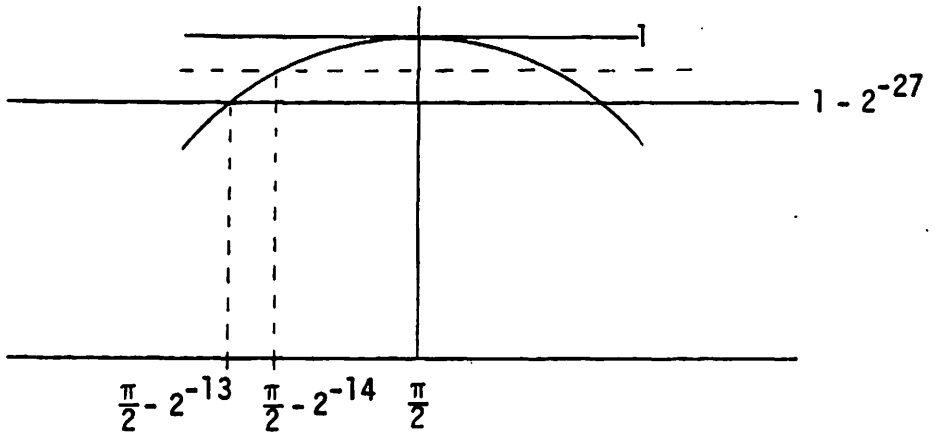
Suppose someone did demand $SIN(X) = \sin(x) \pm \frac{1}{2}$ ulp. Unless x is restricted to an unreasonably small domain this isn't possible. The first step in the SIN routine is to find the fraction $\frac{x}{2\pi} - \lfloor\frac{x}{2\pi}\rfloor$. Unfortunately the value of π is not available in the computer to an infinite number of figures so $\frac{x}{2\pi}$ is computed erroneously. For large x the fraction is only a few bits so that any errors in $\frac{x}{2\pi}$ are revealed highly magnified by cancellation in $\frac{x}{2\pi} - \lfloor\frac{x}{2\pi}\rfloor$.

Clearly we can't compute any more accurately than $\sin(x(1+\xi))$ for some small ξ. Then for $x \doteq \frac{\pi}{\xi}$ the uncertainty in the argument is comparable to π so the computed result has no significant figures. Fortunately the first few integer multiples of π differ from representable numbers only by a modest fraction of an ulp in single precision. By using double precision for π and the division $\frac{x}{2\pi}$ it is possible to get fairly good results for $x \doteq 100$, which is not possible in single precision.

The claim that might be made would be

$$SIN(X) = \sin(x(1+\xi)) \quad .$$

Even this constraint is difficult to satisfy for small ξ, especially in the region of a maximum:



$1 - 2^{-27}$ is the next smallest 7094 number below one. Suppose we wish to report a sine just less than halfway between $1 - 2^{-27}$ and 1. Then the natural output is to round down to $1 - 2^{-27}$, which is the sine of about $\frac{\pi}{2} - 2^{-13}$. The number we wanted to compute was the sine of about $\frac{\pi}{2} - 2^{-14}$. Therefore we compute $\sin(x(1+\xi))$ for $\xi \doteq 2^{-14} \doteq 10^{-4}$, because we have rounded the answer to fit in the machine. This value of ξ seems to be unnecessarily large.

Consequently we limit our claim to

$$\text{SIN}(X) = (1+\varepsilon)\sin(x(1+\xi)) \ , \quad |\xi| < 10^{-15} \ (\approx 1 \text{ ulp } \underline{\text{double}} \text{ precision}),$$
$$|\varepsilon| < .52 \text{ ulp} \quad .$$

It is an inherent feature of the sine function that we must state our accuracy in this complicated form rather than in the simpler forms we considered earlier. Otherwise we would have to make a terribly pessimistic statement about the subroutine.

Since we have such a peculiar form for our uncertainty, we should investigate what useful properties our computed values have. For the Toronto routines it was possible to prove that the difference in the <u>computed</u> SINE for two consecutive representable arguments either had the correct sign was zero. It was also true that

$$\text{ABS(SIN(X))} \leq 1.0$$

$$\text{ABS(COS(X))} \leq 1.0$$

$$\frac{\text{SIN X}}{\text{X}} \leq 1.0$$

$$|\,1 - (\text{DBLE(SIN(X))}**2 + \text{DBLE(COS(X))}**2)\,| \leq 2 \cdot 10^{-8} \quad .$$

One reason many trigonometric identities were very nearly preserved was that the argument reduction $\frac{x}{2\pi} - \lfloor\frac{x}{2\pi}\rfloor$ was done in a uniform manner for each trigonometric function. That is, ξ depends on x but not on the function being computed. ε, on the contrary, depends only on the function <u>value</u> and not at all on x or on the function.

To make things more convenient for scientists and engineers, the following functions are also available:

$$\text{SINPI(X)} = (1+\varepsilon)\sin(\pi x)$$

$$\text{COSPI(X)} = (1+\varepsilon)\cos(\pi x) \quad .$$

Then $\xi = 0$ because argument reduction involves only integer subtraction.

You could reasonably expect COS(X) to satisfy:

$$\text{COS(X)} = (1+\gamma)\cos(x(1+\xi))$$

where the ξ is the same as in SIN(X), $|\gamma| \sim 1$ ulp of single precision. You get the same error ξ because you do exactly the same division and then interpret the integer part differently.

SIN(X) and COS(X) computed for very large operands X may be wrong, but nevertheless they are the sin and cos of some reasonable argument. Consequently:

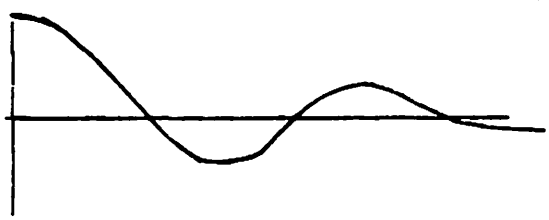
$$\frac{\sin}{\cos} \approx \tan \quad \text{to within a few ulps} \quad .$$

Question: If you pass the sin routine a very large argument, the argument itself may be represented rather poorly.

Answer: That's right. If X is really large and it is at all uncertain (consider, where did X come from, another computation maybe?), the uncertainty in X will cover a large interval, and the sin and cos could oscillate several times in that interval. That does happen. And when it does, I think the most we could hope for is some kind of internal consistency.

Question: Shouldn't there be a diagnostic?

Answer: There is sometimes. But it is hard to say whether this is an error or not. In some asymptotic formulas, although the sines and cosines oscillate in an uncertain way, they are multiplied by things that are very small. Such as in the Bessel functions:



it approaches $\frac{1}{\sqrt{x}} \sin x$. People sometimes do have formulas in which they want trig functions of large arguments. But the uncertainty gets less important as the argument gets larger, as these later terms are a small contribution to some series.

Question: Wouldn't you suggest to people that they write their own sine routine, so that the argument is in some interval in which you can actually compute the sine, using the system subroutine? If they are just going to get garbage, they should get a result that is essentially zero.

Answer: But it really isn't garbage, you see. The function is only important where it is big (in the above picture, say), and there you get reasonable accuracy, sometimes. Remember, you only get troubles like this

on our machine for $x \sim 2^{40}$ or so. That's gargantuan. For numbers like 10,000, or 100,000, the sin and cos will have deteriorated a bit, but this is generally not serious for the applications involved. It is rather difficult for somebody to go through the analysis that tells him he should do something different rather than accept the values as computed.

For people who use abnormally large arguments and may not realize what they are doing, you're right -- they should be given a diagnostic. But that involves a decision -- where to draw the line. Should you tell him when he's lost all his digits, or only half of them, or what? On IBM equipment, it is customary to issue a diagnostic when changing the argument by 1 ulp can run you through an interval comparable to π. On the 360, this means $x \sim 10^6$. On the 7094, $x \sim 10^8$. My programs don't give a diagnostic. They just say here is what you get and if you are worried about it use the SINPI and COSPI routines, for which no diagnostic is needed. Of course, for roughly half the machine number arguments, SINPI and COSPI give you +1, -1, or 0. The numbers are mostly integers times big powers of two; thus SINPI and COSPI usually return 0 or +1. But that's alright. We now have a reasonable way of interpreting what you get and why you get it.

What You Can Expect From Error Analyses Generally

I guess I'm introducing you to the rather interesting notion that instead of being able to say you have gotten something that is wrong by a unit in its last place, it may be that you'll be obliged to say that the answer you have differs by a unit in its last place from the exact answer of a problem that differs by some small amount from the problem you originally posed. And that is normally what is considered to be a successful error analysis. But even a statement like this usually cannot be made. For

non-trivial problems such as solving a set of linear equations, even using a decent numerical method, the best published analyses state that the answer is the precise answer of a problem perturbed slightly <u>in norm</u> from yours. This perturbation may be many ulps of some elements of the matrix, so this statement is not nearly as strong as the statement we would like to make, that the answer is a few ulps from the precise answer for a problem a few ulps from the given problem. That is, if we solve a system of linear equations $Ax = b$ in the usual way, we can show that the computed x satisfy

$$(A+\Delta A)(x+\Delta x) = (b+\Delta b)$$

where $\|\Delta A\| \ll \|A\|$, $\|\Delta x\| \ll \|x\|$, $\|\Delta b\| \ll \|b\|$. But if we generalize slightly to computing the inverse of A, we find for the result X that we get:

(1) $X = A^{-1} \pm$ a few ulps is not true.

(2) $AX \doteq I$ is not true. AX could be much closer to zero than 1.

(3) $(X+\Delta X) = (A+\Delta A)^{-1}$, with each element of ΔA a few ulps of the corresponding element of A, and likewise for ΔX and X, is not true.

(4) $(X+\Delta X) = (A+\Delta A)^{-1}$ for $\|\Delta A\| <$ a few ulps of $\|A\|$
$\|\Delta X\| <$ a few ulps of $\|X\|$
<u>might</u> be true. This corresponds to the assertion we made for the trigonometric functions.

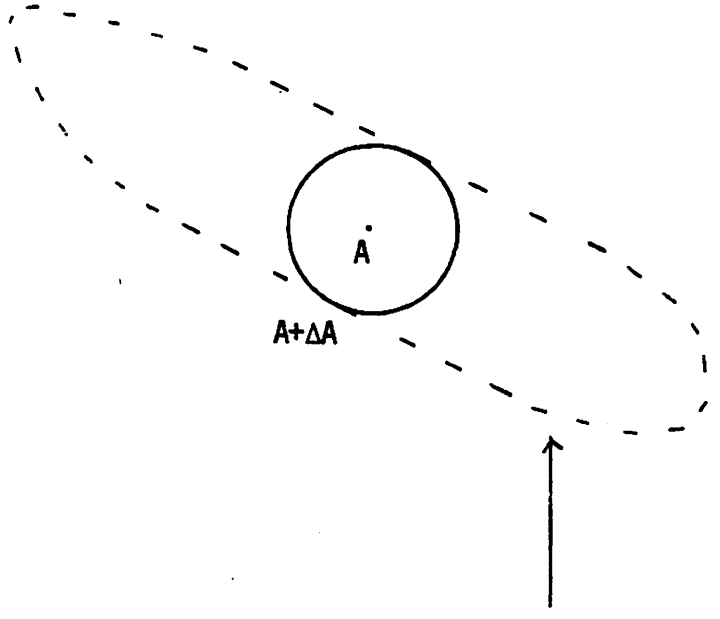
<u>Thesis project</u>: Prove that some standard algorithm does or does not always produce a result that satisfies condition (4).

The best result known is Wilkinson's:

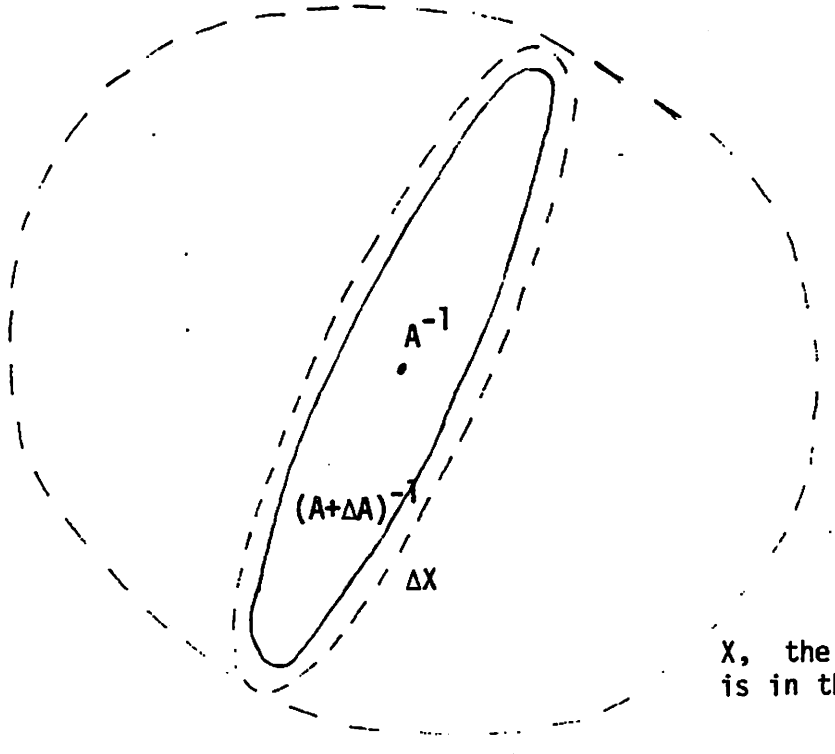
$$\|X-A^{-1}\| \leq \max\|(A+\Delta A)^{-1} - A^{-1}\|$$

where the maximum is taken over ΔA such that $\|\Delta A\| <$ (some number) of ulps of $\|A\|$. "Some number" grows as $n^{1/4(\log n)+\text{constant}}$. Numerical analysts believe this to be entirely too pessimistic -- the evidence indicates that "some number" should be n^{constant}.

The entire situation is best illustrated by a picture. Suppose we have the point A in n-dimensional matrix space, and we allow an uncertainty ΔA about A which includes the matrices we consider indistinguishable from A for practical purposes. Then there is somewhere else the point A^{-1} and the set of matrices whose inverses are those of points in the ball $A+\Delta A$. We would like to state that X is a member of the latter set slightly enlarged by ΔX:



A's such that X^{-1} is in this set



X, the computed A^{-1}, is in this set.

In reality it has only been shown that the X's lie in a very large ball centered on A^{-1}, whose diameter is slightly larger than the largest diameter of the set of $(A+\Delta A)^{-1}$. Then the A's corresponding to this large sphere form a somewhat pointed set centered on A.

Experience seems to indicate that if the condition number $\|A\|\|A^{-1}\|$ is not too large the set $(A+\Delta A)^{-1}$ does not deviate too far from spherical symmetry.

No example has been given of a matrix A whose computed inverse X was very far from the inverse of every matrix near A. No one has any idea how even to construct such a matrix. Nonetheless, no general proof that the assertion (4) above is true seems forthcoming soon.

How Approximate Are Your Results

The sin and cos have thus introduced you to the notions (pervasive in numerical analysis) that you cannot compute approximately the right answer to your problem; you can only hope to compute approximately the right answer to very nearly your problem. If you can do that, people will say you have used a stable numerical method.

There are exceptions which correspond to rather peculiar measures of what we mean by approximately. For example, if you examine the quadratic equation, $Ax^2 - 2Bx + C = 0$, it is clear that A, B and C are pieces of data. But what about the 2 on x^2? Is that datum or part of the structure of your mapping? If you think of 2 as a datum susceptible to variation, so that you might have written $x^{2.00000003}$, then there could be an infinite number of solutions, whereas the equation in x^2 has only two.

The way that the solutions vary with changes in A, B, C and 2 is different from the way the solutions vary with changes in A, B and C only.

So when you talk about a problem very near yours, you may be fixing things that might otherwise have been thought of as data, subject to variation.

What Should Be Data

In some cases, the issue, as to what should be data and what shouldn't, is not altogether clear. What should be allowed to vary?

As an example, I'll talk to you as a CDC engineer or programmer would. He would say that nobody can ever know exactly what the operands should be (I dispute that, by the way). "If you want to compute LOG(X), you should be willing to accept

$$\text{LOG}(X) \approx (1+\epsilon)\log(X(1+\xi)) \quad .$$

(That is, he is willing to perturb the argument). You don't know what X is anyway, so why should you care if I change it a little bit?"

Remember the graduate student working on wing design? [5] He cared.

I would care a great deal if I were computing A^B. You write it as:

$$A^B \approx \text{EXP}(B*\text{ALØG}(A)) \quad .$$

If B is a large number you find you didn't compute A^B but rather something else. If B is large, A had better be close to 1, or you'll overflow. But if A is very close to 1, and you have to take

$$\log(A(1+\xi))$$

where $1+\xi$ is also close to 1, the log can be changed drastically, say by a factor of two. Then when you do the rest of the computation, you're dead.

The engineer would say that's perfectly reasonable because you don't

know A and you don't know B. But you might want to dispute that.

1+ξ Should Not Be In Your Argument

It is my judgement that the (1+ξ) in the log function does not belong there because there are perfectly economical ways to compute the log without it being there. You just have to be a little bit careful. It amounts on our machine to changing statements like X-1.0 to (X-0.5)-0.5. In the first case, if X is close to 1 the answer may be zero, whereas in the second case, the difference is taken correctly.

By using a better approximation and a bit more care, you should be able to get a log function in which the (1+ξ) perturbation term does not appear. You'll have to agree that it's preferable to think of the logarithm without that term.

My argument for wanting to get rid of those terms when you can is that they make the structure very different from what you're used to and from what you expect.

That is my argument for putting the perturbation in SIN(X(1+ξ)) down to double precision. For arguments in the range of π or smaller, you could eliminate ξ by slightly increasing e. We saw that perturbing x by an ulp of double precision might change the single precision answer by a few units in its last place. So you say e is 5 units, instead of 1 and you don't mention ξ at all.

Hirondo Kuki: "Getting rid of them (factors like the 1+ξ) gives you the strictest accuracy requirement for a subroutine that you could conceive of. Therefore it gives the simplest goal for the programmer to aim at, insofar as accuracy is concerned. And in some computations, for example with integer arguments or assuming all prior computations went meticulously

well, where there is no error in the argument, the benefit is real. And, it is simpler to explain to users. However, it may cost diamond where mere glass may have served."

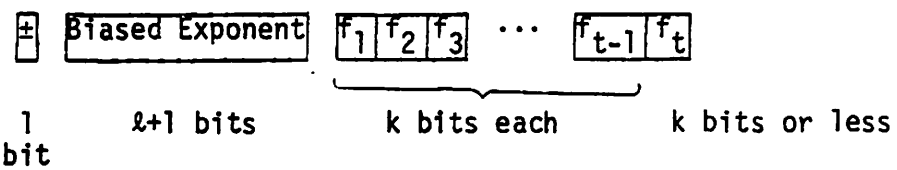
Being simple to explain to users seems to me to be the most important reason of all. Coding the routine is only half the problem. The other half consists of informing the users what exactly the subroutine accomplishes.

14. WHAT IS THE BEST BASE FOR FLOATING POINT ARITHMETIC?

It perhaps seems clear that, if there were an outstandingly best base for arithmetic, humans would have adopted it long ago. The base ten seems just now to be winning universal acceptance, though this may be an accident of history. However, if we take the point of view, as we have often done in this course, that ordinary users should have to learn as little as possible about the workings of computers, then base ten would be preferable as being closest to their normal experience. From other points of view it is inefficient of storage, as we shall see below.

Our machines are basically composed of two state devices so that the most efficient base is a power of two. In this light we see that since base ten requires four bits, it is really like base sixteen except six of the bit combinations are ruled illegal and are not used, which seems wasteful.

Let us restrict our attention then to bases of the form 2^k. Then all numbers will have a representation of the form $(2^k)^i \cdot f$ for some "normalized" f in the range $2^{-k} \le f \le 1 - 2^{-kt}$, represented by t digits, and for some i in the range $-2^{\ell} \le i \le 2^{\ell} - 1$. Then our word length equals the sum of 1 bit for sign, $\ell+1$ bits for biased exponent, and kt bits for the integer part:



t need not be an integer but kt is.

The analysis to follow shortly will assume this kind of representation. Let us pause to consider some other forms:

1. Notice that on a binary machine our normalized numbers always have a one bit in the most significant position. Since this is constant we could drop it and save a bit. But then there is no way of representing unnormalized numbers. We could not use unnormalized numbers as a partial remedy for underflow [6]. It seems that binary to decimal conversion could also be hampered. If we could locate all the present uses of unnormalized numbers and implement them in the hardware, such a scheme might be desirable. (It is used on PDP 11-45 computers.)

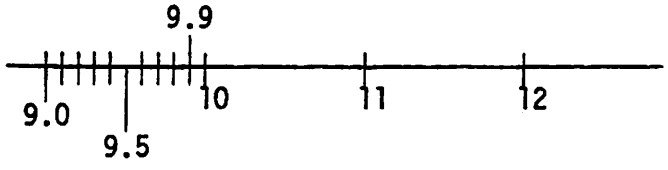
2. Another possibility is suggested by the fact that numbers of magnitude far from 1 occur much less frequently than numbers of magnitude near 1. Perhaps by some suitable encoding we could find a way to represent numbers near one with only a few bits for the exponent, and numbers far from one with more bits in the exponent and less precision, which would be justified because they occur infrequently. Unless someone can prove otherwise we imagine such a scheme would make error analysis rather difficult. For instance, multiplying a system of linear equations through by a scale factor of a power of two would likely change the computed result. We would then be faced with the problem of choosing a scale factor to optimize the precision throughout the calculation, so that the uncertainty in the result is a minimum.

3. We could also try the scheme in [2] where numbers are represented by their logarithms in fixed point form. One of the common objections to this scheme is that addition takes a long time. D. Muller, in an unpublished manuscript, showed that addition in such a scheme could be done almost as quickly as multiplication in conventional systems. With advances in multiplication hardware this may no longer be true. However, the fact that precise representations of 2 and 3 are mutually exclusive is perhaps

the strongest argument against a logarithmic scheme.

Having considered some other possibilities, we will proceed on the assumption of a conventional format, and inquire as to which are the valuable attributes in a number representation.

Clearly precision is an important attribute. By precision we might mean the worst relative spacing of adjacent machine numbers. For instance, on a two digit decimal machine numbers are spaced like:



We see that the relative spacing changes by a factor of the base (from $\frac{1}{100}$ to $\frac{1}{10}$) near a power of the base. This argues for a small base, so that binary is best. Generally, the smallest relative difference is

$$\begin{matrix} 1.00000\cdots \\ .\underbrace{11111}_{kt\ 1's} \end{matrix} \qquad \frac{1-(1-2^{-kt})}{1 \text{ or } 1-2^{-kt}} \doteq 2^{-kt} \quad ,$$

and the largest is

$$\begin{matrix} \overbrace{1.0000\cdots1}^{kt\ bits} \\ 1.0000\cdots0 \end{matrix} \qquad \frac{(1+2^{k-kt})-1}{1 \text{ or } 1+2^{-kt}} \doteq 2^{k-kt}$$

On the other hand, we can say with equal validity that the spacing is always one unit in the last place (ulp)! Of course "the last place" jumps at a power of the base. The difference in these points of view is the difference between the producer and the consumer. The producer of numerical routines is interested in routines which are the best among all imaginable

on some machine. The best possible routine will have an uncertainty of $\frac{1}{2}$ ulp due to the necessity of rounding to the nearest machine representable number, so ulps are the natural unit of measure of precision, and variation in the relative spacing is an implicit constituent of the word "ulp".

The consumer or user of the routine is more interested in its relative accuracy, which might be stated as one part in 10^{13}. After all, the precision of his inputs is most likely to be stated in this way. Then the relative spacing becomes important, since the base affects the maximum precision an algorithm can achieve. We will consider this definition of precision for the present discussion.

Besides precision, we would like to know the range of representable numbers. We shall express this as the ratio of the largest representable positive number to the smallest representable positive number, which is

$$\frac{(2^k)^{2^\ell-1}\cdot(1-2^{-kt})}{(2^k)^{-2^\ell}\cdot 2^{-k}} \doteq (2^k)^{2^{\ell+1}} \quad .$$

Clearly, the more bits we allow for the exponent, the greater the range, but the less the precision, for fixed word length. To be specific, the word length w satisfies

$$w = 1 + (\ell+1) + kt \quad .$$

Now if we define the range and precision parameters as follows:

$$r = \log_2((2^k)^{2^{\ell+1}}) = k\cdot 2^{\ell+1} \quad ,$$

$$p = \log_2(2^{k-kt}) = k - kt \quad ,$$

we see that

$$w = 1 + \log_2(\frac{r}{k}) + p + k = 1 + \log_2 r + p + (k - \log_2 k) \quad .$$

If we can specify p and r a priori then w is a function of the base 2^k. We want to choose k to minimize the amount of storage needed, w. Then we should examine the values of $k - \log_2 k$, to determine the minimum with respect to k.

base	k	$k - \log_2 k$
2	1	1
4	2	1
8	3	1.415
16	4	2

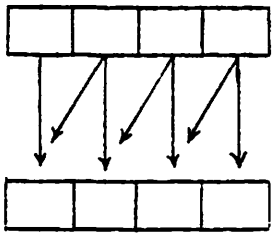
We conclude that the best value of the base is 2 or 4, based on the worst case of a jump in precision. Perhaps it would be more realistic to base our decision on some sort of average relative precision. For a number of plausible distributions of numbers the best base has been asserted to be 4. (Brent, (1972) "On the Precision Attainable with Various Floating Point Number Systems").

We prefer binary over base four because errors propagate in a more predictable fashion, as we shall see later. Surprisingly, many arguments are still put forth in favor of octal and hexadecimal bases. In the case of the 360, base 16 was chosen to reduce the number of shifts required to align the operands prior to the operation and to normalize the result. Empirical tests on the 7094 had demonstrated that most such shifts were one or two places, which could be avoided by using base 16.

This conclusion is valid if the time to shift is proportional to the numbers of positions shifted and if the decision as to how many places to shift requires no time. This is not generally true. The number of shifts

required is determined by counting leading zeros and this definitely takes time unless the base is two, when it suffices to shift until a one appears as the left bit.

Shifting is often accomplished by choosing which of two sets of gates between registers are enabled.



One set is a straight transfer, the other shifts in the transfer process. In the CDC 6000 machines a tree structure of shifting registers is used which, at each node, may either shift one bit or not shift, so that any number of shifts can be eventually accomodated. In general, however, the one or two bit shifts that IBM was worried about can be accomodated in the time it takes to transfer the word between registers.

Perhaps some future machine organization will be able to take advantage of such an idea using a base of 8 or 16, resulting in some simplification of the hardware, which might save 5% of 1% of the cost of a complete computer, which seems negligible. Any such scheme will, however, waste an average of perhaps one or two bits in leading zeros, which is a 1% - 3% loss in available storage for normal length words of 60-100 bits. A few percent of storage costs is a much larger price to pay for a hexadecimal base compared to the simplification in hardware.

However, the foregoing arguments do not yet supply a reason to prefer binary (k = 1) over base 4 (k = 2). The following arguments are intended

to illustrate the main reason for preferring binary, namely that the density of representable numbers is most uniform when the base is smallest.

Error Propagation on Non-binary Machines

We consider division. Let

x = value of X rounded to t digits of base $b \geq 2$, $t > 3$.

y = value of Y rounded to t digits of base $b \geq 2$, $t > 3$.

q = value of x/y rounded to t digits of base $b \geq 2$, $t > 3$.

How different is q from X/Y?

For definiteness, say $b^{t-1} < X < b^t$, so x is an integer in $b^{t-1} \leq x \leq b^t$ and $|x - X| \leq \frac{1}{2}$. Similarly $b^{t-1} < Y < b^t$, $b^{t-1} \leq y \leq b^t$, $|y - Y| \leq \frac{1}{2}$. Now $b^{-1} \leq x/y \leq b$, so $b^{-1} \leq q \leq b$ and there are two cases to consider regarding the rounding of q:

$\gamma = 0$ $\quad b^{-1} \leq x/y \leq 1$ so $b^{-1} \leq q \leq 1$ and $|q - x/y| \leq \frac{1}{2}b^{-t}$,

which is $\frac{1}{2}$ ulp[†] of q unless $q = 1$.

$\gamma = 1$ $\quad 1 \leq x/y \leq b$ so $1 \leq q \leq b$ and $|q - x/y| \leq \frac{1}{2}b^{1-t}$,

which is $\frac{1}{2}$ ulp of q unless $q = b$.

In either case, $|q - x/y| \leq \frac{1}{2}b^{\gamma-t}$. But we want to bound $|q - X/Y|$, so we must next examine $|\frac{x}{y} - \frac{X}{Y}|$. We find $|\frac{x}{y} - \frac{X}{Y}| = |\frac{x-X}{Y} + (\frac{x}{y})\frac{Y-y}{Y}| \leq \frac{1/2}{Y}(1 + \frac{x}{y})$. Again there are two cases:

$\gamma = 0$ $\quad b^{-1} \leq x/y \leq 1$ so $|\frac{x}{y} - \frac{X}{Y}| \leq \frac{1/2}{Y}(1 + 1) = \frac{1}{Y} < b^{1-t}$, and then

$$|q - X/Y| < b^{1-t} + \tfrac{1}{2}b^{-t} = (\tfrac{1}{2} + b)b^{-t} = \underline{\underline{(\tfrac{1}{2} + b) \text{ ulp of } q.}}$$

[†]"ulp" = "unit(s) in the last place..."

$\gamma = 1$ $\quad 1 \le x/y \le b$ so $|\frac{x}{y} - \frac{X}{Y}| \le \frac{1/2}{Y}(1+b) < \frac{1}{2}(1+b)b^{1-t}$, and then

$$|q - X/Y| < \tfrac{1}{2}(1+b)b^{1-t} + \tfrac{1}{2}b^{1-t} = \tfrac{1}{2}(2+b)b^{1-t} = \underline{\underline{(1+\tfrac{1}{2}b) \text{ ulp of } q.}}$$

Note $\frac{1}{2}+b > 1+\frac{1}{2}b$ since $b > 1$.

Can these bounds be approached closely? Yes...Here is how...

First, the bounds upon $|\frac{x}{y} - \frac{X}{Y}|$ can be approached when $x - X$ and $Y - y$ have the same signs and magnitudes near $\frac{1}{2}$, and y is close to b^{t-1}, and x is slightly less than y, in case $\gamma = 0$, or b^t in case $\gamma = 1$. The bounds upon $|q - x/y|$ can be approached when $b^{t-\gamma}x/y$ is nearly an integer plus $\frac{1}{2}$. To accomplish this last condition we assume first that $y = b^{t-1} + m$ for some "small" positive integer $m << b^{t-1}$. Then we assume

$x = b^{t-1} + n$ for some "small" non-negative $n < m$ in case $\gamma = 0$

$x = b^t - n$ for some "small" non-negative $n << b^{t-1}$ in case $\gamma = 1$.

<u>In case $\gamma = 0$</u> we have

$$x/y = (1 + nb^{1-t})/(1 + mb^{1-t})$$
$$= 1 - (m-n)b^{1-t} + (m-n)mb^{2-2t} - (m-n)m^2b^{3-3t} + \cdots < 1 .$$

To have $|q - x/y| \doteq \frac{1}{2}b^{-t}$ it suffices that $(m-n)mb^{2-2t} \doteq \frac{1}{2}b^{-t}$; i.e. $2(m-n)m \doteq b^{t-2}$ with small relative error.

One choice worth considering is $m = b^{[\frac{t+1}{2}]-1}$, $n = m - \frac{1}{2}b^{[\frac{t}{2}]-1}$, and there are many other appropriate choices, as examples will show.

<u>In case $\gamma = 1$</u> we have

$$x/y = (b - nb^{1-t})/(1 + mb^{1-t}) = b - (bm+n)b^{1-t} + (bm+n)mb^{2-2t} - \cdots$$

To have $|q - x/y| \doteq \frac{1}{2}b^{1-t}$ it suffices that $(bm+n)mb^{2-2t} \doteq \frac{1}{2}b^{1-t}$; i.e.

$2m(bm+n) \doteq b^{t-1}$ with small relative error.

Among the many possibilities is the choice $m = \frac{1}{2}b^{[\frac{t}{2}]-1}$, $n = b^{[\frac{t+1}{2}]} - bm$.

Example. We use $t = 4$ digits of base $b = 10$; case $\gamma = 0$. Suppose $X = 1013.5001$, $Y = 1017.4999$, $A = 1000.4999$, $B = 1006.5001$. If we compute $(\frac{x}{y} - \frac{A}{B})$ using correctly rounded 4-digit decimal arithmetic, how much in error can the computed result be? A naive analysis would suggest an error of about .0003 thus:

Round X to $x \equiv 1014$
Y to $y \equiv 1017$
A to $a \equiv 1000$
B to $b \equiv 1007$
} committing in each case an error smaller than $\frac{1}{2}$ ulp.

Now $x/y = .99...$ will be in error by about $(\frac{1}{2}+\frac{1}{2} = 1)$ ulp, and its rounded value $q = .99...$ by about $\frac{1}{2}$ ulp more than that, i.e. by .00015. Similarly, the rounded value $r = .99...$ of a/b will be in error by about $(\frac{1}{2}+\frac{1}{2}+\frac{1}{2} = \frac{3}{2})$ ulp. Their difference will be computed exactly, so we expect naively that $q-r$ will differ from $\frac{X}{Y} - \frac{A}{B}$ by at most about .0003. In fact

$x/y = .99705015$ so $q = .9971$, but $X/Y = .99606899$;
$a/b = .99304866$ so $r = .9930$, but $A/B = .99403855$;
$q - r = .0041$, but $\frac{X}{Y} - \frac{A}{B} = .00203044$.

The error here is not just 3 ulp of .99..., but almost 21 ulp! [cf. twice $(\frac{1}{2}+b)$ ulp in case $\gamma = 0$.]

On a hexadecimal machine ($b = 16$) we could, in a similar calculation, get almost 33 ulp instead of the naively anticipated 3 ulp. Hex is Horrible.

On a binary machine ($b = 2$) we could get at worst 5 ulp instead of

the naively expected 3 ulp. Binary is Best.

The foregoing examples are not entirely persuasive, perhaps because they compare a rigorous and achievable bound with a naively mistaken bound. But, on reflection, the comparison will not appear unreasonable. Accuracy is not like Virtue (which is its own reward) nor like Beauty (which is in the eye of the beholder); rather Accuracy is like Justice (which must be both done and seen to be done). Accuracy which cannot realistically and economically be appraised is of disputable value. ["I am confident, though I cannot be sure, that the number of colors needed to color any map in the plane is 4.00000..."]

Of course, we could "easily" have overestimated the error in the quotient q as follows: x (= rounded X) is uncertain by $\frac{1}{2}$ ulp, as is y (= rounded Y), so their quotient x/y is uncertain by $(\frac{1}{2}+\frac{1}{2})\times b$ ulp, where the growth factor b allows for the uncertain relative value of the absolute uncertainty $\frac{1}{2}$ ulp. Then rounding x/y to q introduces another $\frac{1}{2}$ ulp uncertainty; the total is $(b+\frac{1}{2})$ ulp, as predicted for case $\gamma = 0$ above. But the same argument could be used for a product $p \doteq xy$ to show that p's uncertainty is $(b+\frac{1}{2})$ ulp. This prediction is a bit large; it is left as an exercise for the reader to show that $|p - XY| < (1+\frac{1}{2}b)$ ulp of p, with near equality possible. (And $(b+\frac{1}{2})/(1+\frac{1}{2}b) = 2 - \frac{3}{2+b}$ is an increasing function of b.) Moreover, if xy does not have to be normalized before rounding, $|p - XY| < \frac{3}{2}$ ulp!

The point of the foregoing arguments is to show that the propagation of error and uncertainty is more difficult to estimate realistically and economically when $b > 2$. The difficulty arises when relative error has to be "converted" to absolute error or vice-versa.

Binary Is Best, But For Whom? And Does It Matter?

Today's technology suggests that the cost of a central processor is a relatively small fraction of the total for the system delivered without its I/O peripherals; that's the processor and its storage that doesn't require human intervention (includes fast storage, extended core storage and possibly disk or drum). Lumping all that together, it is clear that the cost of the arithmetic unit is negligible. So you might as well make it right. It is the cost of storage that is high, so you should economize there. That's where the argument that larger bases mean better utilization of storage becomes important.

Another aspect of today's technology is that you can gain speed by adding a little hardware at a small cost. This may not have been true when the 360/30 was designed, so you'd want to limit yourself to small registers and data paths. Then hexadecimal offers the advantage that normalizations do not have to be done as often (since 3 leading binary zeros are allowed). The arithmetic units could be made to look faster, on the average. But on large machines designed to do lots of floating point calculations, you must have large registers. You cannot have fast efficient floating point arithmetic built up from tiny registers (too much microcode needs to be executed). For large registers, shifts are not such a big chore; they don't take very long so you don't care how many shifts are needed. On CDC 6000 machines, a shift is obtained by a tree network in which you have as many levels as you have bits in the count of the possible number of shifts. If you expect to have to shift by as many as 63 places, it requires 6 levels in the shifting network. The first level shifts 0 or 1, the next 0 or 2, the next 0 or 4 and so on. You set up gates according to the bits in the shift count and let things percolate through the tree, which it does in 6

delay times (it only requires time to set the gates and to transmit through the gates). Shifts are simple; it is more interesting to count how many shifts are required, which in binary is made more efficient by noting that most shifts are small; so your counter might decode the first 6 bits to see how many are zeros. If they are all zeros, you know you have a more complicated job but that doesn't happen very often.

I haven't discussed these things in the previous section because I don't think they are important to today's technology, although they were important in earlier times.

The case for binary is not overwhelming, as can be seen. But it does avoid certain inconveniences in error analysis. The bulk of that inconvenience does not fall upon the users, but rather upon people who have to provide special subroutines for those users. Most people do error analyses of only the most superficial kind, which is generally adequate, if the number of digits they are manipulating is rather more than twice the number of digits needed to represent their data. If the number of digits in the machine is large enough, the base of the machine is relatively unimportant. It is hard to believe that binary or hexadecimal as a base can have transcendental importance, since people have gotten along with decimal for at least a millenium.

15. BASE CONVERSION

Arguments have arisen from misconceptions centering around what you mean when you write down a number; do you mean something other than what you have written down? If I write down 31415, some people say that is an integer. If I write 31415., they say it is from a set of real numbers:

$$31414.5 \leq 31415. \leq 31415.5$$

If any number from that set is acceptable, people who do binary-decimal conversions would be much happier.

But this leads to serious troubles. If you say 2. in FORTRAN, you'd be upset if somebody converted that to 1.99...9 on a binary machine; machines have been known to do that.

The difficulty arises because you try to read too much into a simple string of digits, so much that the string can no longer stand for itself.

This problem became acute in PL/1, where different machines would read the same code differently. A string might be converted in single precision, but if you added a zero, it would be converted in double precision and truncated, giving different final results. The bit string representations for 31415. and 31415.0 might actually be different in the machine.

The mistake arises in an innocent but misguided attempt to read more out of a string of digits than is put there. If people had said they would do the conversion to infinite precision (in principle) and then invoke conventions for packing, they would have been much better off. If you write down a string that looks like a number, it would be considered to be a precise real number. What happens to that number when converted depends on where you want to put it. Floating point numbers come under one set of conventions, integers under another.

If you are doing binary-decimal conversions, it is necessary to compute to more accuracy than is requested, in order to have something to round. To get single precision, your table of constants will need to be to double precision and the conversion done with double precision hardware and the result rounded to single precision. Then the job is done correctly except for those miserable cases that fall halfway between; in binary-decimal those cases can be characterized.

Double precision conversion, using double precision hardware is sloppy. You really need some extra bits around and no machine provides those.