AD-769 124

IMPLEMENTATION OF ALGORITHMS. PART I

W. Kahan

California University

Prepared for:

Office of Naval Research

1973

DISTRIBUTED BY:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE
5285 Port Royal Road, Springfield Va. 22151

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Computer Science Division Sciences Department of Electrical Engineering and Computer / University of California, Berkeley	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

IMPLEMENTATION OF ALGORITHMS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Technical Report, 1973

5. AUTHOR(S) (First name, middle initial, last name)

W. Kahan

6. REPORT DATE 1973	7a. TOTAL NO. OF PAGES ~~355~~ 339	7b. NO. OF REFS 0
8a. CONTRACT OR GRANT NO. N00014-69-A-0200-1017 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report 20 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

These are the notes for a lecture-course given in 1971 and 1972.

Examples are presented, and sometimes analysed in detail, to reveal the unpleasant implications for scientific computation of flaws in the design of the arithmetic unit and in the supervisory software associated with it. Attempts to axiomatize floating point arithmetic are discussed and the reasons why they are irrelevant. It is shown that Interval Arithmetic can be misleading or helpful depending on the way it is used. Some factors affecting a choice of radix base are elucidated.

Security Classification

14 KEY WORDS	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
Floating Point Arithmetic						
Significant Digits						
Error Analysis						
Over/Underflow						
Interval Arithmetic						
Base Conversion						

AD-769124

IMPLEMENTATION OF ALGORITHMS

PART I

Technical Report 20

W. Kahan

1973

D D C
NOV 13 1973
B

Lecture Notes By

W.S. Haugeland and D. Hough

Department of Computer Science
University of California
Berkeley, California 94720

Reproduced by
NATIONAL TECHNICAL
INFORMATION SERVICE
U S Department of Commerce
Springfield VA 22151

1973

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

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

PART I

0. Introductory Remarks: Motivation and Outline
1. Significant Digits, Cancellation, and Ill-Condition
2. Rules for Floating Point Arithmetic
3. Cost of the Rules
4. Arithmetic on the CDC 6400[1]
5. Software Conspiracy and the Cost of Anomalies
6. Execution Time Errors
7. Proof of a Numerical Program -- the Quadratic Equation
8. Modifying the Quadratic Equation Solver to Avoid Unnecessary Overflow and Underflow
9. How Can We Add Up a Long String of Numbers? -- Standard Pseudo-Double Precision Algorithm
10. How Can We Add Up a Long String of Numbers? -- Magic Constant Arithmetic
11. How Much Precision Do You Need -- In General?[2]
12. Interval Arithmetic
13. What Claims Should We Make for the Programs We Write?
14. Which Base is Best?
15. Base Conversion

PART II

16. An Eigenvalue Calculation Demanding Little From the Hardware
17. How Much Precision Do You Need to Solve a Cubic Equation?[3]
18. How Should We Solve a Non-Linear Equation?
19. Construction and Error Analysis of a Square Root Routine
20. Students' Report on Improved Versions of CDC SQRT, CABS, and CSQRT[4]

Appendix I. Students' Report on Arithmetic Units in Various Machines[5]

Appendix II. The RUNW.2 Compiler for CDC Fortran[6]

---

[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 θ (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 <u>need not lie between</u> 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) <u>Computation of elementary functions</u>: 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 <u>computing</u> and start trying to prove a <u>theorem</u> 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

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

Note also the case

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

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 <u>performing an exact calculation and then rounding</u>, 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:

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

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

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 <u>no</u> 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 <u>reveals</u> 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 <u>divided difference</u>, which is a function of two points x_1 and x_2:

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

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,

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

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

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

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

With this scheme, cancellation is never a problem. And the moral of the story is that we lose accuracy by <u>rounding</u>. 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	, $M \leq I < 2M$.

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 <u>discrete</u> <u>finite</u> set.

We formulate rules for desirable sets of numbers.

<u>Rule #1</u>: 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.

<u>Rule #2</u>: 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 $f(x) = \text{constant}$

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 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?

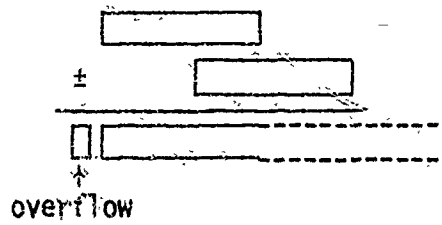
The distributive law is satisfied precisely. But only one of the integers 2 and 3 is representable in this system! [Is this worse than 2 and $\sqrt{2}$ not both representable?] This system has never been fully explored, so that we can't say that it's better or worse than the usual. Certain little tricks 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

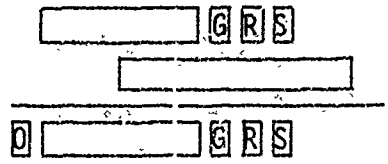
Our purpose is to demonstrate why our rules are too expensive to implement. Some of these costs are begrudged unjustifiably, though for others there are fairly good reasons.

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:



0 = 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 <u>more</u> 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 <u>accuracy</u> is better, the <u>uncertainty</u> 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

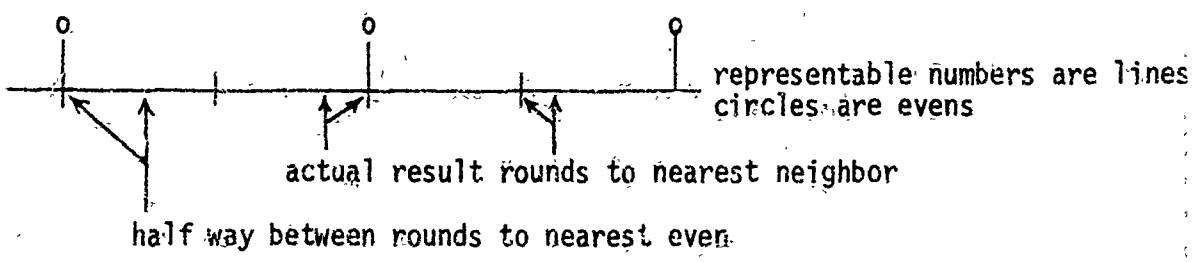
$$a + b = x + y \quad ,$$

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

<u>Exercise</u>: 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?

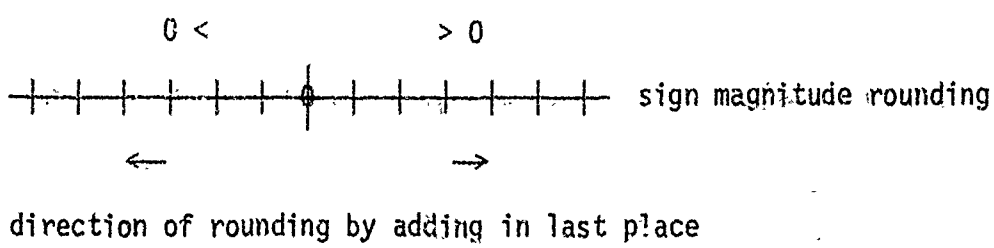
Answer: You round a negative number to its nearest neighbor unless it is half way between. That has nothing to do with where zero is.



You don't care where zero is in rounding.

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

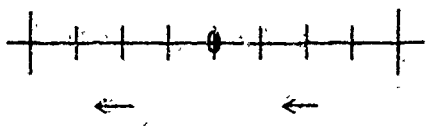
Answer: Sign-magnitude machines do just add 1 in the first bit to be discarded.



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?

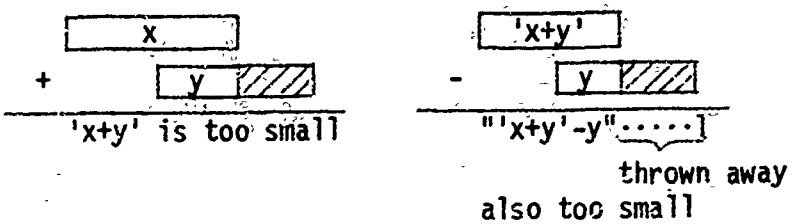
Has "bias" or "drift" been eliminated by this construct?

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.

<u>Test on arithmetic already known</u>

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.

<u>Example</u>. $x_n = 1.00001101$ 9 significant bits

$$\begin{array}{rl} y = & .100000001 \\ \hline 'x_n+y' = & 1.100011011 \end{array} \Longrightarrow \begin{array}{rl} & 1.10001110 \quad \text{stored 'x+y'} \\ -y: & .10000001 \\ \hline & 1.000011001 \end{array}$$

$$1.00001110 \Longleftarrow 1.000011001$$

when stored

stored(stored(x+y) - y) $\neq$ 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.

<u>Same example by rounding to nearest even.</u>

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.

<u>Example</u>: $\sqrt{z}$, $z > 0$

```
    y_θ = (1. + z)/2.
1   y_N = (y_θ + z/y_θ)/2.
    if (y_N .EQ. y_θ) go out
    else y_θ = y_N
    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 $\sqrt{z}$.

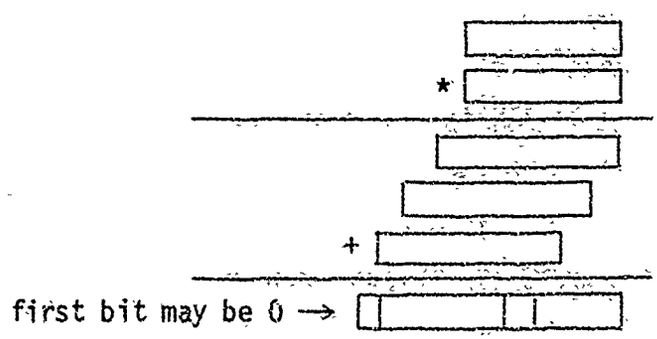
If drift does not exist, it is easy to prove that this terminates. You can also prove that it will terminate for rounding machines. But it might not terminate for machines which truncate.

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	C	I

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 $e = -\Omega$, 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.

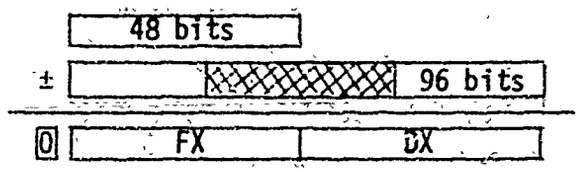
There is a question yet of what constitutes a zero. The multiply and divide units treat any quantity with $e = -1023$ as a zero. This was done to speed processing of sparse matrices by checking for zero factors in advance of multiplication. But the add-subtract logic checks all sixty bits and calls the 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 . 0 0 0	0 0 0 0
- . 1 1 1	1 0 0 0
0 . 0 0 0	1 0 0 0
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 <u>integer</u> subtraction while the divide unit will receive the result of a <u>floating point</u> subtraction which may well be zero. To get by on a CDC machine we could write

```
IF(X-Y.NE.0.0)...A/(X-Y)    ,
```

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:

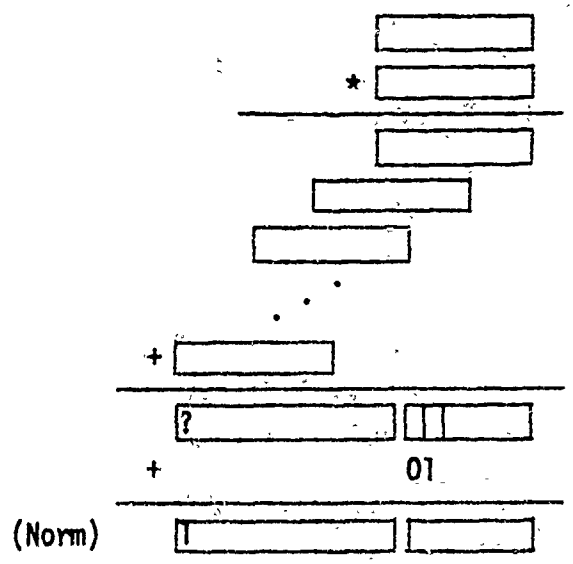
```
  1 1 1 1 1         1
+       1 0         0 0 0 1
-----------------------------
  1 0 0 0 0 1 ——— 1 0 0 1 = 32
```

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

<u>Rounded Multiplication</u>

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 <u>prior</u> 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        45
B = 19      × 19
            0855
             05
            0860 → 860 as the answer

C = 95        95
D = 9.0     9.0
            8550
              05
            8555 → 850 as the answer
```

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

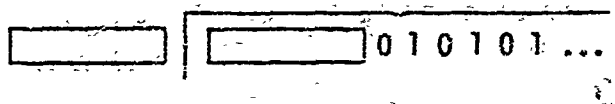
Answer: I used a table of factors, taking a number that had lots of factors and which had a leading digit that was large. I wanted it to be an eight, or the example doesn't work out so nicely.

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

```
FX0    X1*X2
DX3    X1*X2
RX0    X0+X3    .
```

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



On the <u>average</u>, 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.

<u>Unsolved Problem</u>: 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.

<u>Peculiarities of CDC Roundoff Error</u>† (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 <u>rounded</u> reciprocal divide (1.0/3.0) and the second is a <u>truncated</u> 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 θ	θ
T = (((Y-X)-X)-X)-X	0 or θ	∞
U = 1.0/T	∞ or θ	0
V = X/Y	$\frac{1}{4}$ or θ	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 13th 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 $\frac{B}{X} = 2^{-975}\left(\frac{10^9}{2\cdot 10^9+1}\right) < 2^{-976}$ and is therefore set to zero as an underflow

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

$\frac{D}{X} = 2^{-975}\left(\frac{1+10^{-9}}{2+10^{-9}}\right)$ 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

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

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!

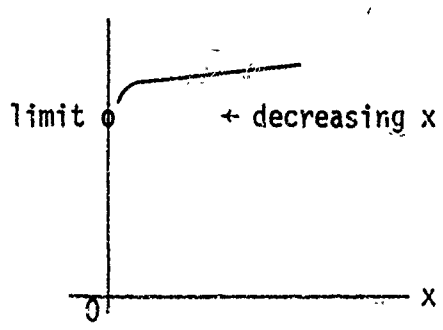
Exercise: What should be done on the CDC 6400 to compare two 60-bit integers to find the larger?

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 $x \to 0$).



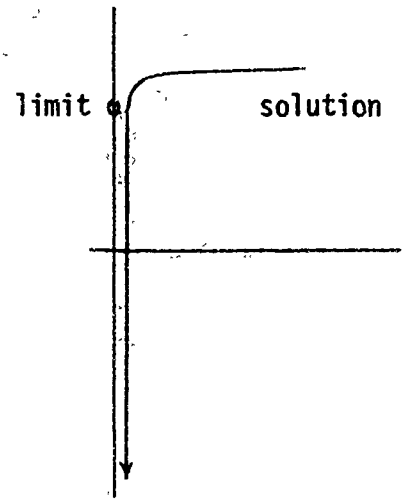
He couldn't show analytically that the limit existed so he turned to numerical methods.†

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.

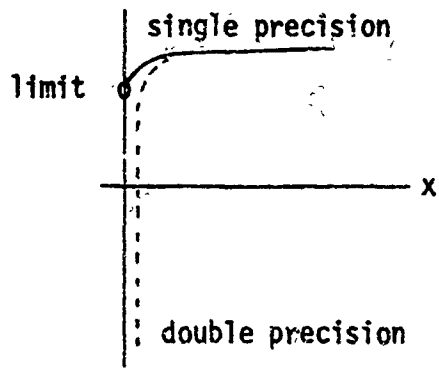
By now he had used up lots of machine time, and seemed at the end of what otherwise might have been a promising Ph.D. thesis.

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.

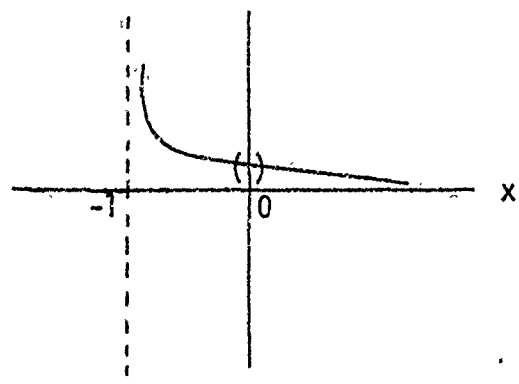


At this point I became interested. He was interested, of course; the single precision answer said he'd get his Ph.D., the double precision that he wouldn't.

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 = 1.0000xxx
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:

$$\Delta f(x_1, x_2) = \frac{\log(x_1) - \log(x_2)}{x_1 - x_2}, \quad x_1 \neq x_2$$
$$= \frac{1}{x} \qquad , \quad x = x_1 = x_2 \quad .$$

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

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

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

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, deter him from his project.

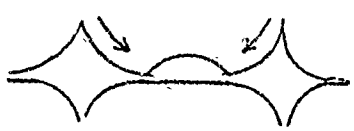
It is very unlikely that a rounding error in a floating point operation would cause a bridge to collapse, because people usually don't trust computer results. They build prototypes and throw in fudge factors.†

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 locked 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) .

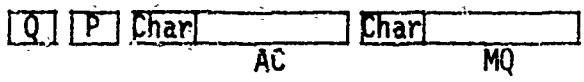
at the end of the job. A similar message was available for underflow. A logical extension to this system would be to include a message

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 <u>end</u> 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. <u>Counting mode</u> 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

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

$d \leq 0$ 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 <u>can't</u> 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

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

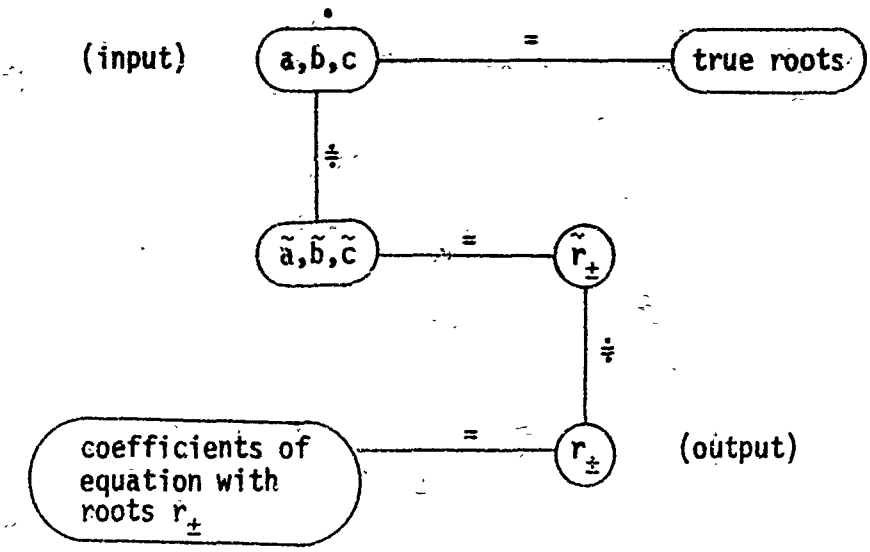
We see that the relative difference in the coefficient c is $1 + \frac{1}{N^2-1}$. That is, the computed roots were the correct roots of an equation whose coefficients differ from the original ones by less than one unit in the last place. Unfortunately, this statement also is not true for our program 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 $\sim$ 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 $d \leq 0$, $\tilde{r}_\pm = \tilde{r}_R \pm i\tilde{r}_I$, $r_R = (1+\delta_3)\tilde{r}_R$, $r_I = (1+\delta_4)(1+\rho)\sqrt{(1+\mu_1)(1+\mu_2)}\tilde{r}_I$. Now when $d > 0$, define

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

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

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

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

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

<u>Theorem</u>. $\left|1 - \frac{\tilde{r}_\pm}{R_\pm}\right| \leq \sqrt{|\gamma|}(\sqrt{|\gamma|} + \sqrt{1+|\gamma|})$.

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

<u>Proof</u>. 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 $|R_-| \leq |R_+|$ so $|z| \leq 1$. Then $|\delta_+| = |-z\delta_-| \leq |\delta_-|$ so we only concern ourselves with δ_-. It satisfies

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

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

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

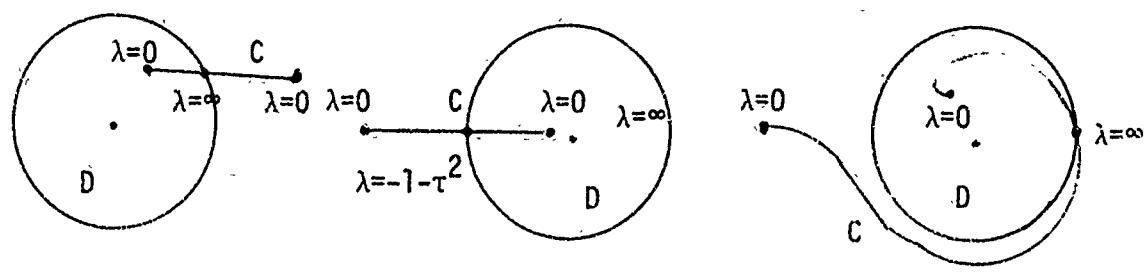
$\delta(z,\tau)$ is a holomorphic function of z except for those critical values where both roots of (*) are equal in magnitude. The critical values are those satisfying

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

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,

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

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

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

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 <u>bound</u> 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 $\sum_{j=1}^{N} x_j y_j (1+\zeta_j)$ 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 $\sum_{j=1}^{N} x_j y_j (1+\zeta_j^2)$. 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 $|b| \pm \sqrt{b^2-ac}$, 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.

$$\begin{array}{lll} ac: & 2^m & \times \boxed{1000\cdots\cdots0000} \\ b^2: & 2^{m-1} & \times \boxed{1111\cdots\cdots1111} \end{array}$$

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, $b = 111\cdots11$. Then we would have:

$$\begin{array}{lll} ac: & 2^m & \boxed{1\cdots\cdots00}\ \boxed{0\cdots\cdots0} \\ b^2: & & \boxed{011\cdots\cdots11}\ \boxed{0\cdots\cdots0}\ 1 \leftarrow \text{shifted off} \\ \hline ac-b^2 & & \boxed{0\cdots\cdots01}\ \boxed{0\cdots\cdots0} \end{array}$$

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

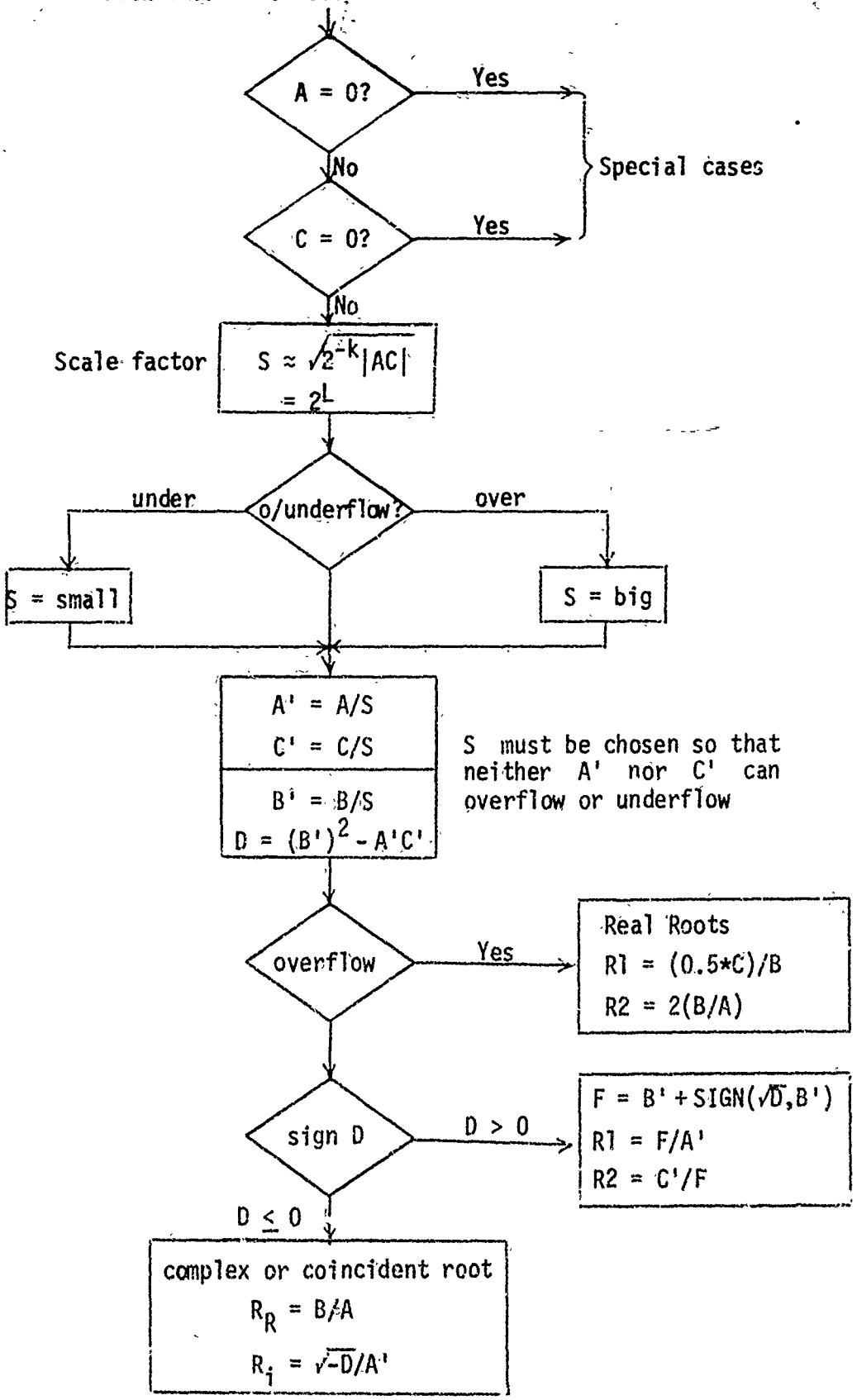
$$Ax^2 - 2Bx + C = 0 \quad .$$

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 $Ax^2 - 2Bx + C = 0$



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^{-30}. 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

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

$$C = 2^{-1024+48} \quad \text{smallest operand}$$

($2^{-1024+47}$ 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.

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

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

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 $\quad 96 - 2 - K = 1$

or $\quad K = 93 \quad .$

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. $A = 2^{1022+48}(1-2^{-47})$, $C = 2^{-1024+48}$.

$$2^{-93}|AC| = 2^{96-2-93}(1-2^{-47})$$
$$\sqrt{2(1-2^{-47})}(1+\epsilon) \approx \sqrt{2} \approx 1.4$$

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.

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

When I take $\sqrt{1}$ 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.

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

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

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

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

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

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:

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

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

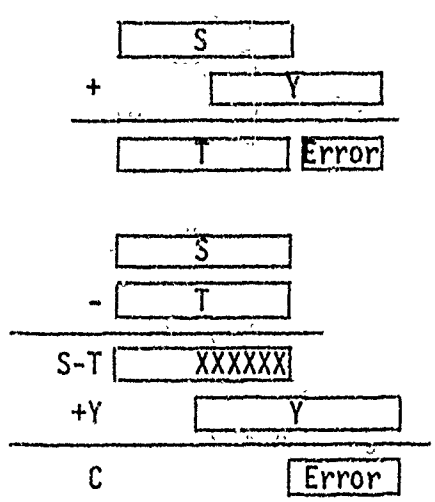
program. To avoid having the double precision propagate into other parts of the program where it is not necessary, truncate S to another single precision variable and use that variable elsewhere.

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

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

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 precision. 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:

```
  [10·····0]
+         [XXXYYYY]
-----------------
  [10····CXXX]
- [10·····0]  subtract the 1
-----------------
  [0·····0XXX]
```

```
  [XXXYYYY]
- [XXX0000]
-----------
  [0000YYYY]
```

now the number is broken into approximately equal parts

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. |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)	$y_j = (x_j+c_{j-1})(1+\eta_j)$
T=S+Y	$s_j = (s_{j-1}+y_j)(1+\tau_j)$
C=(S-T)+Y	$c_j = ((s_{j-1}-s_j)(1+\sigma_j)+y_j)(1+\gamma_j)$
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:

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

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

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

<u>Exercise</u>. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $i \leq n$ 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

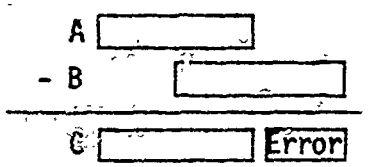
$$C = (S-T)+Y$$

in order to get

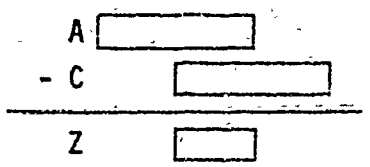
$$c = ((s-t)(1+\sigma)+y)(1+\gamma) \quad .$$

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 $z \doteqdot b$ 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.

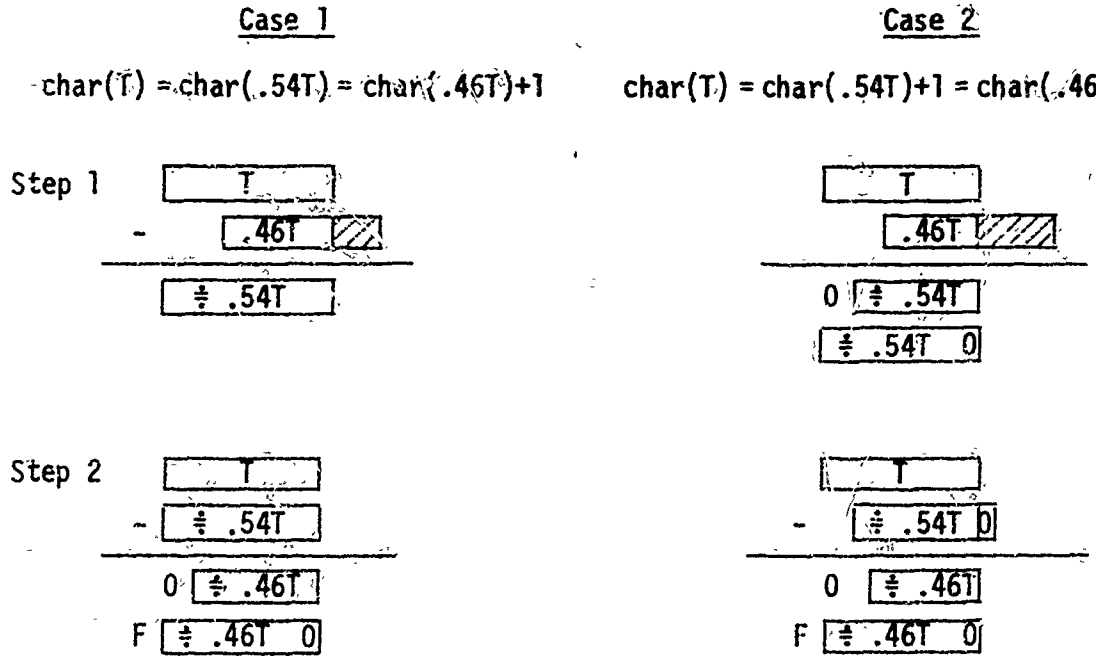
We will replace the statement

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

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:

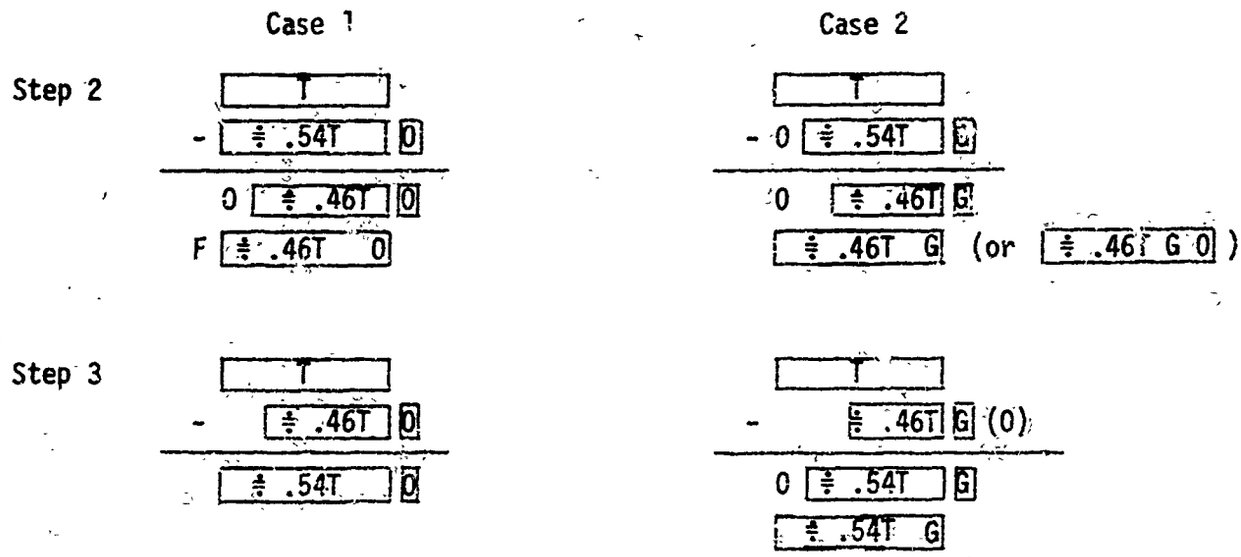


F has been fabricated in such a way that it can be subtracted precisely from T. The proof of this statement depends on the machine. First, consider a machine like the 650 which does arithmetic by dropping right-shifted bits. Such arithmetic is illustrated in the picture. F is formed from a number with 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 6600. 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.

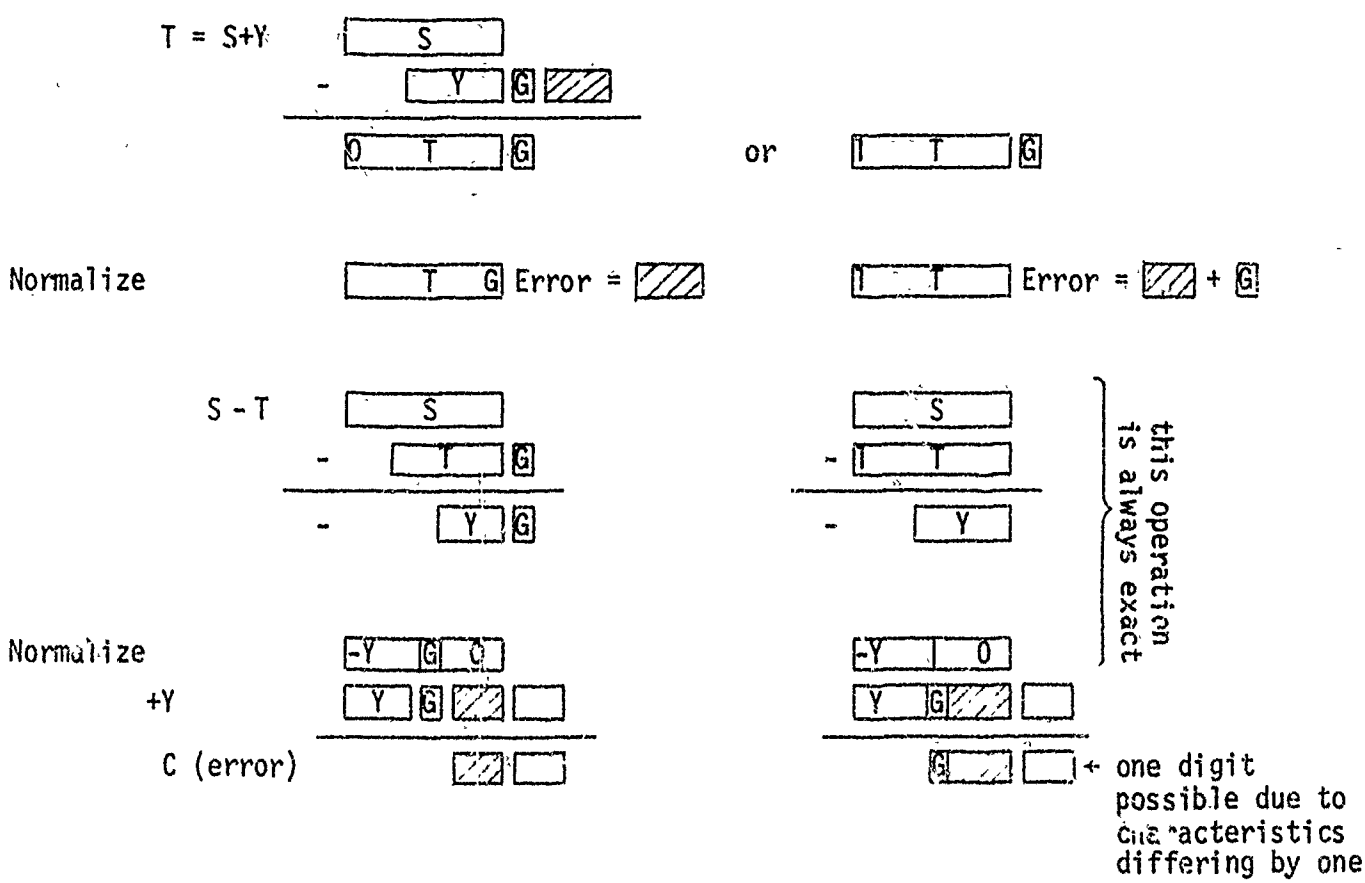
Suppose finally that a whole guard word is kept in the subtraction and used in the subsequent normalization as is done by the IBM 7094. Case 1 is precisely as in the case where digits are discarded. The guard words are entirely zero. Case 2 is precisely as in the case of the guard digit. The only extra 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 $\text{sign}(S) \neq \text{sign}(Y)$. 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 $S \doteq T$, 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. $S \doteq T$ so it might have relative alignment 0 or 1 in the addition unit:

relative alignment 0 relative alignment 1

Now $F \doteq .46T$, 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,

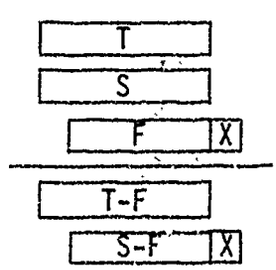
$$\text{char}(T) \geq \text{char}(S) \geq \text{char}(T-F) \geq \text{char}(S-F) \geq \text{char}(F) \quad ,$$

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 comouter (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 $\longrightarrow$ 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.

more significant	least significant	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:

$$\begin{array}{rl} & \boxed{\quad x \quad} \\ + & \boxed{\quad X \quad} \\ \hline & \boxed{X+1} \; +\alpha \end{array} \qquad \alpha = 1, 0$$

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:

$$\begin{array}{rl} & \boxed{X+1+\alpha} \\ - & \boxed{\quad X \quad} \\ \hline & \boxed{1+\alpha} \end{array}$$

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.

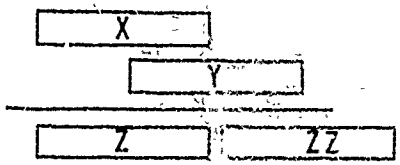
Question: But what if multiplication is funny? The old 360 way of multiplying lost the bottom digit of single precision, if a normalizing

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) $\neq$ 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:

$$\boxed{2^0 \mid \overbrace{.100000}^{t}} \sim \frac{1}{2} \pm 2^{-t-1}$$

$$\boxed{2^3 \mid .000100} \sim \frac{1}{2} \pm 2^3 \cdot 2^{-t-1}$$

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:

$$\text{if } a \le b, \quad [a,b] = \{x: a \le x \le b\}$$

$$\text{if } a < b, \quad (a,b) = \{x: a < x < b\}$$

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 <u>both</u> 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.

<u>Triplex</u> 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:

$$[a,b] \sim \{\frac{a+b}{2}, \frac{a-b}{2}\}$$

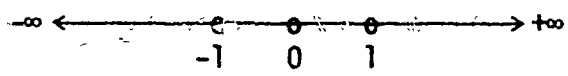
$$[x-\delta, x+\delta] \sim \{x, \delta\}$$

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.

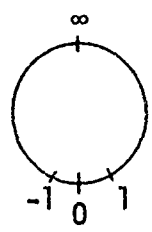
To secure this storage advantage you must give up something important of a practical nature. We would like to use the built-in two word double or complex handling facilities of standard Fortran compilers to implement interval arithmetic. Then we could avoid rewriting much of the Fortran compiler.

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 $\vartheta = [+\infty,-\infty]$, 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 <u>sub</u>-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 <u>inclusion monotonicity</u> 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., <u>Topics in Interval Analysis</u>, 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 <u>scalar</u> 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

$$\phi(\xi) = \frac{\xi}{\xi+2} = \frac{1}{1+\frac{2}{\xi}} \quad .$$

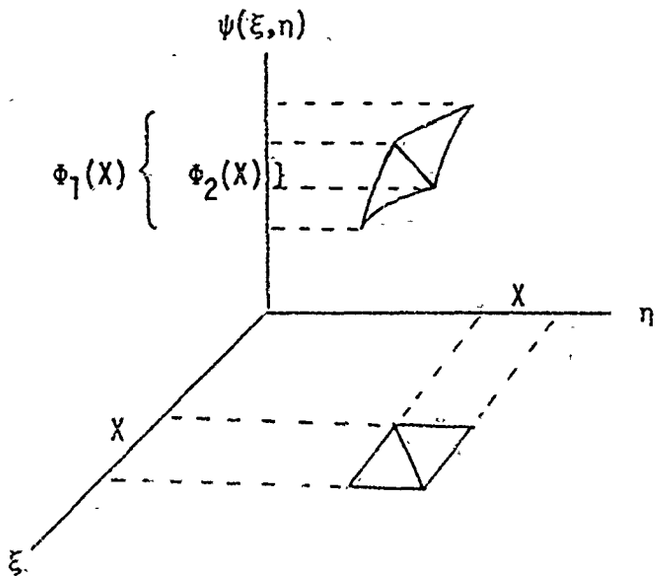
Then $\Phi_1(X) = \frac{X}{X+2}$, $\Phi_2(X) = \frac{1}{1+\frac{2}{X}}$. These interval functions are distinctly different and

$$R\phi(X) = \Phi_2(X) \subseteq \Phi_1(X) \quad .$$

The variable ξ occurs in the second expression only once. When we evaluate

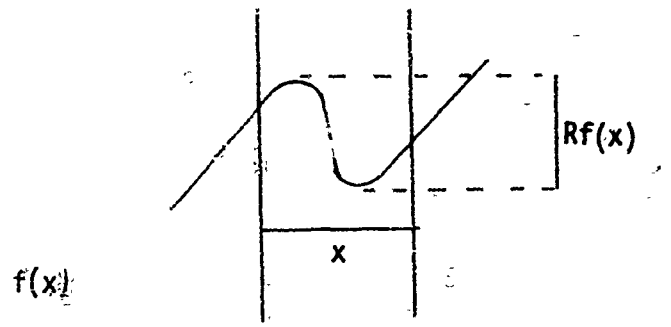
$\Phi_1(X)$ we actually evaluate $\psi(X,X)$ where $\psi(X,Y) = \frac{X}{Y+2}$. That is, the two occurrences of ξ are independent, so that $\psi(\xi,\eta) = \frac{\xi}{\eta+2}$. This is called the <u>Independence Phenomenon</u>.

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 <u>no</u> 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 <u>function</u> $f(x_1;x_2;\ldots;x_n)$ has many corresponding <u>expressions</u> $F(x_1;x_2;\ldots;x_n)$. 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

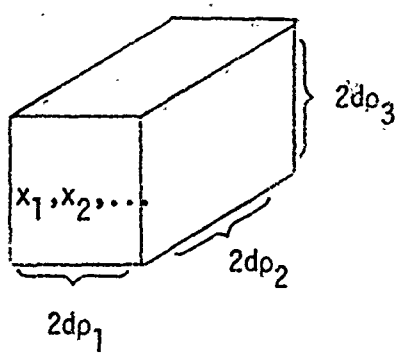
intervals X_j are so small that they are infinitesimal, which means

$$X_j = \{x_j + dx_j : |dx_j| \leq d\rho_j\} \quad .$$

Then we find that

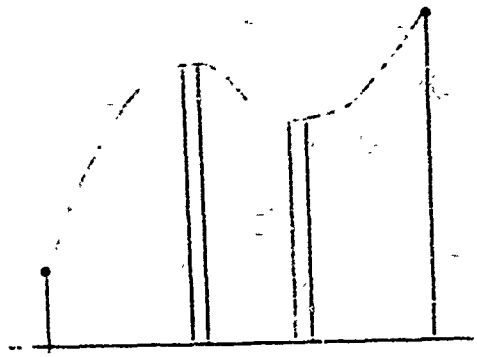
$$Rf(X_1;\ldots;X_n) = f(x_1;x_2;\ldots;x_n) + \bigcup_{|dx_j|\leq d\rho_j} \sum_j \frac{\partial f}{\partial x_j} dx_j \quad .$$

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

$$X = x_0 + [-\Delta, \Delta] \quad .$$

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

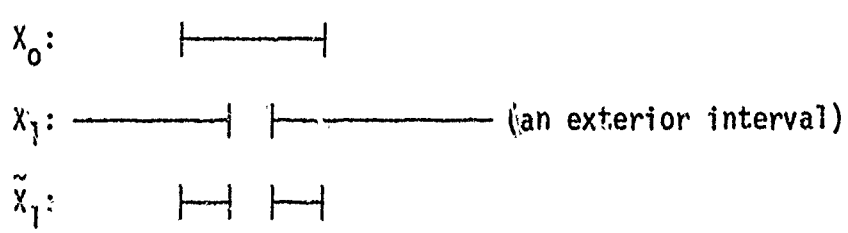
$$\tilde{X}_1 = X_1 \cap X_0 \quad .$$

Now every root in X_0 also lies in $\tilde{X}_1$, 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.

<u>Unsolved Problem</u>. 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:

$$f'(x) = 1 + g(x) \qquad g_0(x) = \frac{1}{2} - \frac{1}{2}\,\frac{(x^2-1)^2 + 8(x-1)^2}{(x^2+3)^2}$$

$$g_1(x) = \frac{2}{\left(\frac{8+(x-1)^2}{1-(1-\frac{2}{x+1})^2} - 4\right)}$$

$$g_2(x) = \frac{8x}{(x^2+3)^2}$$

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

$$
\begin{aligned}
Rg(X) &= G_0(X) \quad \text{if} \quad X \subseteq [0,1] \\
&= G_1(X) \quad \text{if} \quad X \subseteq [0,1] \quad \text{or} \quad X \subseteq [\infty,-1] \\
&\subset G_i(X) \quad \text{for almost all other } X \text{ except degenerate intervals and } [-1,1], \quad i = 0,1,2.
\end{aligned}
$$

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,

$$SQRT(x) = \sqrt{x} \pm \frac{1}{2}\,\text{ulp} \quad \text{if} \quad x \geq 0 \quad .$$

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.

$$\text{SQRT}(x) = \sqrt{x} \pm .50000163 \text{ ulps} \quad .$$

The 7094 had only 27 bits of precision so the subroutine clearly had to compute SQRT to about 100 ulps <u>in double precision</u> to make such a claim.

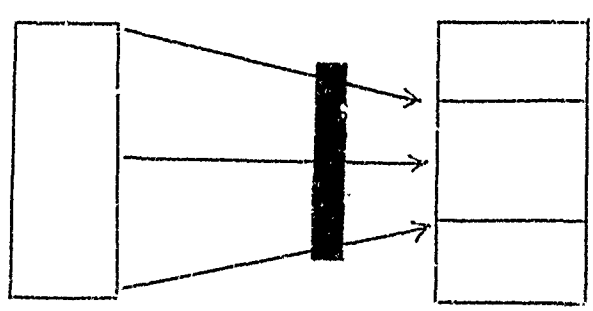
In the 7094 there are eight characteristic bits and 27 integer bits. Counting only positive normalized numbers there are 2^{34} different numbers. Of these many just differ by a power of four and are therefore essentially similar from the point of view of the square root routine. Of these 2^{34} numbers, 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 <u>impossible</u> 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

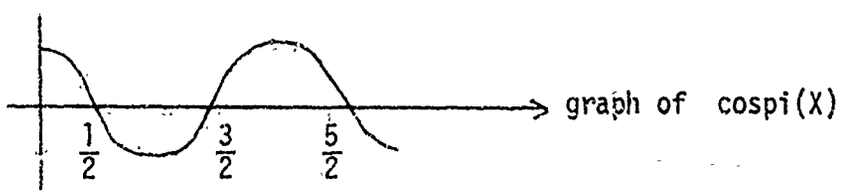
Let us consider some reasonable and plausible bounds for some other functions. These numbers are in ulps for routines on the 7094.

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.

$$\cos \pi\left(\frac{2k+1}{2}+\xi\right) \approx \pm\xi \pm \xi^3/6 + \cdots$$

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.

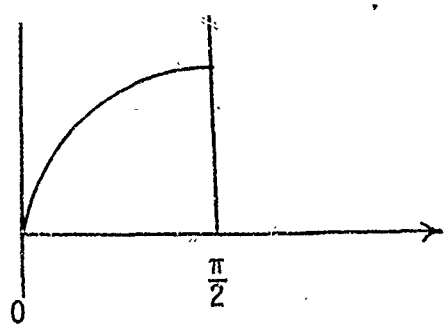
as you want using the power series.

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

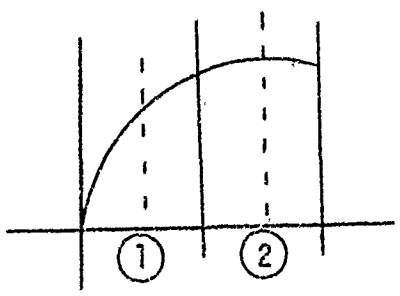
Let's see how this uncertainty in π contaminates our ability to compute the trig functions.

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.

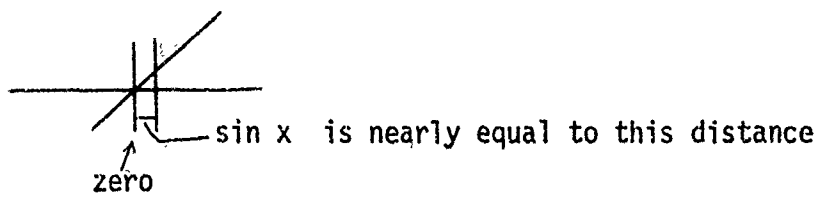
After this the terms get smaller; but you see you'll have to carry 8 decimal digits over and above the 14 you wanted in your answer. That is a more serious fact than having to compute many terms; they just use a do loop and take a few microseconds. But the extra digits require double precision and that 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, $\sin x \approx x$ (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 $x = 10^{-100}$, $\sin x \doteq x$ 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.

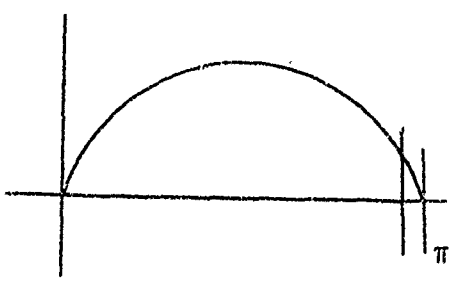
Quadraproduction (Argument Reduction)

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.

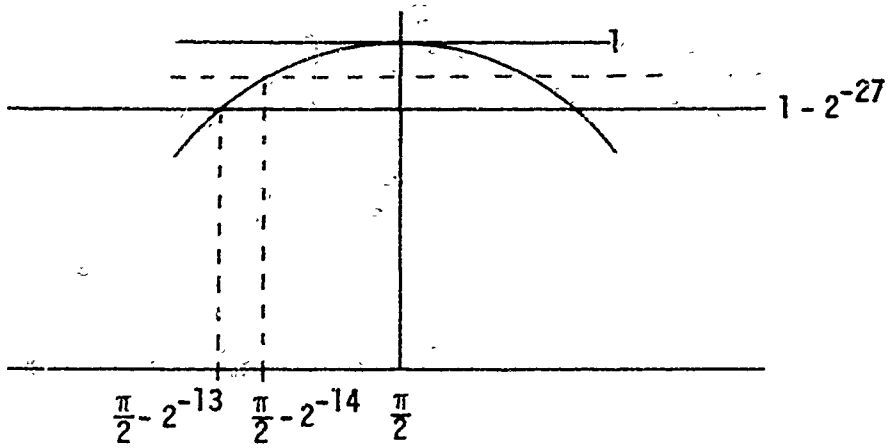
Suppose someone did demand $SIN(X) = \sin(x) \pm \frac{1}{2}$ ulp. Unless x is restricted to an unreasonably small domain this isn't possible. The first step in the SIN routine is to find the fraction $\frac{x}{2\pi} - \lfloor\frac{x}{2\pi}\rfloor$. Unfortunately the value of π is not available in the 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

$$\mathrm{SIN}(X) = (1+\varepsilon)\sin(x(1+\xi)) \ , \quad |\xi| < 10^{-15} \ (\approx 1 \text{ ulp } \underline{\text{double}} \text{ precision}), \quad |\varepsilon| < .52 \text{ ulp} \ .$$

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 or was zero. It was also true that

$$\mathrm{ABS(SIN(X))} \le 1.0$$

$$\mathrm{ABS(COS(X))} \le 1.0$$

$$\frac{\mathrm{SIN\ X}}{\mathrm{X}} \le 1.0$$

$$|1 - (\mathrm{DBLE(SIN(X))**2 + DBLE(COS(X))**2})| \le 2 \cdot 10^{-8} \quad .$$

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:

$$\mathrm{SINPI(X)} = (1+\varepsilon)\sin(\pi x)$$

$$\mathrm{COSPI(X)} = (1+\varepsilon)\cos(\pi x) \quad .$$

Then $\xi = 0$ because argument reduction involves only integer subtraction.

You could reasonably expect COS(X) to satisfy:

$$\mathrm{COS(X)} = (1+\gamma)\cos(x(1+\xi))$$

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:

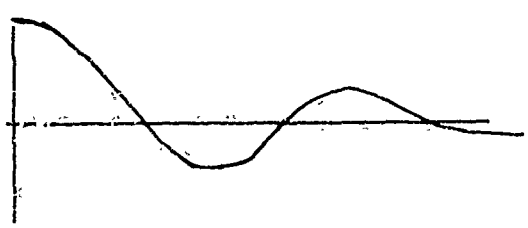
$$\frac{\sin}{\cos} \approx \tan \quad \text{to within a few ulps} \quad .$$

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.

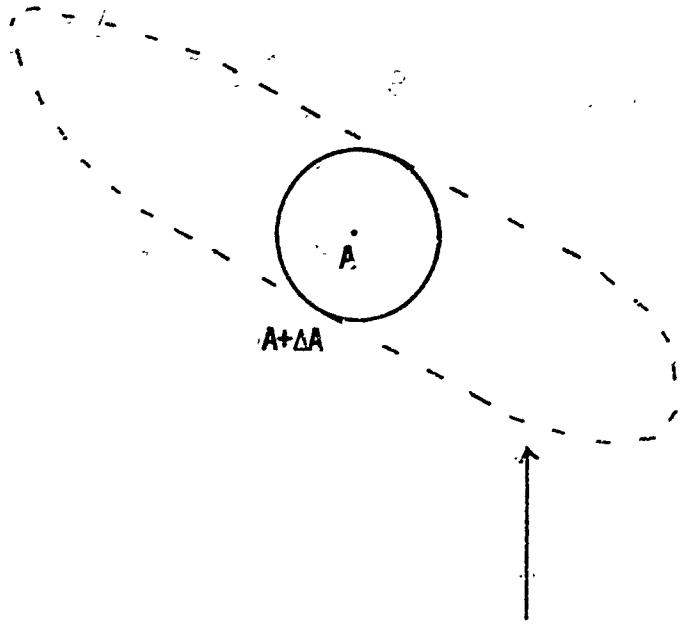
<u>Thesis project</u>: Prove that some standard algorithm does or does not always produce a result that satisfies condition (4).

The best result known is Wilkinson's:

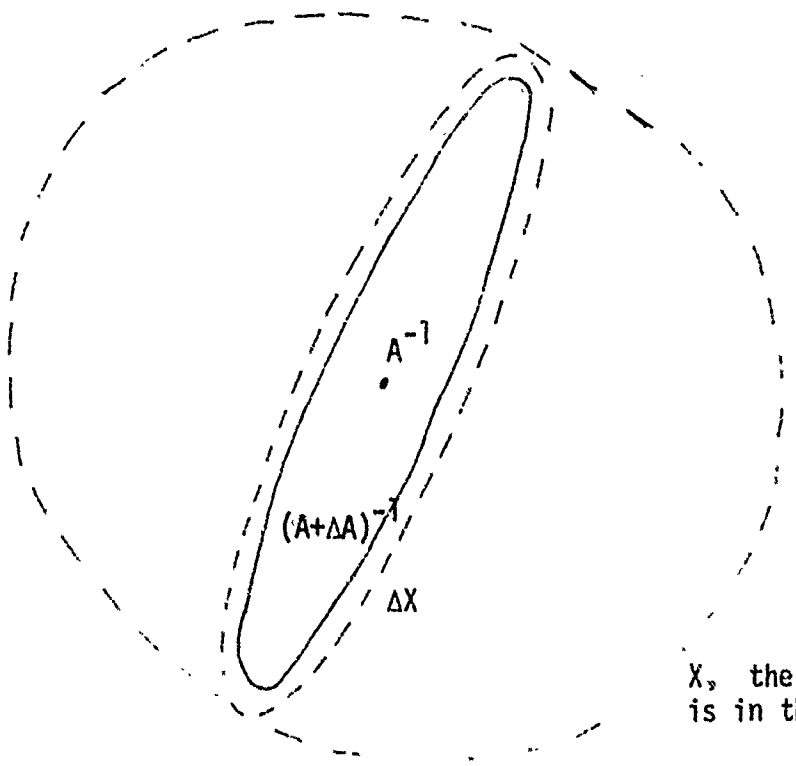
$$\|X-A^{-1}\| \leq \max\|(A+\Delta A)^{-1} - A^{-1}\|$$

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

$$\mathrm{LOG}(X) \approx (1+\epsilon)\log(X(1+\xi)) \quad .$$

(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:

$$A^B \approx \mathrm{EXP}(B*\mathrm{AL\emptyset G}(A)) \quad .$$

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."

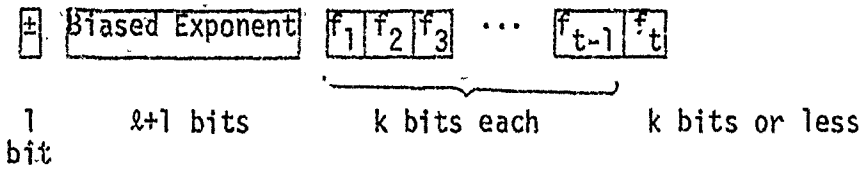
Being simple to explain to users seems to me to be the most important reason of all. Coding the routine is only half the problem. The other half consists of informing the users what exactly the subroutine accomplishes.

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 $2^{-k} \leq f \leq 1 - 2^{-kt}$, represented by t digits, and for some i in the range $-2^{\ell} \leq i \leq 2^{\ell} - 1$. 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.)

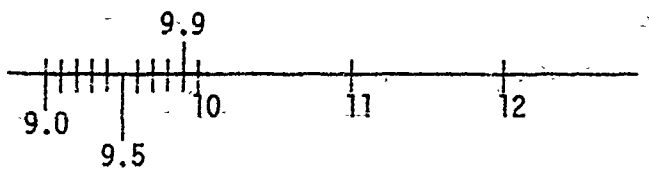
2. Another possibility is suggested by the fact that numbers of magnitude far from 1 occur much less frequently than numbers of magnitude near 1. Perhaps by some suitable encoding we could find a way to represent numbers near one with only a few bits for the exponent, and numbers far from one with more 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

$$\begin{array}{c} 1.00000\cdots \\ .\underbrace{11111}_{\text{kt 1's}} \end{array} \qquad \frac{1-(1-2^{-kt})}{1 \text{ or } 1-2^{-kt}} \doteq 2^{-kt} \quad ,$$

and the largest is

$$\begin{array}{c} \overbrace{1.0000\cdots 1}^{\text{kt bits}} \\ 1.0000\cdots 0 \end{array} \qquad \frac{(1+2^{k-kt})-1}{1 \text{ or } 1+2^{-kt}} \doteq 2^{k-kt}$$

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

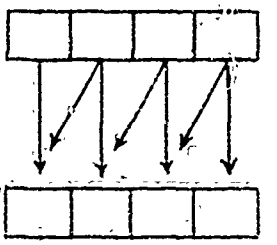
We conclude that the best value of the base is 2 or 4, based on the worst case of a jump in precision. Perhaps it would be more realistic to base our decision on some sort of average relative precision. For a number of plausible distributions of numbers the best base has been asserted to be 4. (Brent, (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

$$|q - X/Y| < b^{1-t} + \frac{1}{2}b^{-t} = (\frac{1}{2} + b)b^{-t} = \underline{\underline{(\frac{1}{2} + b) \text{ ulp of } q}}.$$

---

†"ulp" = "unit(s) in the last place..."

$\gamma = 1$ $\quad 1 \le x/y \le b$ so $\left|\frac{x}{y} - \frac{X}{Y}\right| \le \frac{1/2}{Y}(1+b) < \frac{1}{2}(1+b)b^{1-t}$, 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 $\left|\frac{x}{y} - \frac{X}{Y}\right|$ 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 $m \ll b^{t-1}$. 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 $n \ll b^{t-1}$ 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.

$2m(bm+n) \doteqdot b^{t-1}$ 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:

$$\left.\begin{array}{lll} \text{Round } X & \text{to} & x \equiv 1014 \\ Y & \text{to} & y \equiv 1017 \\ A & \text{to} & a \equiv 1000 \\ B & \text{to} & b \equiv 1007 \end{array}\right\} \text{committing in each case an error smaller than } \tfrac{1}{2} \text{ ulp.}$$

Now $x/y = .99\ldots$ will be in error by about $(\frac{1}{2}+\frac{1}{2} = 1)$ ulp, and its rounded value $q \doteqdot .99\ldots$ by about $\frac{1}{2}$ ulp more than that, i.e. by .00015. Similarly, the rounded value $r = .99\ldots$ 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

$$\begin{array}{lll} x/y = .99705015 & \text{so } q = .9971, & \text{but } X/Y = .99606899; \\ a/b = .99304866 & \text{so } r = .9930, & \text{but } A/B = .99403855; \\ & q - r = .0041, & \text{but } \frac{X}{Y}-\frac{A}{B} = .00203044. \end{array}$$

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.

15-3

IMPLEMENTATION OF ALGORITHMS

PART II

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

CONTENTS

PART I

0. Introductory Remarks: Motivation and Outline
1. Significant Digits, Cancellation, and Ill-Condition
2. Rules for Floating Point Arithmetic
3. Cost of the Rules
4. Arithmetic on the CDC 6400[1]
5. Software Conspiracy and the Cost of Anomalies
6. Execution Time Errors
7. Proof of a Numerical Program -- the Quadratic Equation
8. Modifying the Quadratic Equation Solver to Avoid Unnecessary Overflow and Underflow
9. How Can We Add Up a Long String of Numbers? -- Standard Pseudo-Double Precision Algorithm
10. How Can We Add Up a Long String of Numbers? -- Magic Constant Arithmetic
11. How Much Precision Do You Need -- In General?[2]
12. Interval Arithmetic
13. What Claims Should We Make for the Programs We Write?
14. Which Base is Best?
15. Base Conversion

PART II

16. An Eigenvalue Calculation Demanding Little From the Hardware
17. How Much Precision Do You Need to Solve a Cubic Equation?[3]
18. How Should We Solve a Non-Linear Equation?
19. Construction and Error Analysis of a Square Root Routine
20. Students' Report on Improved Versions of CDC SQRT, CABS, and CSQRT[4]

Appendix I. Students' Report on Arithmetic Units in Various Machines[5]

Appendix II. The RUNW.2 Compiler for CDC Fortran[6]

---

[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.

16. AN EIGENVALUE COMPUTATION DEMANDING VERY LITTLE FROM THE HARDWARE DESIGN

Our object, in considering specifications for numerical hardware and software, is not to make life easy for numerical analysts. Rather, it is to determine what features make it least likely that an architect designing cathedrals will have to get a Ph.D. in numerical analysis in order to use the computer efficiently.

However, we would also like to make it possible never to repeat an error analysis of an algorithm for every new machine that is manufactured or operating system that is written with previously unheard-of laws of arithmetic. Error analysis is such a burden that we should hope to do it only once.

The aim of this course is to describe the considerations that should be borne in mind by the designer of a new system or the repairman of an old one. We have seen that certain computations require rather stringent restraints on the way arithmetic is done. We will now demonstrate that some complicated calculations require little more than that the hardware be monotonic.

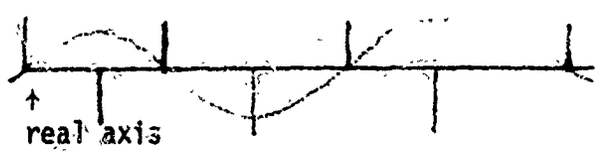
The eigenvalue algorithm is described in a Stanford Report ("Accurate Eigenvalues of a Tri-Diagonal Matrix," Stanford Computer Science Department Report #CS41 (1966)) and in Kahan's Notes on Error Analysis (1968) for the University of Michigan Summer School. The input is a real symmetric tri-diagonal matrix J:

$$J = J_N = \begin{bmatrix} a_1 & b_1 & & & & & \\ b_1 & a_2 & b_2 & & & & \\ & b_2 & a_3 & b_3 & & 0 & \\ & & \cdot & \cdot & \cdot & & \\ & & & \cdot & \cdot & \cdot & \\ & & & & \cdot & \cdot & \cdot \\ & 0 & & & \cdot & a_{N-1} & b_{N-1} \\ & & & & & \cdot & b_{N-1} a_N \end{bmatrix} \qquad b_0 \equiv b_N \equiv 0$$

All the eigenvalues are real. As in all symmetric matrices, if we strike off any row and column, the eigenvalues of the matrix left are interlaced with the original eigenvalues:

N Eigenvalues original matrix

N-1 Eigenvalues reduced matrix



The curve represents the polynomial $\det(J-\lambda)$ which vanishes at each eigenvalue.

We will exploit Sylvester's Inertia Theorem: If $J - x = LDL^T$, where L is non-singular (it will be lower triangular too), and D is diagonal, the number of positive, zero, and negative entries in D is equal to the number of positive, zero, and negative eigenvalues of $J - x$, respectively.

It seems remarkable that the theorem is true no matter which of the many D's one considers. In any event we have located an eigenvalue of J between x_1 and x_2 if the number of positive eigenvalues of $J - x_1$ is one less than the number of positive eigenvalues of $J - x_2$.

Let $U = DL^T$ so $J - x = LU$. Then L is almost always a non-singular unit lower triangular matrix and U is upper triangular.

$$L = \begin{pmatrix} 1 & & & & \\ \cdot & 1 & & 0 & \\ & \cdot & \cdot & & \\ & & \cdot & \cdot & \\ 0 & & \cdot & 1 & \\ & & & \cdot & 1 \end{pmatrix} \qquad U = \begin{pmatrix} u_1 & \cdot & & & \\ & u_2 & \cdot & & 0 \\ & & \cdot & \cdot & \\ & & & \cdot & \cdot \\ & 0 & & & u_N \end{pmatrix}$$

We see that the diagonal elements of U are the same as those of D. Also note that these triangular matrices remind us of Gaussian elimination. Therefore we summarize our algorithm as follows: Do Gaussian elimination without pivoting on $J - x$ to find the factors L and U. If we don't blow up on division by zero, the number of positive u's is the same as the number of eigenvalues of J greater than x. Then we could use a binary chop to test values of x to home in on any particular eigenvalue.

The algorithm for the u's is as follows:

$$u_1 = a_1 - x$$
$$u_n = a_n - x - \frac{b_{n-1}^2}{u_{n-1}}, \quad n = 2,\ldots,N .$$

This seems simple enough, but suppose u_{n-1} vanishes? We could fudge things by perturbing a_{n-1} by ε so that $u_{n-1} = \varepsilon$ for some tiny ε. Is this legitimate? There is a reassuring theorem. Suppose the eigenvalues of J, λ_i, are indexed in order so that $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_N$, and the eigenvalues of $J + \Delta J$, $\lambda_i + \Delta\lambda_i$, satisfy $\lambda_1 + \Delta\lambda_1 \le \lambda_2 + \Delta\lambda_2 \le \cdots \le \lambda_N + \Delta\lambda_N$. The theorem states that

$$|\Delta\lambda_j| \le \|\Delta J\| , \quad j = 1,\ldots,N$$

for some suitable norm. A suitable norm is $\|A\| \equiv \max_{x \neq 0} \sqrt{\frac{x^*A^*Ax}{x^*x}}$.

By choosing an ε small enough compared to the eigenvalue we seek and

setting $u_{n-1} = \varepsilon$, we can continue without worry. There are risks of overflow and underflow; the paper discusses these problems. For our present purposes we will assume nothing bad will happen if we replace u_{n-1} by a suitable ε.

Our program looks like the following

```
        .
        .
        .
        DO 9 I=1,N
9       BB(I) = B(I-1)**2                    (preparing the b²_{i-1})
        .
        .
        .
        U_0 = 1.0
        ν = 0
        DO 3 I=1,N
            U_I = (A(I)-BB(I)/U_{I-1})-X
            IF (U_I) 2,1,3
1           U_I = -ETA                       (if u_i = 0)
2           ν = ν+1                          (if u_i ≤ 0)
3       CONTINUE
```

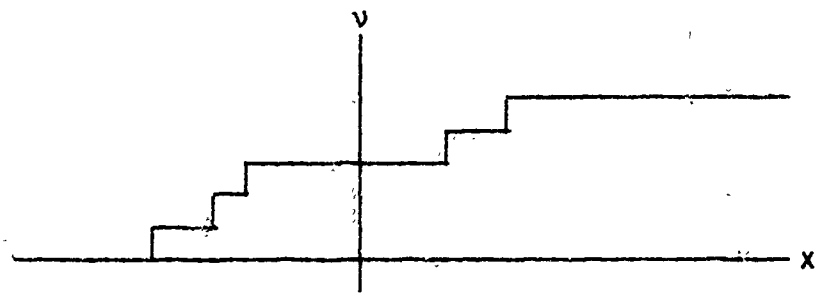
(The subscripts don't actually need to appear in the program.)

Then $\nu(x)$ = the number of eigenvalues $\leq x + \text{ETA}$ (?)

If ΔJ is negative semidefinite so that, for every vector v, $v^T \Delta J v \leq 0$, then $\Delta\lambda \leq 0$. By choosing ETA always negative we guarantee that the eigenvalues are always perturbed down slightly, never up. But there is the uncertainty that eigenvalues within ETA of x could be shifted to either side of x so that they could be counted either way -- hence the (?) in the previous equation. But ETA is usually smaller than a unit in the last place of the results we are going to quote in the end.

Aside from the question of whether the algorithm computes accurate eigenvalues, which we shall not consider here, there is the question of whether the subsequent logic dealing with ν could be thrown off because ν is

inaccurate due to rounding errors. In the absence of rounding errors, ν plotted as a function of x will look like



That is, it will be monotone nondecreasing with a jump at each eigenvalue. A program which expected a monotonic ν might conceivably hang up if ν were somewhere to decrease because of rounding or possibly the substitution of ETA for a 0.

Our purpose is to show that if we only assume that our arithmetic is monotonic, then ν will be monotonic despite rounding errors or ETA.

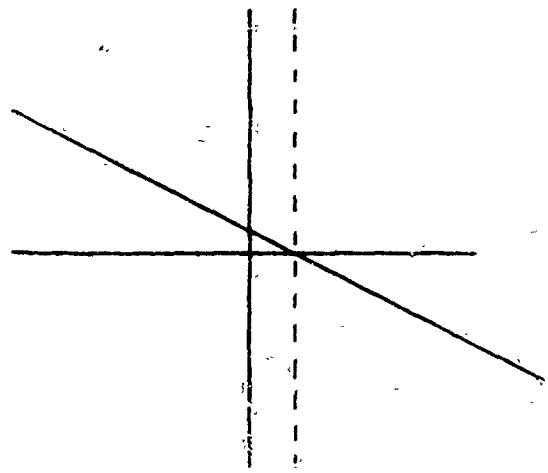
To see this we must plot u as a function of x. We know

$$J_N - x = \begin{pmatrix} 1 & & & 0 \\ \cdot & 1 & & \\ & \cdot & 1 & \\ 0 & & \cdot & 1 \end{pmatrix} \begin{pmatrix} u_1 & \cdot & & \\ & u_2 & \cdot & \\ & & \cdot & \cdot \\ & & & u_N \end{pmatrix} .$$

Then $u_1 \cdot u_2 \cdots u_N = \det(J_N - x)$ so that

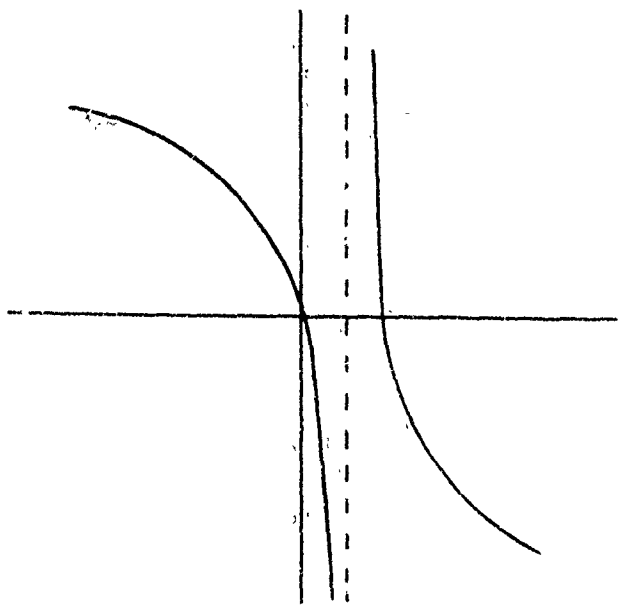
$$u_N = \frac{\det(J_N - x)}{\det(J_{N-1} - x)} .$$

Then $u_1 = a_1 - x$ has the graph



It is monotonic, and so is its computed value on any machine that has monotonic arithmetic.

Next, $u_2 = a_2 - x - \frac{b_1^2}{u_1}$

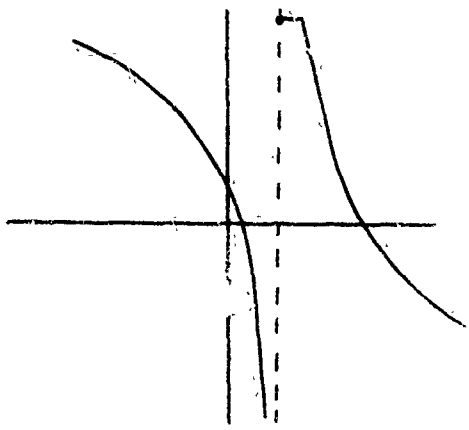


This function is monotonic except at its poles, where $u_1 = 0$. A similar statement, proved by induction, is also true of each of the other u's; they are decreasing except for jumps at poles where the previous u had a zero.

Now let us consider computed values:

$$U_I = (A_I - \frac{B_{I-1}^2}{u_{I-1}}) - X \quad .$$

Clearly this is monotonic if $A_I - \frac{B_{I-1}^2}{u_{I-1}}$ is monotonic decreasing. Considering the problem of round-off first, we can show by an induction that, if u_{I-1} is monotone decreasing, then $\frac{B_{I-1}^2}{u_{I-1}}$ is monotone decreasing, so $A_I - \frac{B_{I-1}^2}{u_{I-1}}$ is, except when $u_{I-1} = 0$. In this case of zero, we replace 0 by ETA, so we get an enormous jump. Then the graph of the computed u resembles

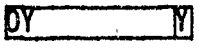


That is, we choose ETA so that no other quotient $\frac{B_{I-1}^2}{u_{I-1}}$ formed by representable numbers in the machine can be larger than B_{I-1}^2/ETA in magnitude. Clearly, then, ETA depends on the machine. On the 6400, for instance, we choose ETA to be the number smallest in magnitude but differing from zero, which has characteristic 0 and a non-zero integer part. The machine must operate in the mode which tolerates out-of-range operands, because the divider produces an ∞ with the correct sign. (The possibility that B_{I-1} is zero can be coped with in several ways discussed in the paper.) Consequently any other value of u_{I-1} will produce a quotient no larger than ∞

so monotonicity will certainly be preserved.

So monotonicity in the arithmetic is all it takes to guarantee the monotonicity of the u's in this algorithm. We can see that if x is increased then $\nu(x)$ cannot decrease. Suppose x is increased by one ulp. Then the u's may decrease a bit. If they decrease and preserve their signs, the count does not change. If they change their sign, the count might be affected. But the only way a u can go from a negative to a positive value (decreasing ν) is for the previous u to go from a positive to a non-positive value, increasing ν. The only time ν has a net change is when the last u goes from positive to negative, so that ν increases. This is the way to detect an eigenvalue.

Therefore we can go a long way with this algorithm if the machine satisfies the simple requirement of monotonicity! Yet even this simple requirement is not always assured. For several years the 360 long word multiplication was not monotonic, before the guard digit was added to the hardware. Then for certain positive X, H, and Y, X*Y > (X+H)*Y. If X has the significant hexadecimal digits FF...F, then its product with Y was Y minus one ulp. If X was increased to 1000...0, then the product formed would be



before postnormalization, and the failure to provide a guard digit lost the last hexadecimal digit of Y. The amount of Y lost could be as large as fifteen ulps.

Likewise CDC's RX* was not originally monotonic, or even commutative. Nowadays such defects are mostly limited to software floating point packages.

17. HOW MANY SIGNIFICANT FIGURES DO YOU NEED TO SOLVE A CUBIC EQUATION?

Theorems in numerical analysis are often of a negative sort and prove that certain calculations can't be performed. Yet correct theorems that seem to apply to certain problems often do not, as in the case of Viten'ko's theorem [10]. Very often the "impossible" calculation can be performed.

An example of a fruitful area for such theorems and surprising counter-examples is in the answers to questions such as, how accurate is the result of a computation if n significant figures are carried? If its error analysis does not provide realistic bounds, such a theorem may be misleading when we ask the complementary question: how many significant figures must be carried to achieve a desired pre-assigned accuracy?

In particular, we shall study the solution of a cubic equation to see what precision must be carried to get roots correct to single precision. In general, we can imagine solving for the roots by an explicit formula involving the coefficients or by some sort of iteration such as Newton's method.

Newton's Method $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ converges almost always to a root of a cubic equation. Can such a method get around rounding errors? Hardly. We must compute f, after all. Suppose our stopping criterion is $f(\xi) = 0$. We will find that many simple functions don't vanish for any value in our machine. Consider

$$F(X) = ((((((1.-X)+1.)-X)+1.)-X)+1.) \quad .$$

Certainly the function $f(x) = 4-3x$ has a root of $\frac{4}{3}$. When we substitute for X a number near $\frac{4}{3}$, we get approximately $-\frac{1}{3}$, $+\frac{2}{3}$, $-\frac{2}{3}$, $+\frac{1}{3}$, -1, and 0 for our partial results. Now recall that, on any non-ternary base machine in the Western world with floating point hardware, 1 and numbers

near $\frac{4}{3}$ are represented with the same characteristic, so that their subtraction occurs without error. The result near $-\frac{1}{3}$ has zeros inserted on the right when it is normalized. Therefore, since it was formed from a number near $\frac{4}{3}$, it can be added to 1 precisely. The digits shifted off and lost, or put in a guard digit or word, are always zeros and are of no consequence. The same argument applies to each of the six additions and subtractions. In each case a number formed from a 1 or a $\frac{4}{3}$ is added to another 1 or $\frac{4}{3}$, always precisely, so that $F(X)$ is always computed precisely near $\frac{4}{3}$. Therefore $F(X) = 0$ only when $X = \frac{4}{3}$. But no machine with a non-ternary base can represent $\frac{4}{3}$ precisely. Therefore $F(X) \neq 0$ on any such machine.

Therefore, when iterating we must wait for f to become negligible or for the sequence x_n to settle down. In the latter case settling occurs when $\frac{f}{f'}$ is small, and this may not mean that f is especially small. Indeed, rounding could cause the computed value of f to become zero at the wrong place.

To see how far wrong roots computed by any method could become, consider a cubic such as

$$f(x) = x^3 - \tilde{3}x^2 + \tilde{3}x - \tilde{1} \quad ,$$

where $\tilde{3}$ means a number near 3. Then the roots are $\tilde{1}$. Suppose the only error e is in the last multiplication so that

$$\frac{((x-\tilde{3})x+\tilde{3})x}{1+e} - \tilde{1} = 0 \quad .$$

Since $1+e$ won't fit in a word length, we round it to 1 so that we actually solve $f(x) \doteq e$. Suppose now that we are actually trying to solve the equation $(x-1)^3 = 0$ so that instead we solve $(x-1)^3 = e$, whence $x = 1+e^{1/3}$. If six figures are carried, $e^{1/3} \doteq 10^{-2}$. The root may only be good to

one third as many figures as were carried.

Similarly, if an explicit formula is used and part of the rounding error is applied to any of the coefficients, the perturbation in the roots could be the cube root of the perturbation to the coefficients. Hence triple precision seems to be required. We don't actually have a thoerem here, only a good argument.

Oddly enough, double precision will suffice. This is possible because computers do better than our model of arithmetic implies! There is evidently some hidden order to the arithmetic which we have not explicitly uncovered.

G.W. Stewart III concluded that there was no way to avoid triple precision in Mathematics of Computation 25, January 1971, pp. 135-139. To see why he came to this conclusion, we must examine the way in which the ill-condition of certain cubics is customarily cured.

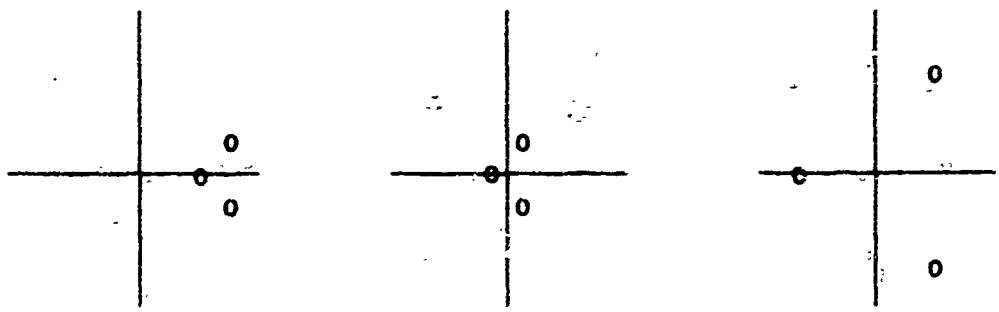
Usual Algorithm

The worst cases are when the cubic has nearly a triple root. When the roots are well spread out, nearly single precision results from a single precision calculation. When one root is nearly double, the perturbations are of order $e^{1/2}$ which can be handled using double precision.

Therefore we want to separate at least one root from the other two by a transformation. One way to do this is to translate the origin to the point that is the arithmetic mean of the roots. Suppose the cubic is $f(x) \doteq c(x-\xi)^3$. Then we want to find q such that

$$q(y) = f(\xi+y) \doteq cy^3 \quad .$$

Then, if our roots are still small, we can scale the problem by multiplying the coefficients by scale factors.



Problem Translation Scaling

Unfortunately, the usual method of translation causes rounding errors, which leave you as far from a correct solution as you were before!

Let us investigate what happens. The mean of the roots of a cubic $f(x) = a_0x^3 + a_1x^2 + a_2x + a_3$ is just $\xi = \frac{-a_1}{3a_0}$. The usual way of computing the coefficients of the new polynomial $g(y) = b_0y^3 + b_1y^2 + b_2x + b_3$, with $b_1 \doteq 0$, is as follows. Consider Horner's recurrence:

$$b_0 = a_0$$

$$b_{i+1} = \xi b_i + a_{i+1}, \quad i = 0,1,2 \quad .$$

We can write $f(x)$ in terms of the b's and ξ as follows:

$$\begin{aligned} f(x) &= \sum_{j=0}^{3} a_j x^{3-j} \\ &= \sum_{j=0}^{3} (b_j - \xi b_{j-1})(x^{3-j}) \qquad (b_{-1} = 0) \\ &= \sum_{j=0}^{3} b_j x^{3-j} - \xi \sum_{k=0}^{2} b_k x^{2-k} \\ &= b_3 + (x-\xi) \sum_{j=0}^{2} b_j x^{2-j} \quad . \end{aligned}$$

Then $b_3 = f(\xi)$. We think computationally of an arrangement such as the

following (note the re-definition of the b's).

$$b_0 = a_0$$

$$b_1' = \xi b_0 + a_1 \qquad b_1'' = \xi b_0 + b_1'$$

$$b_2' = \xi b_1' + a_2 \qquad b_2 = \xi b_1'' + b_2' \qquad b_1 = \xi b_0 + b_1''$$

$$b_3 = \xi b_2' + a_3$$

Then $f(x) = f(\xi+y) = b_0y^3 + b_1y^2 + b_2y + b_3$. We would use this recurrence because we expect the coefficients to be small near a triple root. After all, $b_3 = f(\xi)$, $b_2 = f'(\xi)$, $b_1 = \frac{1}{2}f''(\xi)$, and they would all be zero at a triple root.

Unfortunately rounding errors interfere in substantial numbers. Consider the equation $x^3 - \tilde{3}x^2 + \tilde{3}x - \tilde{1}$. Then $\xi \doteq 1$. Let us see what numbers are generated.

$$b_0 = 1$$

$$b_1' \doteq -2 \qquad b_1'' \doteq -1 \qquad b_1 \doteq 0$$

$$b_2' \doteq 1 \qquad b_2 \doteq 0$$

$$b_3 \doteq 1 - 1 = 0$$

We see that b_1, b_2, and b_3 are primarily composed of rounding errors revealed by cancellation! Our coefficients have been perturbed by rounding errors of order ϵ, so that we can expect the roots derived to have the usual $\epsilon^{1/3}$ uncertainty!

Stewart shows that the polynomial computed this way is not $f(y+\xi)$ but is instead $g(x)$ where

$$|f(x+\xi) - g(x)| \leq 6\epsilon|f|(|x| + |\xi|) \quad .$$

$|f|$ is the polynomial obtained by replacing all the coefficients b_i by their absolute value. Indeed, it is only realistic to suppose that the result of cancelling large computed numbers will only reveal their accumulated rounding errors. Hence, we must use triple precision with this algorithm to get single precision roots.

Kahan Algorithm

Much to our surprise, there is an algorithm, little different from this, that allows computation of singly-precise results using only double precision. Let

$$Q(x) = a_0x^3 + 3a_1x^2 + 3a_2x + a_3$$

$$Q(z+\mu) = b_0z^3 + 3b_1z^2 + 3b_2z + b_3$$

($z+\mu = x$, where μ is the origin shift). This rewriting is convenient, so that our new recursion is written

$$b_0 = a_0 \qquad b_1 = a_0\mu + a_1 \qquad b_2' = a_1\mu + a_2 \qquad b_3' = a_2\mu + a_3$$

$$b_2 = b_1\mu + b_2' \qquad b_3'' = b_2'\mu + b_3'$$

$$b_3 = b_2\mu + b_3''$$

Let us try this new algorithm on the previous example. Then $a_i \doteq 1$ and $\mu \doteq -1$. Thus

$$b_0 = 1 \qquad b_1 \doteq 0 \qquad b_2' \doteq 0 \qquad b_3' \doteq 0$$

$$b_2 \doteq 0 \qquad b_3'' \doteq 0$$

$$b_3 \doteq 0$$

Cancellation is done <u>first</u>, so there is no rounding error to reveal.

Then arithmetic is done on the cancelled results. All the products are done with near-zero operands except $a_i\mu$, which can be held in double precision.

When $a_i\mu$ is performed, the product may occupy as much as double precision. The single precision part will cancel out when $a_i\mu + a_2$ is computed, leaving either zero or a small number that was in the double precision part of the product. By carrying double precision throughout we can nearly get single precision results in the end. Actually a bit more than double precision is necessary. To avoid this, choose μ to be zero in as many bits as possible at the right, so that the effects of rounding products are postponed as late as possible.

The algorithm we will use for choosing μ is to look at the successive quotients $\mu_i = \frac{-a_i}{a_{i-1}}$. (Then

$$\mu_1 = \frac{\xi_1+\xi_2+\xi_3}{3}, \quad \doteq \xi$$

$$\mu_2 = \frac{(\xi_1\xi_2+\xi_1\xi_3+\xi_2\xi_3)}{(\xi_1+\xi_2+\xi_3)}, \quad \doteq \xi \text{ whenever all } \xi_i \doteq \xi.)$$

$$\mu_3 = \frac{3\xi_1\xi_2\xi_3}{\xi_1\xi_2+\xi_2\xi_3+\xi_3\xi_1}, \quad \doteq \xi$$

Now we find a number which matches as many of the leading digits of the μ's as possible.

$$\begin{array}{lcccccc}
 & & & k & & & \ell \\
\mu_1 & XX & a & b & c & d & \downarrow \\
\mu_2 & XX & e & f & g & h & \\
\mu_3 & XX & i & j & m & n & \\
\hline
 & & \boxed{\mu} & \boxed{\quad 0 \quad} & & &
\end{array}$$

Then μ matches each μ_i to k digits of the ℓ-digit word.

When we compute $b_1 = a_0\mu + a_1$, $a_0\mu$ will match $-a_1$ to k digits, so the result b_1 has at most about ℓ digits. Hence b_1, b_2', and b_3' will all fit in a single word. We can't be sure what happens next, but we have reduced the rounding error in the coefficients by 10^{-k}.

We have reduced rounding errors by losing significance! As the roots become closer together, the algorithm works better at shifting the origin with little error. A proof of this, assuming the roots are sufficiently close together, is given in Kahan's Notes for the Summer Institute at the University of Michigan, 1968. This proof is bad because it assumes something we don't know in advance. It's possible for the μ's to agree to k digits while the roots only agree to $\frac{1}{3}k$.

<u>Thesis topic</u>: Discover quickly a satisfactory and rigorous proof of this algorithm.

<u>Examples</u>

Let's consider a few examples. Let

$$Q(x) = 353x^3 - 984x^2 + 915x - 284$$

and we shall carry three figures in single precision. $\mu = .93$, and the usual Horner scheme yields

$\alpha_0 = 353$	$\alpha_1 = -984$	$\alpha_2 = 915$	$\alpha_3 = -284$
$\beta_0 = 353$			
$\beta_1' = -655.71$	$\beta_1'' = -327.42$	$\beta_1 = 000.87$	
$\beta_2' = 305.1897$	$\beta_2 = 000.6891$		
$\beta_3 = -000.173579$			

For the last subtraction, we needed nine digits, i.e. triple precision, of which three digits cancelled. Had we carried only six figures, instead of 9, we should have gotten $\beta_3 = -000.173$ or -000.174, with an error equivalent to perturbing the coefficient 284 by about 5×10^{-4}, or a relative perturbation of about 10^{-6}. We would run the risk of getting only two significant figures correct in the roots. In the second scheme

$$
\begin{array}{llll}
a_0 = 353 & a_1 = -328 & a_2 = 305 & a_3 = -284 \\
b_0 = 353 & b_1 = 000.29 & b_2' = -000.04 & b_3' = -000.35 \\
 & & b_2 = .2297 & b_3'' = -.3872 \\
 & & & b_3 = -.173579 \quad .
\end{array}
$$

Six digits (double precision) was enough to get the coefficients precisely. It would seem that the μ_i's agreed to two figures, but the roots agreed to but one, being 1 and $.8937677 \pm .0755768i$.

Perhaps a more typical example would be

$$Q(x) = 3x^3 - 813x^2 + 13449x - 2212111 \quad .$$

Then the zeros are 90.1150133, $90.4424934 \pm 1.6439023i$ -- agreement to one figure. This cubic is very sensitive. If we change a_3 to 2212110, the roots become 90, $90.5 \pm 1.6583124i$. A change of five parts in 10^7 changes the roots by one part in 10^3.

If $\mu = 90.3$, then $Q(z+\mu) = 3z^3 - .3z^2 + 8.01z + 1.5111$. Horner's scheme requires ten digits for precise coefficients, while the new scheme requires eight. The new scheme wasn't particularly designed for this type of cubic. In any case double precision can give you the translated cubic precisely.

As a final example, invert the order so that

$$Q(x) = 2212111x^3 - 13449x^2 + 813x - 3 .$$

Then $\mu = .0111$, and

$$Q(z+\mu) = 2212111z^3 + 214.2963z^2 + .09478893z + 10^{-6} \cdot .289041 .$$

The new scheme required 9 digits while Horner's required 13, to get the coefficients precisely.

We conclude that if double precision is used, the errors in the new scheme affect the roots far less in the critical cases, than the errors in Horner's recurrence. If the roots are <u>not</u> clustered, we don't need to translate the roots to the origin. Double precision suffices for the solution of the untranslated equation if the μ_i don't agree to any digits.

We have seen now that triple precision is not necessary for solution of the cubic, and double precision will suffice. We are tempted to ask if we can do without double precision. It is suspected, but not proved, that there is <u>never</u> any need for explicit multiple precision! But the general schemes that have been proposed are rather costly in time and space.

Suppose They Built a Strange New Machine?

Now here is an upsetting fact. An algorithm which looks very innocent depends in a crucial way upon factors or aspects of floating point arithmetic which appear to be present in all the machines that I'd thought of at the time. Yet I can imagine somebody building a machine or implementing his single precision in a way that would invalidate this program. How would you ever debug it? You could say it was rounding errors, but then why does it work on other machines that also presumably commit rounding errors?

The tricks I've been telling you about are important because we would

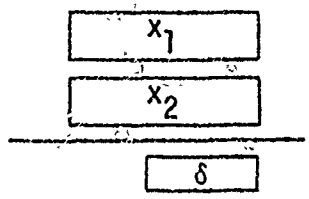
like to know how to design machines which are economical, easy to understand, and have a rich set of nice properties that would allow you to write reasonably efficient programs. Then your programs would run on machines that satisfied these few reasonable rules. It is important to find out what these rules are. Alas, nobody has been brave enough to write them down.†

Are There Any Machine Dependent Parts in the Code for the Cubic?

We have just seen that there is an alternative to Horner's method for translating the origin of the cubic equation problem, which only seems to require double precision. It absolutely requires cancellation for maximum effectiveness! We would like to know if there are any machine-dependent parts of the algorithm.

Remember that we are required to compare three numbers and extract as many leading digits as are equal in them. This may seem to be a machine dependent operation.

Suppose we wish to compare two decimal numbers x_1 and x_2, which we can suppose to be positive. Then let $\delta = |x_1 - x_2|$.



Suppose we can multiply by a power of the base (10). Then we want to

†We have rules which we think are reasonable, but nobody has built a machine like that, except for the BCC machine. If it ever gets straightened out it might be the first of a family of machines sufficiently decent in its hardware that you could imagine all sorts of other machines copying it, or copying it well enough that you could have machine independent code. Right now, the situation is anarchic.

Note: The Berkeley Computer Corporation folded and their machine was never completed, but some aspects of its arithmetic are discussed in [Appendix I].

find k such that

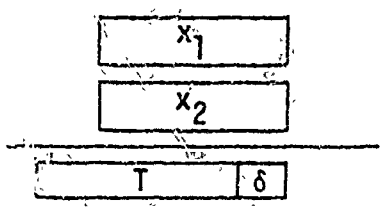
$$\frac{1}{10} \leq 10^k \delta < 1 \quad .$$

Then if we form

$$\frac{\text{Integer}(10^k \cdot x_1)}{10^k}$$

we get the leading digits. This could easily be programmed. The method is satisfactory but requires the base of the machine.

There is another method. Suppose we found another number T such that it formed a whole word to the left of δ:



Then $(T+x_1) - T$ would give the leading digits we seek. The non-agreeing digits of x_1 would fall off in the addition $T + x_1$, then the removal of T leaves the leading digits of x_1.

Since $\delta \doteq 1$ ulp of T, if we knew the rounding error level ε we could write $T \doteq \frac{\delta}{\varepsilon}$. Now we need to know the rounding error level instead of the base.

It is possible to write complicated machine independent coding to discover the base or ε. (M. Malcolm, "Algorithms to Reveal Properties of Floating Point Arithmetic," Stanford Report CS-71-211, 1971.) However, we can get ε roughly, to within a factor of two, with comparative ease, which is good enough for the cubic algorithm.

On machines with base 2, 4, 8, 16, 32,..., or 10 (this covers North American and West European machines)

H = 1.0/2.0 is always exact;

T = 2.0/3.0 is never exact and has an error less than 1 ulp.

In fact, T must be formed by chopping $\frac{2}{3}$ ulp or rounding in $\frac{1}{3}$ ulp. Now if we form 4H - 3T it should be about zero, plus the error in T, plus addition errors. If we compute in the form EPS = ABS(((((H-T)+H)-T)+H)-T)+H) we will see that no rounding error is committed due to addition or subtraction. H and T must have the same characteristic so H-T is exact and is about $-\frac{1}{6}$. This number was formed from H so it can be added to H. Similar arguments apply down the line so that we compute EPS to be |2 - 2 ± 1 or 2 ulps| so that EPS is 1 ulp on a rounding machine and 1 or 2 ulps on a chopping machine. Thus we have a simple procedure that is machine independent which we can use for the cubic algorithm.

Students' Report on Coding the Cubic Equation Algorithm

Our problem was the following: Given a cubic equation

$$a_0x^3 + 3a_1x^2 + 3a_2x + a_3 = 0$$

where the a_i are exact to single precision, solve for the three roots, exact to a certain small number of digits in the last place of the solution.

Separating the Roots

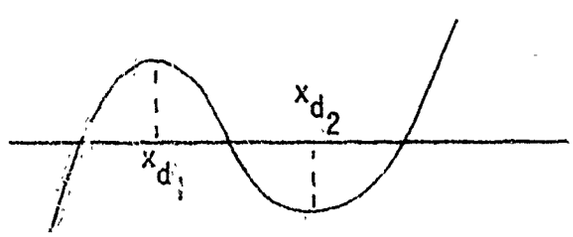
The program first tests to see if the roots are triple or very nearly so. If the roots are exactly triple, there is a relation which holds between the coefficients and the answer is obtained immediately. If the roots are not

exactly triple, we apply the transformation given previously to shift the origin and separate the roots. The shifting is done using double precision so that the coefficients of the new cubic equation will be correct to single precision. The shift factor, obtained from the digits that match in the ratios $-\frac{a_i}{a_{i-r}}$, $i = 1,2,3$, is never more than 48 bits in length. We arbitrarily decided that at least the leading 4 bits of the ratios should match for the roots to be considered close.

If necessary, the shifting can be done more than once. Our program applies the shifting until the roots are completely separated.

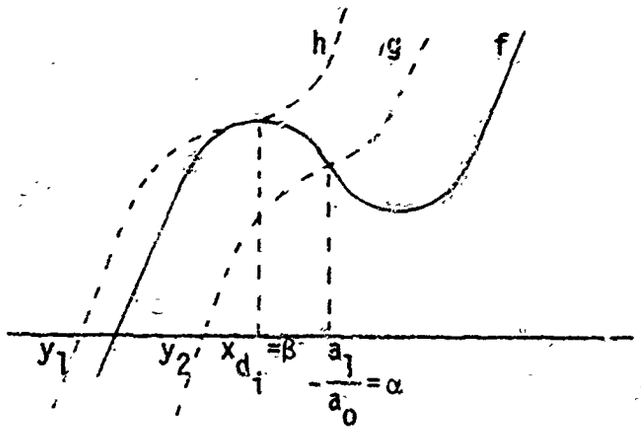
After Roots Are Separated

Once the roots are well separated, we compute x_{d_1} and x_{d_2}, the two roots of the derivative of the original equation. This is done to get an initial approximation to start a Newton-Raphson routine to find a real root (at least one root must be real in a cubic).



cubic with three real roots, showing the roots of the derivative

When x_{d_1} and x_{d_2} are real, we construct two functions, one through the point x_{d_i} where the function is largest in absolute value and the other through $-\frac{a_1}{a_0}$, the inflection point where the second derivative vanishes.

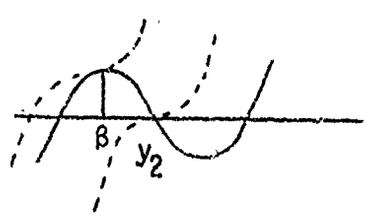


$$g(x) = f(\alpha) + a_0(x-\alpha)^3$$
$$h(x) = f(\beta) + a_0(x-\beta)^3$$

You are expanding f around these two points. Then, to the left of α, f is greater than g; to the left of β, h is greater than f. (In other cases, it may be to the right of α and β that these relationships hold). There must be a real root of f between the real roots of g and h. Those two roots, y_2 and y_1, are easy to find; they are each the cube root of a real number. You then take a linear combination of y_1 and y_2 as the initial value for starting the Newton-Raphson subroutine. There is a magic factor, obtained by looking at the case of three real roots,† which tells us which linear combination we should take. We get

---

† The factor m is obtained by figuring out what combination of y_1 and y_2 will, in this case, give us exactly the root we are looking for. For m to work in other cases, you only need show that x_0 is to the left of β.



$$m = \frac{\sqrt[3]{2}+1}{\sqrt{3}} - 1$$

$$x_0 = \frac{y_1 + my_2}{1+m}$$

This gives us an x_0 which is to the left of β and the Newton-Raphson method will converge to the desired root.

Question: What is the rationale for finding the initial approximation in this way? If the cubic has three real roots, starting Newton's method almost anywhere will lead you to a root, unless you get sent to infinity, or get into a loop oscillating between two points. However, in the last case, one rounding error is enough to destroy the loop and then you'll converge. Why go through such an elaborate procedure when almost any starting point will work?

Answer: If you are not careful, you run the risk of converging to the root in the middle and that can be fatal to finding the other roots.

Get Smallest Root First

Once the first root r_1 has been found by the Newton-Raphson method,† using double precision, the original cubic is divided by the factor $(x-r_1)$. But r_1 must be the smallest root in order to get the precision needed to solve the resulting quadratic. The reduction to the quadratic is accomplished by:

$$b_0 = a_0$$
$$b_1 = 3a_1 + r_1 a_0$$
$$b_2 = 3a_2 + r_1 b_1$$

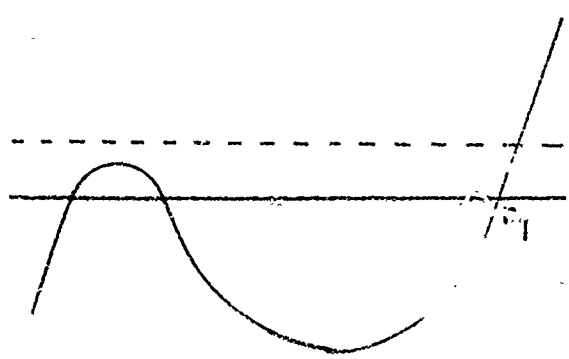
†The iteration continues as long as convergence is monotonic or until the new function value does not differ significantly from the old one.

Notice what happens if the root is very large (consider a_1 to be 1). $-3a_1$ is the sum of the roots; if r_1 is very large, $3a_1$ and r_1 are essentially equal and lots of cancellation (or almost complete cancellation) could occur. So that cancellation will not occur, you want r_1 to be the root smallest in magnitude.

Kahan: This argument is not valid because it depends upon some cancellation occurring that you say you don't want, whereas actually if the cancellation occurred properly you'd be very happy indeed. The issue is that when you compute

$$f(x) = (x-r_1)\underset{\substack{\uparrow \\ \text{quadratic}}}{Q(x)} + f(r_1)$$

$f(r_1)$ is supposed to vanish, but that may not happen. Suppose you made the error of accepting r_1 as a root, when it was only good to a few ulps of double precision admittedly. Then the Q you get, even if it is correct to double precision will be the quadratic factor, not of your polynomial f, but of your polynomial f modified by the subtraction of $f(r_1)$. Suppose r is a large root.



Where you have a big root you generally have a big derivative as well.

Then a small change in a big root, like a few ulps, can make a large absolute change r_1. And since the derivative is big, the change in $f(r_1)$ might also be big. Making that big change in $f(x)$ is like moving the horizontal axis (dotted line) somewhere, which tends to shatter the little root. That's why you don't want r_1 to be big.

If r_1 is the smallest root, the shift in the horizontal axis will be small and will hardly affect the biggest root.

You know you'd like the smallest root. But what if Newton's method is not so obliging and gives you the biggest root?

If We Find the Biggest Root First

If r_1 is the biggest root, we take the inverse polynomial by making the substitution $z = \frac{1}{x}$; that interchanges a_0 and a_3, a_1 and a_2, and the biggest root becomes the smallest. We solve this cubic and if necessary, reverse again.

Question: What if I move the origin to be very near that root with a large derivative? Then that argument about the derivative doesn't hold.

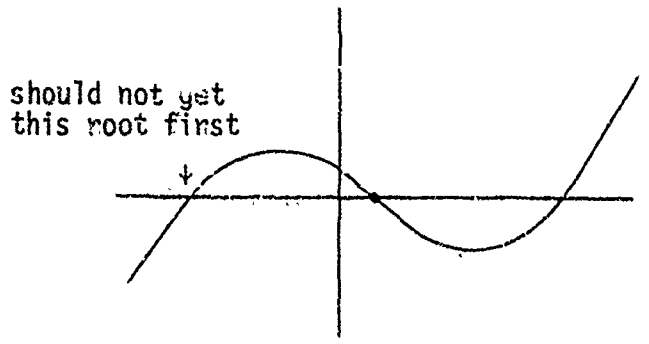
Answer: Yes, but a unit in the last place of a small root is a very small number so you have a small change. But then the other two roots, which are badly blighted by this change, will be blighted by almost any change in the coefficients of an ulp, so they are not well determined. The argument I gave is incomplete, but its essence is not that a certain type of cancellation does or does not occur, but rather that if you throw in all the other rounding errors, you'll discover that the division process will give you a quotient Q which is in fact the Q that corresponds to a slightly wrong polynomial, slightly wrong because of $f(r_1)$ and each coefficient having been altered by a little. Even if r_1 is the smallest root, the rounding

errors that will be most important will not be the errors associated with $f(r_1)$, but the errors associated with the perturbations in the coefficients, and if those perturbations cause the roots to fly around a lot, you just have to live with that. Of course, our perturbations are in double precision and will cause at worst a change in the roots of a few ulps in single precision. Since we have shifted the origin so that we don't have near triple roots, $e^{1/3}$ does not appear, but rather $e^{1/2}$ turns up. So if you do everything to double precision, you get single precision results essentially.

But let's get back to why it is bad for Newton's method to converge to the middle root.

Why Not Get the Middle Root First?

The middle root can also be a big root and then the same argument as before applies. The tiny root gets abnormally badly shifted by the rounding errors. And now you can't avoid the issue by inverting the cubic; you can't escape the rounding error problem. That's why you must not get the middle (in magnitude) root first.

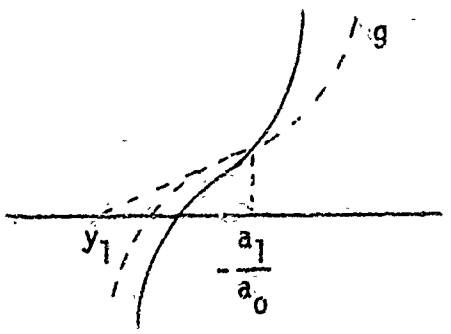


The strategy of the program will give the righthand root (which is the biggest) and then you invert the problem. If you had tried to divide out the factor for the lefthand root, you'd destroy either the largest or the

smallest root. When two roots are nearly equal and the third is very different, it doesn't matter which you get first, because you can always invert the polynomial. The problem arises only when the three roots are different in magnitude; you must not get the middle in magnitude first. The strategy is designed to avoid doing that.

Derivative Has Complex Roots

When the original cubic has no maximum or minimum but only inflection points, the derivative roots are complex. Then the picture looks like this:

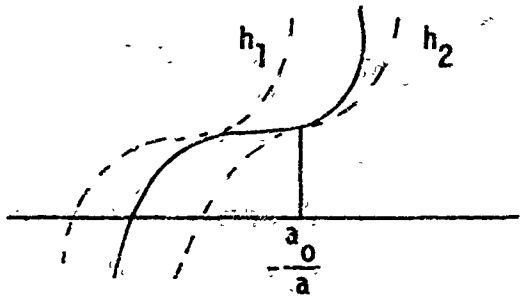


We still compute the function g through $-a_1/a_0$, which guarantees a point to the left (in this picture) of the root and a suitable starting value for the Newton-Raphson method. Another suggestion was to take the derivative at $-a_1/a_0$ and extend the line to the axis and see which point is closer to $-a_1/a_0$. With either approximation, the Newton-Raphson method converges.

The Derivative May Have Close Roots

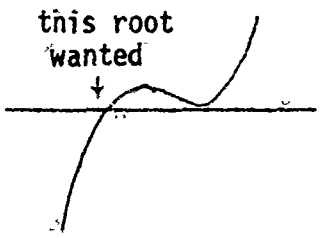
Problems arise when the quadratic has two real roots that are very close. The center of the graph becomes nearly horizontal. If we try to use the point for which the function is larger, we could make the wrong choice. However, the linear combination of points will give us a point to

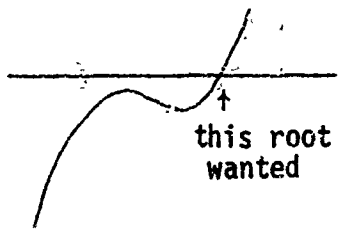
the left of $-a_1/a_0$, even if we made the wrong choice.



It doesn't matter which h we use in this case.

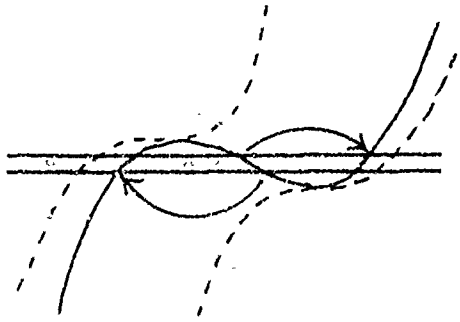
Kahan: The issue is to distinguish these two cases:





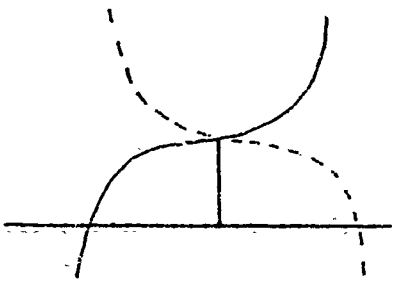
It is easy in the cases drawn here.

But what happens when the horizontal axis is so close to the point of inflection that you cannot tell which of the two lines is the axis, because of rounding errors.



If the lower line is the axis, you should go to the left; if the upper

line is the axis, you should go to the right. Now you need a formula such that the roundoff committed in evaluating the polynomial at the point of inflection will not do something bad to you. The original program compared the magnitude of f at the zeros of the derivative, but that has problems when the curve is nearly flat. The logic was such that given the graph below, you thought you had the dotted graph and you'd never find the zero.

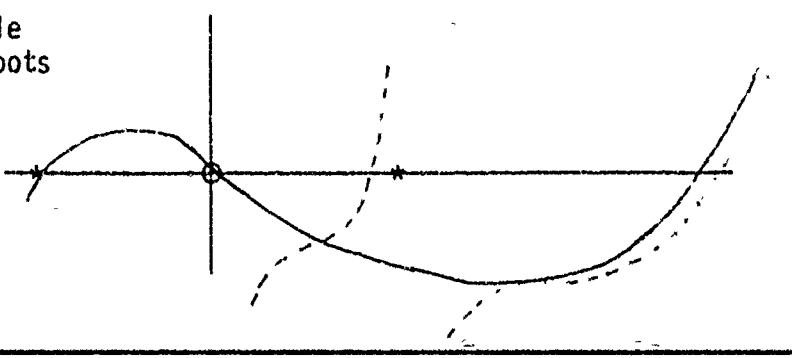


You need a formula such that, if you don't know if h should be to the right or left of the inflection, it will not matter which one you construct. You'll still get a satisfactory approximation to the biggest or the smallest root.

Question: I'd like to go back to the middle root problem. I can make any root middle by moving the origin.

Answer: There is a problem only when the three magnitudes are very different. If they are all close, it doesn't matter which root I get first. The algorithm appears to be independent of the origin but what it does is select a root to go to first, which has the property always, that if the three roots have very different magnitudes, you will not go to the one of middle magnitude.

* can be middle
(magnitude) roots



If the middle root is on the right, the algorithm won't take you there. So what if it is on the left. Then you'll go to the large root at the far right.

18. HOW SHOULD ONE SOLVE A NON-LINEAR EQUATION?

What Should We Mean By "Solve an Equation?"

How accurately can you solve an equation? I'm going to limit myself to a single equation in one unknown with a real variable, plus some further limitations later. The reason for imposing these restrictions is to have a reasonably definite object to study.

We want to solve: $f(x) = 0$

First, we should not take that imperative too seriously. Solving $f(x) = 0$ could very well be impossible for either or both of two reasons.

(1) When you compute $f(x)$, the value you compute will be contaminated by roundoff. It may be that even though you have the correct root representable precisely in the machine, an attempt to compute f, using reasonable arithmetic, will lead to rounding errors which necessarily produce a value of f that is not zero. It is conceivable that f as computed may never vanish, because the value is contaminated by roundoff. An example is a polynomial equation with reasonable integer coefficients, one of whose roots is an integer, but whose degree and coefficients are large enough that a rounding error necessarily occurs; once it occurs, it doesn't go away and the value is not zero where it should be.

(2) It is conceivable that you could compute f quite precisely, but will f vanish at any value of x available to you? An example of such a function is:

$$f(x) = (((x - 0.5) + x - 0.5) + x)$$

$$\text{or} \qquad f(x) = 3x - 1$$

This function has the property that, for any machine, in the neighborhood of the zero, no rounding error will occur when the function is evaluated. Why?

Of course 1/3 is not representable precisely on a binary or power of 2 or decimal base machine. But numbers very close to 1/3 are representable and they'll have the same characteristic as 1/2. So $x - 0.5$ will be done precisely. The difference will be $-1/6$; that may have a different characteristic from x, but when it is right shifted, no digits will be lost; so adding x doesn't give a rounding error; the result is now $-1/6$. Subtracting 1/2, even with a right shift loses no digits, so there is no error and the result is $-1/3$. Finally, adding $-1/3$ causes no rounding error and almost all digits will cancel. But because no rounding errors have been committed, you cannot get zero because you did not put in 1/3. Therefore, $f(x)$ never vanishes

Thus, if you insisted on solving $f(x) = 0$ explicitly, you could fail to do so even though you had committed no rounding errors. This example points out that there really are two reasons why equations are troublesome to solve.

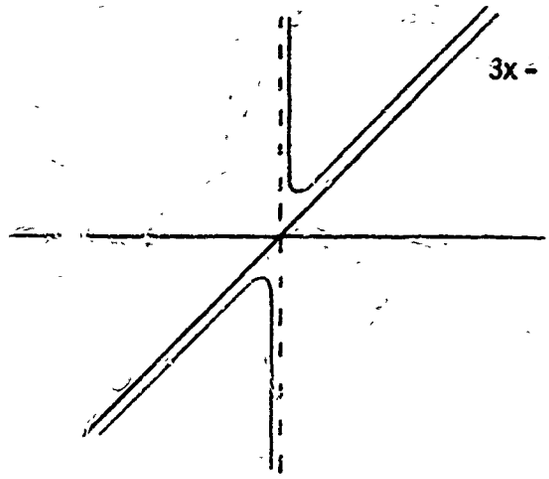
1) You cannot compute the function you'd like to have vanish exactly.

2) You may not have a place where the function is small enough to be called zero simply because your set of representable numbers may be too coarse.

It is possible to construct functions which, when computed in the machine with rounding error, will exactly match other functions that don't have the property you expect.

Consider the following two functions:

$$Z = 3x - 1 + \frac{e}{3x-1}$$



Denominator computed as $(((x-\frac{1}{2})+x-\frac{1}{2})+x)$ so it doesn't vanish. Z has a pole at 1/3, so it doesn't vanish anywhere.

Y and Z have the property that for all numbers in your machine, assuming underflow is set to zero without a message, their computed values are exactly the same, for suitably chosen e (≈ -150 on our machine).

Since Y and Z are indistinguishable, there must be certain things about zero finding that cannot be said with confidence. You have to be more circumspect in how you describe the problem. In effect, what we have to say is when the value of the function is small enough to be called zero. To do that you need to know more about the function than merely its computed value.

To know that we are not trying to solve an insolvable problem, we have to have a bound on rounding error. I want to compute $f(x)$ and I get $F(X)$. I need some tolerance e such that

$$e \geq |F(X) - f(x)|$$

I must know the uncertainty in the computed value. If I do not know

that uncertainty, I do not know if I'm trying to solve a reasonable or unreasonable equation.

The problem must be changed to read as: Solve $|f(x)| \leq \epsilon$. That might make more sense, if you have ϵ in advance. But the example of $3x-1$ showed that it is not enough to know a bound on rounding errors. Here the rounding error was zero and had I been asked to solve $|f(x)| \leq \epsilon$ with $\epsilon = 0$, I couldn't do it.

This defect will be repaired shortly. The point is that to solve an equation you have to state more than the subroutine that defines the function. This problem is not due to any particular programming language, but rather resides in the mind of people who want to use equation solvers. These people must be educated to realize that equation solvers that require only a function defining subroutine cannot be depended upon. Additional information in the nature of an error bound is necessary.

Consider a modification of the above problem, which will have a solution.

Suppose $|F(X) - f(x)| \leq \epsilon(X)$, where F and ϵ are known. A subroutine computes what is intended to be $f(x)$ to within a known tolerance, which may vary with X. Suppose also that I know that if $|x-x'| \leq 1$ ulp of x, then

$$|f(x) - f(x')| \leq \delta(x)$$

$\delta(x)$ is a bound on the variation of the function when you vary the argument by 1 ulp. So you know (1) how to compute the function approximately, (2) how approximate is that approximation and (3) how rapidly the function varies.

Question: You're not talking about the kind of approximation where you compute approximately some approximate value, are you? [See 13].

Answer: That has to be bound up in the ϵ. It can include rounding

errors and truncation errors (taking a finite part of an infinite series). Like for $\sin x$, you expect the result to be within a unit or two of the sine of some number which is almost what you put into the machine. ϵ must reflect all that error.

As we will see $\delta(x)$ is not as independent of $\epsilon(x)$ as it would appear. Not only are there errors in rounding the output; there are also errors in rounding the arguments which will appear in δ and also in ϵ. Normally δ need not be known in advance.

When Will a Solution Exist

I will show that if $f(x)$ vanishes anywhere, then necessarily F must become smaller than the sum of the two tolerances.

"Solve $|F(X)| \leq \epsilon(X) + \delta(X)$" has a solution, provided "solve $f(x) = 0$" has a solution. The main subroutine is F; you must also give the equation solver a subroutine that provides $\epsilon(X)$ (not too exactly computed) and another that provides $\delta(X)$ (although this one is not really needed). I didn't say I would solve $|F(X)| \leq \epsilon(X) + \delta(X)$, only that it has a solution which is quite a different thing.

The trouble with this theorem is that it is non-constructive and trivial. The proof sheds very little light upon the nature of the problem.

Proof. Say $f(x) = 0$ defines a value x, not necessarily representable. Let X be the closest representable number to x, so that

$$|X - x| \leq 1 \text{ ulp of } x \text{ or } X$$

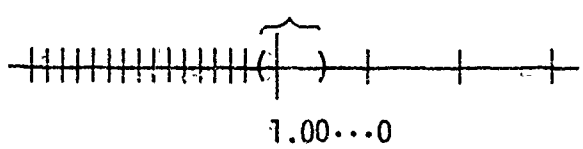
Therefore

$$\begin{aligned} |F(X) - 0\ (= f(x))| &\leq |F(X) - f(X)| + |f(X) - f(x)| \\ &\leq \quad \epsilon(X) \quad + \quad \delta(X) \text{ by hypothesis} \end{aligned}$$

The proof is trivial; a solution exists, but all the proof does is assure us that the requirement "solve $|F(X)| \leq \epsilon(X) + \delta(X)$" is not yet known to be impossible.

Question: I'm still confused by one unit in the last place. It seems you'd find a place where the exponent changes so that the nearest representable number ...

Answer: The nearest representable number differs from the given number by less than 1 ulp of that representable number, even if it is $1.00\cdots0$. It may be a good deal less than 1 ulp of the representable number. That's why I have $\delta(X)$, not $\delta(x)$.



Lines are representable numbers.

What is the number closest to 1? The difference is less than 1 ulp of 1.0, which is the large gap to the right of it. I could have said 1/2 ulp and gotten essentially the same result.

The problem is how do you go about computing $\delta(X)$ and $\epsilon(X)$? Normally, $\delta(X) < \epsilon(X)$, so ϵ is a bound on δ and not too large a bound. (I say normally because of the example $3x-1$ where $\epsilon(x) = 0$.) $\epsilon(X)$ comes from two sources:

(1) You used an expression that is not exactly the function you want.

(2) Roundoff alone (I'll consider just this one).

I'll get a bound for $\epsilon(X)$, considering roundoff only. Had there been truncation errors, ϵ would be bigger and the result would be even more true (if one thing can be more true than another).

Rounding Error Analysis

$$F(X) = \mathcal{E}(X,X,\ldots,X) = \mathcal{E}(X_1,X_2,\ldots,X_n)$$

Every operand that appears in calculating $F(X)$ is named separately. For example

$$F(X) = \frac{1+X}{1-X} \qquad \mathcal{E}(X,X) = \frac{1+X_1}{1+X_2} = \mathcal{E}(X_1,X_2)$$

Subscript each appearance of each operand so that you could think of them as independent variables. The function you compute is what you get when all X_i's have the same value.

What would a rounding error bound look like? There will be many rounding errors. Among them will be the ones attached to our attempts to use the operands X_i. For example, on the 6400, when we add anything to X, something else is computed.

$$\text{"}X \oplus Y\text{"} \text{ becomes } X(1+\xi) + Y(1+\eta)$$

We only know a bound on ξ, that $|\xi|$ is at most 1 ulp; $(1+\eta)$ only makes the error bigger. When you compute rounding error bounds in the usual way, every use of X introduces a rounding error which may be attached to that letter X as a perturbation of at most a unit in the last place. Other stuff gives other perturbations, which tend to make the error even bigger.

Let us consider how big is the contribution of those rounding errors that are attached to a letter X, every time it appears. The total error will certainly be even bigger than that.

The easiest way to discuss this is to differentiate (even though that is not necessary). What I compute in place of $\mathcal{E}(X,X,\ldots,X)$ is at least as bad as $\mathcal{E}(X(1+\xi_1),X(1+\xi_2),\ldots,X(1+\xi_n))$. In each case, $|\xi_j X| < 1$ ulp of X or so.

How Does & Vary

$$|\mathcal{E}(\{X+\xi_j X\}) - \mathcal{E}(X,\ldots,X)| \lesssim \sum_j \left|\frac{\partial \mathcal{E}}{\partial X_j}\right| |\xi_j X_j|$$

I'm using the notion that each one of the X's is an independent variable and I can differentiate $\mathcal{E}$ with respect to that independent variable. I now have some notion of how the expression can be altered by rounding errors. The bound is in some respects realistic; it is conceivable that all the rounding errors ξ_j could have just the correct sign to match with the derivatives and ξ_j could be as large as a rounding error ever is but realistic to the extent that only if there were an extremely large number of rounding errors involved would we believe that you could not find an argument for which all the rounding errors would be about as bad as they could be.

The contribution due to roundoff is at least as bad as $\sum_j \left|\frac{\partial \mathcal{E}}{\partial X_j}\right| |\xi_j X_j|$, because roundoff will include some things we haven't taken into account (the $(1+\eta)$ and truncation error). We can also write this as

$$\sum_j \left|\frac{\partial \mathcal{E}}{\partial X_j}\right| |\xi_j X_j| \le \sum_j \left|\frac{\partial \mathcal{E}}{\partial X_j}\right| * (1 \text{ ulp of } X) \le \underset{\substack{\uparrow \\ \text{roundoff error}}}{\epsilon(X)}$$

Now let us consider what $\delta(X)$ has to be like; δ is a bound on the variation in the function caused by altering X by 1 ulp.

$$|f(X) - f'(X)| \simeq \left|\frac{df}{dX}\right| |X-X'| \lesssim \left|\frac{df}{dX}\right| * (1 \text{ ulp of } X) \doteqdot \delta(X)$$

That's the best bound we can hope to get for $\delta(X)$, so that's what we expect to get.†

† δ may be a bit bigger; I should look at the maximum value taken by the derivative on the interval [X,X'], but being too rigorous will just obscure the issue.

Compare the expressions for $\delta(X)$ and $\epsilon(X)$. To do so, observe that

$$\left|\frac{df}{dX}\right| = \left|\frac{d}{dX}\mathcal{E}(X,\ldots,X)\right| = \left|\sum_j \frac{\partial}{\partial X_j}\mathcal{E}(X,X,\ldots,X)\right| \le \sum_j \left|\frac{\partial \mathcal{E}}{\partial X_j}\right|$$

That last sum appeared in the expression for $\epsilon(X)$. So you see why I claim that normally $\delta(X) < \epsilon(X)$.

There are many gaps in the reasoning, aside from the fact that I've approximated in many places, but they are approximations that can be patched up. What I've really done that's unforgiveable is to assume that every appearance of X is going to have its own independent rounding error. That is visibly untrue, because we had an example where each appearance of X had no rounding error at all. I've also neglected to consider those machines in which roundoff behaves in a somewhat better way (where, when you sum, you round the sum and not the operands). It is an exercise to verify that even in that case, normally you expect $\delta(X) \le \epsilon(X)$. I say normally, meaning that when I add something to X, that something has been rounded, so that the rounding error that occurs can be attached to X instead when you look at the value of the whole sum; you allow the error to migrate to X. So this argument is reasonably general in spirit, but incapable of being proved precisely in all cases, since we have a counterexample.

This nontheorem allows us to simplify the specifications on a equation solving subroutine to read:

"Solve $|F(X)| \le 2\epsilon(X)$"

An exercise would be to consider a polynomial, evaluated in the usual way by Horner's recurrence

$$f(X) = a_0x^N + a_1x^{N-1} + \cdots + a_{n-1}x + a_n \quad \text{the function}$$

$$\mathcal{E}(X) = (\cdots((a_0X_1 + a_1)X_2 + a_2)X_3 + \cdots) + a_n \quad \text{the expression}$$

Each of the X's is multiplied and each multiplication generates an error that can be attached to the X. So e(X), constructed in any way you like, will necessarily have the property that it also bounds the variation in the function caused by altering X by 1 ulp. You really have a bound for the derivative.

What Methods To Use To Solve Equations

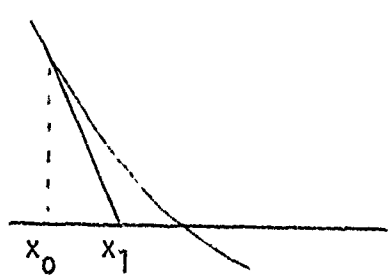
We now have some idea of the sorts of equations we could hope to solve. Now we need to consider what type of method to use to accomplish that solution.

The presentation has not been rigorous, but was intended to show the nature of things that could be proved rigorously. Only in exceptional cases can you hope to solve equations exactly in any sense.

When you start to look for the roots of an equation, a very interesting thing happens. Normally we say: Try some algorithm; if it doesn't work, try something else. That isn't much help. But for any algorithm that doesn't have an ironclad and necessarily trivial guarantee, you can expect to find counterexamples for which the algorithm will fail.

Newton's Method

An example is Newton's method. If you ever get close enough to a root of $f(x) = 0$, convergence is necessarily rapid.

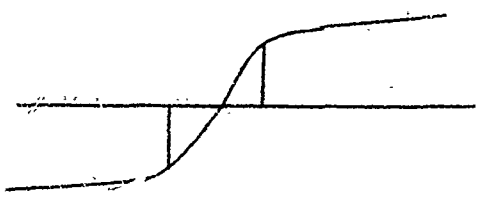


$$x_1 = x_0 - f(x_0)/f'(x_0)$$

Even convergence to a multiple zero is not unduly slow, provided you measure it the right way.

Unfortunately, the theory for Newton's method is of a local character. If you get close enough, then something will happen. The close enough means you can approximate your curve, to within a difference that doesn't matter, by a straight line. That's not generally what you have in mind when you start the problem. You shouldn't be surprised that there are many examples for which Newton's method doesn't work.

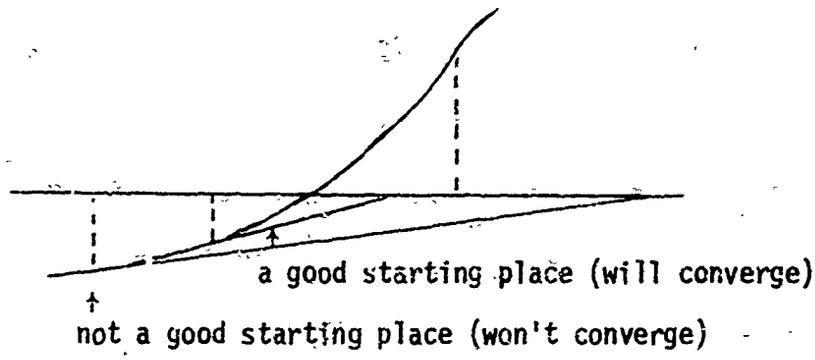
Suppose your function looks like this (like arctan):



If you start close enough, the method will work. But if you start outside the dividing lines, you'll go off to infinity and it won't take you long to get there.

When Will Newton's Method Work

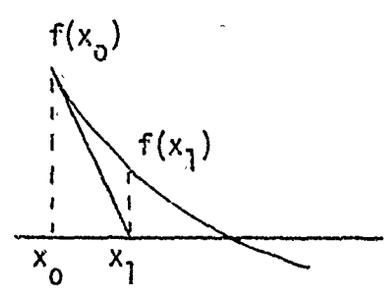
You'd like some sort of theory that tells you that if you use Newton's method in this case, it will always work. That takes some fairly strong global statements about your function, such as if the function is convex in some neighborhood of a root, then anywhere in that neighborhood, you can expect Newton's method to work as long as you don't get thrown out of that neighborhood by the first iteration.



A more precise statement (by Fourier) would be that if a function is convex in a certain interval at one end of which there is a root and certain sign conditions are met and if you start in that interval, you stay in it and convergence is monotonic.

However, a condition of this type is not entirely satisfactory, but it is applicable in many cases. The situation is complicated by our inability to recognize when Newton's method is convergent.

What many people do with methods like Newton's is to observe that when things are working, the value of the function decreases with every iteration. From the picture, $f(x_0) > f(x_1)$.

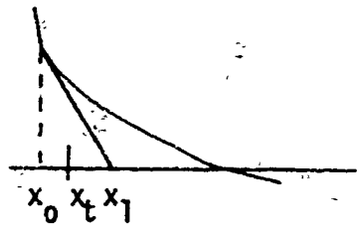


The direction that Newton's method tells you to go is in a sense a downward direction for the magnitude of the function.

Modified Newton Method

So modify Newton's method so that instead of moving a distance $f(x_0)/f'(x_0)$, you move a fraction t of that distance. Say

$$x_t = x_0 - tf(x_0)/f'(x_0), \quad 0 \le t < 1$$



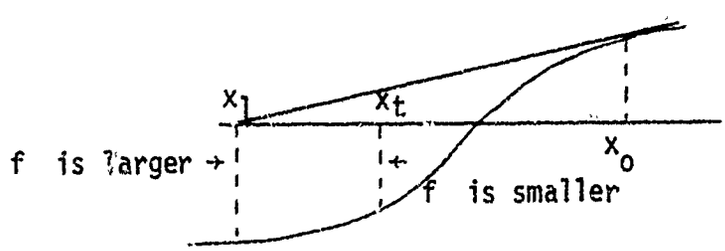
x_t moves in the direction Newton's method points, but not so far

How does f change?

$$\frac{d}{dt}f(x_t)\Big|_{t=0} = f'(x_t)\left(-\frac{f(x_0)}{f'(x_0)}\right)\Big|_{t=0} = -f(x_0)$$

The derivative, with respect to motion from x_0 to x_1, of f has a sign opposite to that of f. At least initially, the magnitude of the function declines, in the direction that Newton's method takes you.†

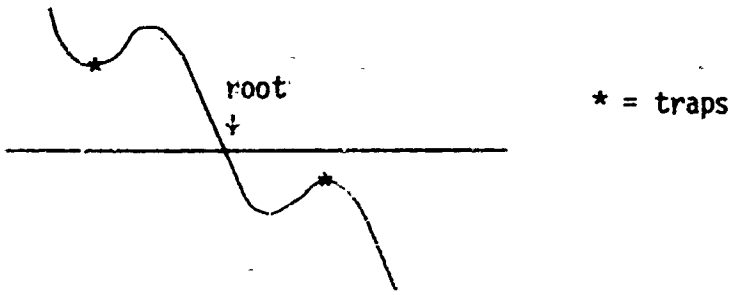
So people have attempted to guarantee, by the selection of t, that the value of f will always decline. That means $t = 1$ may not be such a good choice; t is some fraction that makes $|f|$ decrease. That will greatly improve convergence in the following case:



If you repeated this process you'd hope eventually for something good. There is a difficulty in that you are seeking a place where $|f|$ is minimal and it might not be a root. So usually appended to this is a method to see if

†You can apply Newton's method in the complex plane with similar results, or in the multidimensional case ($f'(x)$ becomes a Jacobian matrix and $1/f'$ becomes $(f')^{-1}$.)

you've reached a local minimum in magnitude.



The root is hidden by little traps. If the guy is a bit unlucky, he might end up in one of the traps and never find the root. He'd have to use something that is not already in the algorithm to discover his plight.

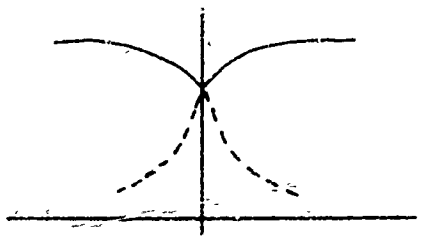
But normally, you do have a way of detecting this situation and then you do something else.

Question: Why not just take constant steps if you're going to scale down Newton's steps anyway?

Answer: In principle, by taking constant steps of 1 ulp you could exhaust all the arguments and find the solution if it existed. But that's not fast. You take Newton's step and hopefully that value is so much closer to the root that you'd verify this fact by noticing an enormous decrease in $|f|$. You're confirmed in that choice and do another Newton step. If you don't see the enormous decrease in $|f|$, then and only then do you use a different strategy. You put in t and cut the step in half, quarter, etc. I'm not recommending this method, but just indicating a rationale people might use and the way these things go wrong.

There are certain cases in which it is known that if you hit a minimum of the magnitude, there is an obviously right thing to do. These are cases when you're dealing with analytic functions in the complex plane. The

Minimum Modulus Theorem says that the only way for an analytic function's modulus to be minimum is for it to be zero (also called d'Alembert's principle). Therefore people often feel they have here a guaranteed method, if only they jump into the complex plane. Unfortunately, this doesn't always work out. That is because, although it is true that a minimum of the magnitude can only be a zero for an analytic function,† if the function has singularities, a minimum of the magnitude could easily be a singularity.



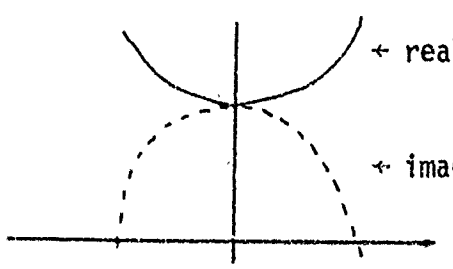
complex plane
$1 + Z^{1/2}$

$1 + Z^{1/2}$ has a local minimum in magnitude at $Z = 0$, but that is not a zero of the function. The graph has a break. In this particular case, if you turned yourself and plotted orthogonally, you'd get the dotted line and discover you were at a saddle point and now were at a maximum of the magnitude. This difficulty is quite typical.

Programs which are based on d'Alembert's principle generally hang up in one or both of two ways.

(1) They find a minimum in the direction Newton's method tells them to take.

$f(Z) = Z^2 + 1$



← real axis plot

← imaginary axis plot

†An analytic function is one which in the interior of a neighborhood has no singularities; in that neighborhood, d'Alembert's principle holds.

I'm willing to use complex values for Z, but begin by using a real value. Newton's method tells me to go along the real axis and no matter how I choose t I never get into the complex plane and I only find the local minimum. If you looked at the problem along the imaginary axis, the graph is rather different. But you can't get onto the i-axis using Newton's method.

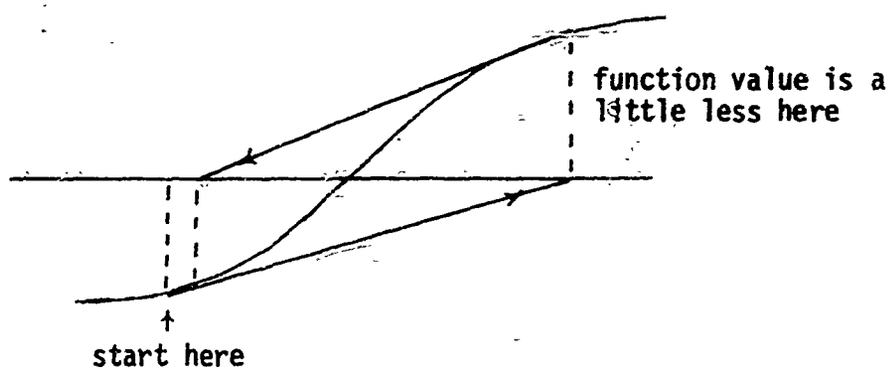
(2) So people are obliged to discover that they are at a minimum of the magnitude of an analytic function with respect to variation along a line which is necessarily a saddle point. Then they must turn the problem through an appropriate angle which depends on how many derivatives vanish.† You either know all the derivatives or are willing to make a large number of guesses.

But all of these algorithms have their own hangup. Their hangup is in their inability to recognize when they are nearing a minimum of the magnitude. You'd recognize that you were nearing a minimum by noticing that successive computed values of $|f|$ appear to no longer be decreasing sensibly; they've practically stopped decreasing. You'd identify the minimum because $|f|$ stopped decreasing at all or decrease only but a couple ulps. Unfortunately, it takes a long time to identify this fact, because convergence to the minimum of $|f|$ is generally very slow and there doesn't appear to be a decent way of speeding it up. You can even construct functions that don't have a minimum but somehow the algorithm does ugly things to you.

---

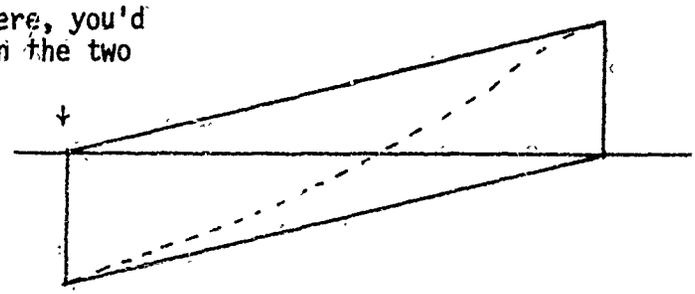
†If $f' = 0$ and $f'' \neq 0$, turn through 90°. If $f' = 0$ and $f'' = 0$ and $f''' \neq 0$, turn through 60° or 120°. If $f' = f'' = f''' = 0$, $f'''' \neq 0$, turn through 45°.

Here's an example specifically to refute algorithms based on d'Alembert's principle.



5th degree polynomial

If you started here, you'd oscillate between the two points.



If you were oscillating, you'd catch yourself if you were testing for a decrease in $|f|$. But say you start a little bit outside that box and get to a point also outside the box but a little closer. You end up traversing a path outside the parallelogram, getting closer all the time.

Examples like this can be constructed so that although the sequence of values $|f|$ decrease, they decrease arbitrarily slowly. The decrease at step n of $|f|$ could be $\frac{1}{n^2}$; you can easily figure out how big n would have to be to be decreasing $|f|$ by only a few ulps; perhaps at that point you'd be willing to give up that particular iteration.

Question: Wouldn't you notice something fishy in that your X values

are alternately positive and negative?

Answer: Did you notice that? You'd have to put in logic to notice that and you'd have to be sure that the logic wouldn't be tricked when the iteration is supposed to be like that. In this same example, if you get close enough the method is cubically convergent but the X's alternate in sign.

I'm going through all this to show you the lengths you must go to have a program that you can guarantee. It is not possible to write a program that you can guarantee for arbitrary subroutines defining f. It appears possible that in principle no matter what logic you use and what constraints you put on the functions (like demanding that they be continuous and really have roots), if your program accepts arbitrary functions, someone could look at your logic and construct an example to confound your method.

More Reasonable Claims for Equation Solvers

We must settle for a more modest type of claim which severely restricts the classes of functions for which the zero-finders will work. For example, some programs only take polynomials; even in this case no one has proved that, including all errors, his program will work for all polynomials. The only programs for which people have given even approximate proofs in the literature are those programs which are known to be exceedingly slow. For example, there's a method due to Lehmer that involves drawing circles and in the absence of rounding error tells you which circle contains a zero. If a circle contains a zero, you subdivide it into smaller overlapping circles and look again. This is obviously slowly convergent.

There are other algorithms which say that if you do something long enough, then an event will occur after which convergence will be fast. In principle you can show that you should not have to work very long to have

worked long enough. But the proofs cannot say how long is long enough.

There is a method by LaGuerre that is cubicly convergent if you are close to a zero. Programs using this method on the 7094 will accept polynomials up to degree 80, although it has been modified for polynomials of degree 2500. This may sound like a great accomplishment, but remember that such a polynomial has 2500 zeros which means they are almost everywhere.

The only trouble with LaGuerre's method is that it cannot guarantee to give a starting value (for the fast convergence) in the time you are willing to wait.

Then there are programs that use intimately everything you know about the function whose zero you seek. That of course includes the error bound because that's the only way you know when to quit looking.

Can Binary Chop Be Bettered

As an exercise consider finding a zero of a function known to be continuous and at the ends of a given interval the function has opposite signs. Is it possible to write a foolproof program that is faster than the obvious binary chop algorithm? To within certain limits, it is possible to construct a method that converges superlinearly. Once you get close enough, the number of correct digits is multiplied by some constant bigger than 1 at each iteration. The function does have to be smooth, but most functions writeable in FORTRAN are sufficiently smooth.

If you want to find where the function vanishes, you need more than the sign of the function. You also need to know an error bound. If you want only to know where the function changes sign as computed, you don't need an error bound but only need the sign digit correct. But then, binary chop is the best that you can guarantee.

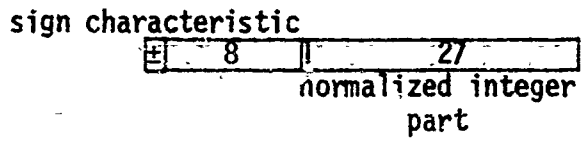
For an interesting algorithm of this type, see that of T.J. Dekker in <u>Proceedings of the Symposium on Constructive Aspects of the Fundamental Theorem of Algebra</u>.

19. CONSTRUCTION AND ERROR ANALYSIS OF A SQUARE ROOT ROUTINE

Now I'll give you a successful error analysis. It is based on a very intimate appreciation of the hardware of the machine. You have to choose a simple algorithm -- I have chosen a square root. This analysis must be done for elementary functions.†

I'm going to show you how to analyze a square root, completely. Every detail will be covered.

The 7090 and 7094 are signed magnitude machines:

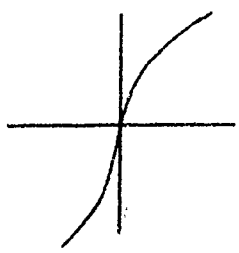


Specification for the SQRT routine:

i) $SQRT(X) \doteq \sqrt{X}$, $X \geq 0$

ii) $SQRT(X) \doteq -\sqrt{-X}$, $X < -0$††

and an error trace and message "SQRT(-X) = -SQRT(X)."



graph of SQRT(X), a nice continuous graph

---

† All the elementary functions for IBM 360/50, in single and double precision, have been analyzed in this way to the extent that number theory wasn't needed. The man who wrote them has done this and is able to say something like the error is no more than 15 units in the last place. Then the machine is tested on thousands of operands to see if his predictions are justified.

††The response to taking a square root of a negative number is not at all obvious. It's not obvious that we should be kicked off the machine. See [6].

Specifications on the ERROR

i) Error cannot exceed .50000163 ulp's (recall 27 bits is roughly 8 decimal digits, so the error bound is given to 8 digits).

ii) Among the 2^{34} essentially different positive floating point numbers (2^{27} different operands -- 2^{26} from the significant digits, 2^1 for whether the exponent is odd or even), only 29×2^7 produce incorrectly rounded square roots (neglecting powers of 4 that is only 29 different operands). By this I mean that for only that many operands will the value written out by SQRT be other than what you would have gotten by taking the square root exactly and then rounding it correctly to 27 bits.

The error bound is obtained by exhibiting those 29; for those correctly rounded, the error is 1/2 in the last place; for the others, the bound tells you how much worse the error is.

One way to tell how bad a subroutine is, is to enumerate all the errors. But you could tell how long that would take, even on the 7094; that was not what was done. You have to do an error analysis sufficiently accurate so that all the places where the errors are likely to be big are exhibited and in those regions you enumerate all the arguments.

Question: When you are checking these routines for the largest errors, suppose you are checking your double precision version?

Answer: That's not the way it's done. The double precision routine could also be wrong. There is actually a number theoretic way, which is quite precise.

Question: Would you go through the argument of what the 2^{34} numbers are, and in particular, do you accept unnormalized numbers?

Answer: No, we don't allow unnormalized numbers. There are 27 bits,

of which the leading bit must be a 1, so there are 2^{26} different operands. Then you can have 2^8 different characteristics, for a total of 2^{34} different numbers. But there are only 2^{26} times 2^1 essentially different operands, or 2^{27}. Only 29 of those give incorrect rounding.

Question: Is it part of the specifications that the routine only produces correct results for normalized operands?

Answer: Yes, all the subroutines on the 7090 are set up that way. Everything is assumed normalized.

Specifications to be Matched

Let's consider now some of the more valuable and interesting parts of the specifications.

i) If $X \geq Y \geq 0$, then SQRT(X) $\geq$ SQRT(Y)
(preserves monotonicity)

ii) SQRT(X**2) = SQRT(RND(X*X)) = ABS(X)
(exact for all X for which X^2 doesn't overflow)

Number ii) seems reasonable. But say in a fit of overambition, I tried to match the following:

SQRT(X)**2 = X .

It is not possible to do this for all x. Why not? It is true for all X which are perfect squares. (That case is insured by ii) above anyway.)

Question: Suppose on the 6400 that X and Y are sufficiently close that their square roots differ by 1 in the last place, so that when you subtract the 1, what is left is in the double precision part of the register.

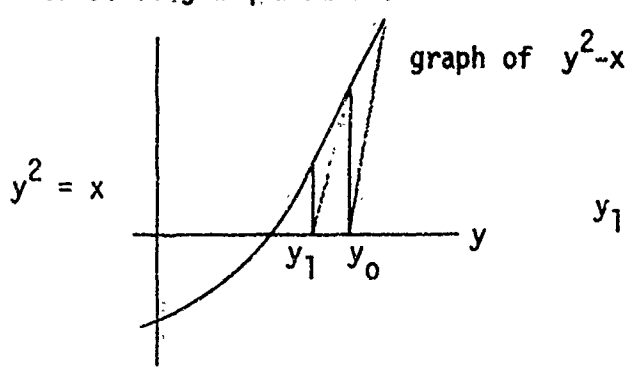
Answer: That is a problem that will have to be looked at by the people who do that project. But I'm talking about a 7090, and on it if two numbers are different their difference is nonzero, unless it underflows, and then you

get a message.

There are questions of how long the program should be, but that won't concern us except that it should not be appreciably longer than other programs. Then there are questions of what alternatives should be used. A properly documented program should say how long it takes, how much storage is required, what alternatives are there, are there any systems side effects. For example, in this SQRT, it is possible to take the square root of a number that has temporarily overflowed into the P and Q bits, e.g., for CABS. Another part of the documentation is the method used in the program.

Heron's Rule

The method is based on what used to be known as Heron's rule, now known as Newton's method for solving a quadratic.



$$y_1 = \frac{1}{2}\left(y_0 + \frac{x}{y_0}\right)$$

From the picture, this clearly converges. It is important for us to know how fast it converges. Unless it converges quickly, it is not a good method to use.

Convergence in this case is quadratic. If I can manage to get y_0 to match the square root of x to a reasonable number of digits, each iteration will about double the number of correct digits.

The easiest way to show this, without resorting to Taylor series and such is to pretend:

$y_0 = \sqrt{x}\left(\frac{1+\delta_0}{1-\delta_0}\right)$ relative error is $\approx 2\delta_0$ for small δ_0

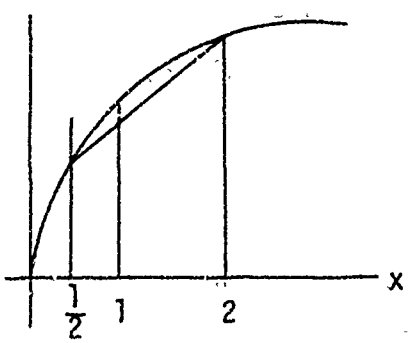
$y_1 = \frac{1}{2}\sqrt{x}\left(\frac{1+\delta_0}{1-\delta_0} + \frac{1-\delta_0}{1+\delta_0}\right) = \sqrt{x}\left(\frac{1+\delta_0^2}{1-\delta_0^2}\right) = \sqrt{x}\left(\frac{1+\delta_1}{1-\delta_1}\right)$ so $\delta_1 = \delta_0^2$ (quadratic convergence)

How do you begin? What is your first approximation for y_0?

<u>To Get the Approximation</u>

The idea is to choose a simple function, one that is extremely easy to compute and use it to approximate the graph of the square root.

Say you wanted to use a linear function. But if you do that over a large range, something will go wrong. So you restrict the range of your approximation to numbers within a factor of 4. All numbers will fit into this range by appropriate multiplication by powers of 4.



approximate in this range of x

Question: Wouldn't it be the range 0 to 2, not $\frac{1}{2}$ to 2?

Answer: No. Remember, zero is not a normal, floating point number. SQRT(0) = 0; it is too easy. And the square root of all other numbers can be obtained by taking off all but the last digit of the characteristic, and that puts them into the range $\frac{1}{2}$ to 2, without any rounding error.

Once the number is in this range, you can start to talk about simple, say linear, approximations. What is the best linear approximation? It turns out, however, that the best one is not the right thing to use, necessarily; it depends on the machine. On some machines, multiplying is expensive,

unless you multiply by a power of two. So things like the following are done:

$$X = 2^{(2I-J)} \cdot F \qquad 0.5 \leq F \leq 1.0$$

$$J = 0 \text{ or } 1$$

$$\sqrt{X} = 2^{I} \cdot \sqrt{2^{-J}F} \qquad \text{the } \sqrt{\ } \text{ is in the range } \frac{1}{4} \text{ to } 1$$

$$y_0 = 2^{I}(F/2 - J/4 + c)$$

I tried all possible programs of a given length, and this one turned out the best.

Question: Did you try table lookup?

Answer: Yes. One of the best programs on the 7090 was a rather elaborate table lookup, but on the 7094, my scheme was faster.

Question: I have a question about the function you chose to start the SQRT. You said there were other choices. What were they?

Answer: The idea is to consider all possible programs, no longer than one program that already worked, that could compute a square root. There are certain programs so implausible that you can rule them out immediately.

You begin by doing necessary things, like loading the arguments. You make a table of the possible first, second, third, etc. instructions. This listing generates a tree, in which each node represents a choice of instructions; each node is the state of the machine at that point (the value of a function computed) if you follow the tree to that node. The tree gets pruned quickly because you throw out obvious things not to do (like increment an index you don't know).

Question: It seems to me like a shotgun kind of thing.

Answer: Isn't it?

Question: Most times you have some sort of objective in mind?

Answer: Many people would like to believe that if you know what you want to compute you can deduce how to do it. And, in a rational world, that would perhaps be true. But as you will discover, there is an enormous amount of trial and error in these things. Even after you've done all the deduction that can be done, you still have to try a few things. You should not decide beforehand that you will only use such and such an approximation.

I worked out this tree and had various functions computed at the nodes. You don't have to go down more than a few levels to get a tree that is

already unmanageable. But you can rapidly prune the leaves; you'll find you have the same function at two different nodes, and then you prune off the longer path unless it has advantages. If you don't prune diligently you won't get a decent set of programs; you must check at each stage what functions you can compute. You must be careful not to name any constant that need not be named. Say if I do an ADD; I don't say what I'm adding until later, so I can optimize the course of the calculation.

It helps to know how the program is going to end. I knew I had to end with at least one step of Heron's rule; there is no other economical way known to tidy up a square root. Any calculation will be contaminated by rounding errors; to make them as small as possible, one step of Heron's rule is very nice. Of all high order convergent iterations, Heron's rule is fastest on a binary machine.

The argument goes roughly as follows: Although rapidly convergent iterations are infinite in number, it is possible to do some analysis to restrict the kind you have to discuss. For example:

$$x_{n+1} = \phi(x_n) \qquad \text{iteration scheme}$$

This converges to $x_\infty = \phi(x_\infty)$; we then talk about the speed with which this iteration converges. Convergence can be arbitrarily slow. But if ϕ is differentiable and if $|\phi'(x_\infty)| < 1$, then convergence is at least linear; i.e., the number of correct digits will be a linear function of the time spent doing the iteration.

If $\phi'(x_\infty) = 0$ and $\phi''(x_\infty) \neq 0$, then the convergence is quadratic; i.e., the number of correct digits nearly doubles with each iteration.

$$\frac{x_{n+1} - x_\infty}{(x_n - x_\infty)^2} \to \text{constant} \neq 0$$

However there are infinitely many ϕ that will give quadratic convergence to any particular root. All you have to do is write down an equivalent equation.

$$x = \phi(x) \qquad \text{(there are infinitely many of these)}$$

Say you want to solve

$$f(x) = 0 .$$

You could just as easily say you want to solve:

$$\psi(x)f(x) = 0 \qquad \text{(for any } \psi\text{)}$$

Then consider:

$$x = x - \psi(x)f(x) = \phi(x)$$

Saying $x = \phi(x)$ is like saying $f(x) = 0$. Of course, there is the question of choosing $\psi(x)$. Or, what other functions could I get?

But here is something interesting. All quadratically convergent iterations are essentially Newton's method, applied to some equation equivalent to yours.

There has to be a function $F(x) = \psi(x)\cdot f(x)$, which vanishes at the same place that your function does, with the property that:

$$\phi(x) = x - F(x)/F'(x) \qquad \text{(this must be true)}$$

It is not necessary that $F(x)$ be some multiple of $f(x)$, only that they vanish at the same point So if the iteration is quadratically convergent, it is necessarily one in which the iterative function has the above form, for some F which has the appropriate property that $F'(x_\infty) \neq 0$.

We know we want to solve the equation $x^2 = X$; so $f(x) = x^2 - X$. So we ask, what are the equations equivalent to this one? But to do the

iteration, the quotient has to be computable and it had better not be too complicated. What simple functions can you compute; you can add, subtract, multiply, but you might be reluctant to divide very often on some machines. When we limit ourselves to rational functions, $F(x)$ is rational and proportional to $f(x)$. That's a pretty strong limitation. You discover that $x^2 - X$ is about as good a function as you can get and still converge quadratically.

If I write

$$F(x) = x^p(x^2-X) \quad ,$$

for some choices of p this can be cubicly convergent, but then $\phi(x)$ is more complicated to compute.

Some of this rather elaborate theory is discussed in a book by Traub in which he discusses families of iteration methods† (he fails to prove some things he says he does).

The tree is not as ramified as you might at first think, since you have some idea how it must end. You are generating a first approximation to be used in one of these rapidly converging iterations.

Question: Why did you limit yourself to the number of instructions in another program?

Answer: Once I have a program that computes the square root, it is clear that there is no point in looking for programs worse than that one. They might be longer but faster, of course. So I guess it wouldn't be "none longer" but "none much longer." But I had some programs that didn't use much floating point, so most instructions were 1 or 2 cycles. If I was to do anything clever using floating point (which takes 3 or more cycles), I couldn't have a longer program or I'd be slowing it down. This was for

†J.F. Traub, Iterative Methods for the Solution of Equations, Prentice-Hall, 1964.

a 7094; on another machine you might have to think differently, like maybe no more than twice as many instructions. It is important to have an upper bound. You should have a program in hand, or you have nothing to optimize.

Question: I don't see that you can get an upper bound. How many iterations of Heron's rule do you intend on using normally?

Answer: In this program I was using three. On the 7090 program I used two. You can figure out how many you need; you do need at least two, so that the last one gives you an error of less than 1 in the last place. The one before that has to have at least a half-word length correct; it is obvious that you won't get that half word correct with just a few arithmetic operations. That is because the square root is too complicated.

You get indications of how complicated a function is from the entropy theory of approximation; the theory is an attempt to decide how complicated a function is to compute (the theory is at best rudimentary). There are discussions which say analytic functions are infinitely less complicated to compute than, say, nonanalytic ones which satisfy Lipschitz conditions. To see more about this, look into a survey by Timan (Approximation Theory, Pergamon Press). A man named Sprecher also does work in that area.

The digression (in the questions) may have frightened you into thinking that to write a square root routine you have to have spent years studying obstruse 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. If the problem were

complicated, the tree would soon get far too large to encompass in any machine storage you could think of.

In wandering through the trees, it appeared there were several functions which could conceivably be considered as approximations to a square root. Every time you get one of these functions, you find there are some undetermined parameters, and it would be nice to know what they are.

Approximating Functions That Have Symmetry

Let me first mention, with an illustration, the existence of a theory that tells us that certain functions can be best approximated, in a fairly obvious sense. Consider the case of a rational approximation

$$\frac{ax+b}{cx+d} \approx \sqrt{x}$$

on an interval that is cunningly chosen: $\phi \leq x \leq \phi^{-1}$ (The interval is symmetric; but any interval whose endpoints are in the same ratio would do).

Naturally, we want to get our approximation to be as good as possible in some sense. The most natural is relative error for floating point numbers. The relative error is perhaps most easily written in the log form.*

$$\min_{a,b,c,d} \max_{\phi \leq x \leq \phi^{-1}} \left| \ln\left(\frac{ax+b}{cx+d}\right) - \ln \sqrt{x} \right|$$

In discussing a specific problem, I get rather different answers

---

*Alternate ways to measure relative error are:

$$1 - f/F \quad \text{or} \quad 1 - F/f$$

What I have is

$$\pm \ln(f/F) \quad .$$

These are all approximations of the same thing; and they are monotonic functions of each other; if I manage to decrease one, I've also decreased the others, especially if the error is small, so f and F are close.

sometimes, depending on which measure of error I've used. If instead of the above, I used the absolute error,

$$\left|\frac{ax+b}{cx+d} - \sqrt{x}\right| \quad .$$

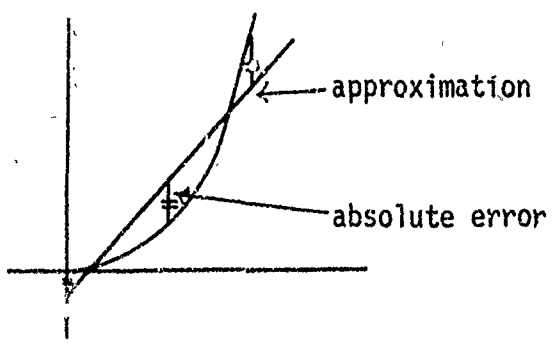
I would get different coefficients. And the coefficients might have been harder to compute.

How Many Coefficients Are There?

With some forethought about the type of function you want to approximate, you can often diminish the labor needed to get the coefficients. It looks like I have four degrees of freedom while actually there are only three (I can divide all coefficients by a constant to make one of them 1). Actually, there are only two coefficients that matter.

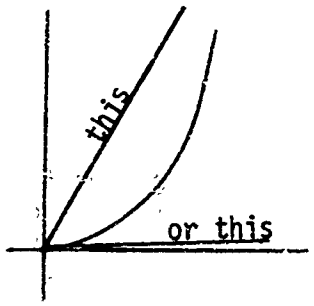
Question: I'm still worried about the fact that errors where the approximation is greater than $\sqrt{x}$ are treated differently from the other side. Is there very much difference there?

Answer: There is a difference between the way the relative form and the absolute form treat errors. If you minimize one you get different coefficients than if you minimize the other. Look at this example. Say you want to approximate something like the following, where the slope goes to zero near the origin.



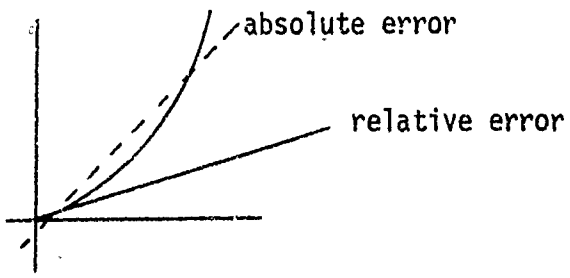
The best linear approximation will have the maximum error in each section equal, using absolute error.

If one error was biggest, I could make it smaller by slightly increasing another, thus decreasing the maximum error.



Using the relative error measure of the ratio of logs, the linear approximation must go to zero at zero. Both approximations are bad.

Had I chosen a slightly less drastic function (one that didn't have zero slope at the origin), the relative error linear approximation must share the slope at the origin and is thus uniquely determined. You really use the first two terms of the Taylor series expansion.



If we restrict ourselves to a sufficiently narrow region, the two measures will not give very different results for the square root function.

Symmetry Helps

I have my problem, to minimize the relative error. The problem is not as complicated as it may seem, when applied to elementary functions, because elementary functions have certain symmetries. The symmetry in the square root may not be obvious. So consider the sine function. It is an odd function and it would be odd to approximate it by something else. So you choose an approximating function with the same symmetry. Of course, the symmetry will depend on the interval chosen.

The symmetry properties are tied in with the interval over which you wish to approximate the function. What is the symmetry, in the region of interest, of the square root? Well, I've somewhat begged the issue by providing the interval $[\phi,\phi^{-1}]$.

Another aspect of these theories is that frequently you can show that a best approximation exists and is unique. Not always, unfortunately, but frequently. The square root is such a case.†

The Best Approximation

Let us assume that a best approximation exists and is unique. Then, observe what happens if I replace x by ξ, its reciprocal. Then ξ is in the interval $\phi \le \xi \le \phi^{-1}$, and the approximation becomes:

$$\frac{b\xi + a}{d\xi + c} \approx \sqrt{1/\xi} \quad \text{or} \quad \frac{d\xi + c}{b\xi + a} = \sqrt{\xi} \quad .$$

The function is replaced by one of the same kind; the function exhibits the same symmetry properties as the square root.

---

†Some books on this subject are: J.R. Rice, The Approximation of Functions, Addison-Wesley, 1964, two volumes; E.W. Cheney, Introduction to Approximation Theory, McGraw-Hill, 1966 (extremely good), and occasional papers by Dunham, and a whole journal of approximation theory. These show lots of circumstances in which the relative error minimum exists and is unique.

Then you can use the same criterion for relative error:

$$\min_{a,b,c,d} \max_{\phi \le \xi \le \phi^{-1}} \left| \ln\left(\frac{d\xi+c}{b\xi+a}\right) - \ln \sqrt{\xi} \right| \quad .$$

This is just the problem we had before. But remember I said that in this circumstance, the solution is unique. If I ever find values for a, b, c, d which work for x, they must also work for ξ. Therefore, the parameters must be related in this way:

$$a = d \quad \text{and} \quad b = c \quad .$$

There are only two independent parameters.†

This is an example of the type of thinking that goes into optimization and here you see there is a systematic theory. You aren't always that lucky. Sometimes you may have to approximate functions in which the standard theory turns out to be inapplicable. The gamma function is an example. There are degeneracies that turn up and then people are reduced to what amounts to a certain amount of intelligent trial and error.

Obviously, in my tree, I had some rational functions like these. And I was able to see what values of the constant would give me the best approximation. Then I was able to work out if that program was as good as some other program in the tree. On the 7090 a program like this was fine.

On the 7094, because of timing changes, overlap and faster floating point it worked out that a different program was best.††

---

†For elementary functions, symmetry properties like this reduce by near two the number of parameters to be varied to seek an optimization. It is important that you find these symmetries, and use an approximating function that has these symmetries, to preserve as much of the character of the function as possible in your implementation.

On most machines, you cannot guarantee that the square root of the reciprocal is the reciprocal of the square root, since reciprocals are not exact. But on a machine that used log representation, you would expect to have to preserve that quite precisely, and you could.

††That program would probably also be best for the CDC, because the floating point is so fast, and the fixed point is so horrible. On CDC, you have to use floating point to do any interesting arithmetic.

The Approximation for Starting Heron's Rule

So let's look at this approximation in detail.

$$X = 2^{2I-J} \cdot F \qquad \frac{1}{2} \leq F < 1$$

$$J = 0 \text{ or } 1$$

$$y_0 = 2^I\left(C + \frac{F}{2} - \frac{J}{4}\right) \qquad \text{starting approximation for the square root}$$

Where did y_0 come from? I said it came from the tree and that is what got us going. You see, in the tree there existed, among other sets of instructions, an initial sequence that went like this:

```
CLA   operand > 0    (leave out test for sign and 0 in this discussion)
STO   X (op'd)       store X
ORA   776 [77···7]   ← fraction part: this picks J off the exponent
ARS   1              right shift 1
ADD   X              fixed point add (J added to fraction of X)
ADD   constant       to be figured out later
ARS   1
STO   S              store the approximation
  ↓ Heron's rule 3 times (coded, not in a loop as it is very short)
```

This sequence of instructions is extremely difficult to explain, so I'll change it slightly for didactic purposes only. Replace the ORA 77677···7 by ANA 00100···0 and adjust the constant. The ORA is used because it takes 2 cycles and allows overlap; the ANA takes 3 cycles and suppresses overlap.

Question: You were really able to discover that the OR with the particular bits you had there was the same function as the AND with the other bits and a different constant. How did you happen to pick those particular bits?

Answer: It's not the particular bit pattern; it is that they are the same function. If I OR and later add, I get the same thing as if I AND and later add a different constant. It's because we're dealing with positive numbers and the X recurs. What looks like one ADD node of the tree also includes SUB, ADD-carry-logical, ADD magnitude, SUB magnitude -- they're all imbedded in the same node of the tree.

<u>What Happens in the Code</u>

	J = 0	J = 1
CLA (clear and add)		
STO (store)		
ANA (and of accumulator)	J = 0	J = 1 in accumulator B_8 = binary point 8 bits to the right of the sign bit
ARS (right shift)	$\frac{1}{2}J = 0$	$\frac{1}{2}J = \frac{1}{2}$
ADD (fixed point)	$2I + F$ (2I is actually biased by 128)	$2I - J + F + \frac{1}{2}$ (but $F + \frac{1}{2}$ will carry into exponent) $(2I-J+1) + (F - \frac{1}{2})$ $(2I) + (F - \frac{1}{2})$
ARS	$(I) + (\frac{1}{2}F)$	$(I) + (\frac{1}{2}F - \frac{1}{4})$
ADD	$(I) + (\frac{1}{2}F + C)$	$(I) + (\frac{1}{2}F - \frac{1}{4} + C)$
(last two instructions have been swapped, doesn't matter except maybe for overflow)	recall J=0	recall J=1

Question: There's got to be more to it than you just having figured it out on a big sheet of paper with a tree. You had to work out the functions and that meant a lot of interpretation of what all those bits meant and I think it is funny.

Answer: Okay.

We interpret the result as a floating point number. 'I' goes into the characteristic, the rest is the fractional part, and we have our approximation $y_0 = 2^I(C+F/2+J/4)$.

Question: How long did it take you to run the tree?

Answer: I used to work on it 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 they computed the same function, using leftover telephone wire. It really was mechanical; no great cleverness went into it. Some equivalences, like ANDing one constant and ORing its complement, are obvious. There could be more subtle equivalences that I might not have noticed, but I was only dealing with rational functions.

Question: This rather reminds one of a chess game. There, at each move you have roughly 15 moves. You seemed to look at all possible next steps, not just the reasonable ones. So you had 64 choices and you claimed you went 50 steps deep.

Answer: It didn't actually get that bad. Although I was prepared to go that deep, I did know what the end was going to be like, and a little of how to start. There were questions of ADD, ADD logical, SUB, but those are relatively trivial. If it had been as bad as it sounds, it ought not to have been done. After this was done, a man by the name of Hirondo Kuki came up with exactly the code I have written with the AND instruction. He had constructed it himself, whereas I had the one with the OR, constructed by the tree.

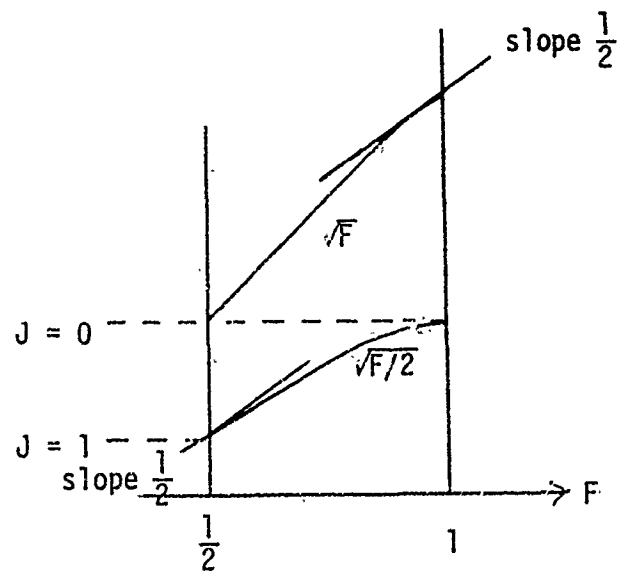
Choosing C

In order to explain what to do with the approximation, I will again have to introduce an artifice. The question which arises now is how best to choose C. It is not at all clear that one value of C should be chosen. The program could have been written to use different C's if J was 1 or 0. The approximation then would be:

$$y_0 = 2^I(C_J + F/2 - J/4)$$

It will turn out that really only one value for C is needed; 2 values don't make that much difference. To see how this works, it is clear that the value of I is irrelevant; so ignore 2^I for now.

For the two values of J, I have two graphs.



I'm approximating $\sqrt{F}$ by a linear function of slope $\frac{1}{2}$. C_J has the task of shifting the line up and down in parallel.

I already knew I could approximate $\sqrt{F}$ by a line with slope $\frac{1}{2}$, so that when F/2 appeared in the tree, it had to be scrutinized most carefully.

All I have to do is choose the vertical displacement so as to minimize the error. However this is not the nicest way to think of things. What I want to do is:

$J = 0$ $\qquad\qquad$ $J = 1$

$y_0 = C_0 + F/2 \approx \sqrt{F}$ $\qquad\qquad$ $y_0 = C_1 + F/2 - 1/4 \approx \sqrt{F/2}$

Do a transformation for $J = 1$

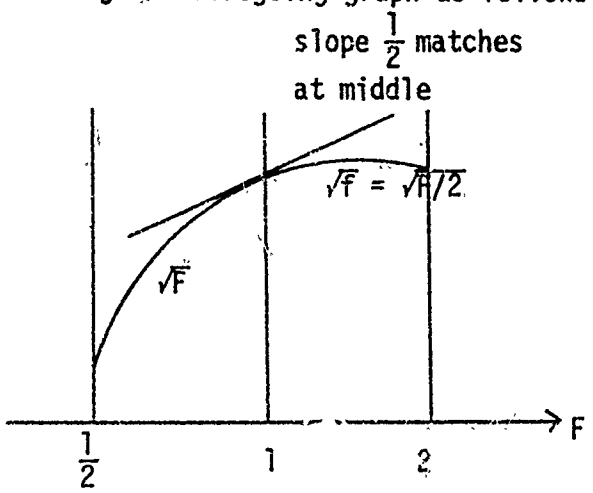
$$F \equiv \frac{1}{2}f$$

$$y_0 \equiv \frac{1}{2}Y_0 \doteq \frac{1}{2}\sqrt{f}$$

$$Y_0 = (2C_1 - \frac{1}{2}) + \frac{f}{2} \approx \sqrt{f}$$

$$1 \leq f < 2$$

Now I have the same function for $J = 0$ and $J = 1$; it is just a function of a different letter. And it is on a different interval. That is tantamount to remaking the foregoing graph as follows:



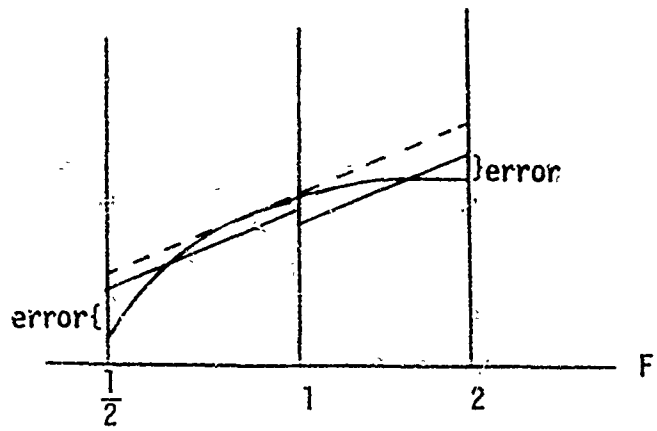
It is just a matter of scaling, so the relative error is still the same.

What Hirondo Kuki had done was to use a linear approximation that was one straight line on the big graph. He did that by choosing C_0 and C_1 so that the two constant expressions would be the same.

$$\left.\begin{aligned} C_0 &= 2C_1 - \frac{1}{2} \\ C_0 &= C_1 \end{aligned}\right\} C = \frac{1}{2}$$

But my tree had led me down a different path. I still wanted to choose one constant if I could get away with it. But what were the two constants to choose for the two graphs?

The Best Value for C



As I let the tangent be displaced downward, letting C_1 and C_0 be equal, the line will break. It will go down twice as fast on the right as on the left.† That is rather nice because it means that the error at the left end has the same relative importance as the error at the right end. So my object was to choose C in such a way that the error in the middle is just as bad in a relative sense as the errors at each end. That would minimize the maximum of the relative error.

It wasn't really the relative error I wanted to make small. It is almost that. Recall what Heron's rule says:

$$\text{If} \quad y_0 = \sqrt{X}\left(\frac{1+\delta_0}{1-\delta_0}\right)$$

$$y_1 = \frac{1}{2}(Y_0 + X/Y_0) = \sqrt{X}\left(\frac{1+\delta_0^2}{1-\delta_0^2}\right) = \sqrt{X}\left(\frac{1+\delta_1}{1-\delta_1}\right)$$

$$\delta_1 = \delta_0^2$$

---

†If I move the line down by reducing C_0, I can also reduce C_1 and bring the line down, but since C_0 and C_1 are the same, you'll see that C_1 is doubled in the constant $(2C_1 - \frac{1}{2})$. So decreasing C_0 reduces the constant in brackets twice as much and there is a break in the line.

So it was δ_0 I wanted to minimize. If I minimize the maximum that δ_0 takes over this range, then I get the best approximation I can possibly get, when the approximation is as bad as it ever becomes on that interval.

The right value for C worked out to be:

$$C = \frac{1}{\sqrt{1/8} + \sqrt{\sqrt{8}+1/8}} = .4826004\cdots_{10}$$

$$= .367056630_8$$

The Best Value To Use For C

So C is a little less than a half. Needless to say, although I have the optimum value for C, that value is not actually optimum. By this time you would expect that every time you have accomplished your goal, there is yet another consideration. So let me say that it is true that C should not differ from this by more than a few units in the last place. The fact remains that in order really to minimize the error at the very end of the program you have to see what happens to rounding errors. It turns out that by the time you're finished with the iteration it is really the rounding errors that are much more important than by the error caused by the fact that we are using an approximation in the first place and making it better by Heron's Rule. The error, committed because we use Heron's Rule three times instead of using the exact square root (usually called truncation error, in the sense of truncation of an infinite process), turns out to be extremely small.

To within a factor of two, here are these 'truncation' errors.

$\delta_0 < .0177$

$\delta_1 < .000313\cdots$ after one application of Heron's rule

$\delta_2 < 9.841 \times 10^{-8}$

$\delta_3 < 9.68 \times 10^{-16}$

After 3 applications of Heron's rule, we have a very accurate result, in the absence of rounding error. We only need 27 bits, which is 10^{-8} or 10^{-9}. The error is down to 10^{-13}.

Question: You said that you wanted to pick a C accurate to within a few ulp's, but it seems that actually you can have quite a wide range on C, and still have the truncation error small enough.

Answer: That is true. We could still get the truncation error small even with $C = \frac{1}{2}$. That's what Kuki did and his truncation error was down to 10^{-10} or so. But there was something else I wanted. Remember, this is for didactic purposes as well as for a program and I wanted to do really well, to do the best possible program. So 10^{-13} is how small δ_3 could be without rounding errors, and if you want to keep it that small you cannot change C by much.

This tells us that the program is now feasible.

<u>What If You Use a Less Accurate C</u>

If you use the less accurate value for C (say $\frac{1}{2}$) so that δ_3 is roughly 10^{-10}, instead of having $\delta_3 \sim 10^{-13}$, the machine will be able to see the difference. It shows up in the running time of one of the tests. You have to do some tests to find out what is the best value for C and how big the error is. In order to be able to make that decision, it'll be quite important that there be 13 zeroes in δ_3. If I had only 10 zeroes there, the time needed to find out how good the program was would have been multiplied by 10^3.

Question: But you don't have that many digits around.

Answer: Actually, when you try to minimize the maximum error, it doesn't look like: $\cup$ but rather it is like $\vee$. If you change C from the optimum, the error will change more abruptly than is customary for minimization. So I really have to stay within a few ulp's of C.

Question: You just said that if $C = \frac{1}{2}$, you do better than a few ulps. You still have more accuracy than the machine can hold.

Answer: That's true, but on the other hand for the best C, $\delta_3 \sim 10^{-13}$. If I change C to $\frac{1}{2}$, $\delta_3 \sim 10^{-10}$, larger by a factor of 10^3. The error is a thousand times as big, by using a slightly less accurate C.

The machine will see that error, you'll see.

Question: That's a different argument than you've been using for why you wanted the best value of C.

Answer: I only want that much precision in C because I want to know what the best value of the constant is. But you're right. If I used a value for C as different as .5, I would clearly be able to get an adequate square root routine and that's what Kuki did.

Question: And it would be just as accurate as yours?

Answer: No, that would not be true. Kuki's routine has an error of .5001 ulps, while mine is .50000163, and there are other little discrepancies.

Question: Is that a large difference in error?

Answer: Actually, it looks very large to me right now. But as far as the ordinary innocent user is concerned, he'd not be able to tell the difference, except that my program was faster as well as more accurate. As long as I'm going to change Kuki's program, I might as well change it to a program that can't be beaten.

Think of it in practical terms. If every time someone thought of a way to improve a program epsilonically, he said "come on now librarian, put this on the system's tape", whatever he hoped to save the users would be blown by the cost of the new library update. So I said I'd make mine sufficiently good that it won't be worth someone else's while to introduce a new library update.

Question: Hadn't that been reached with Kuki's .5001?

Answer: No, his program was a lot slower than mine, too. He took 77 microsec instead of 63.

Question: What if you had used the exact same program, just with a constant closer to $\frac{1}{2}$?

Question: Wouldn't you have ended up with his error and your speed?

Answer: Yes. But I was determined to get the best possible program. 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 where Kuki did. We can't afford too many guys like me. But we can't afford to do without them either.

How Accurate Are the Results?

To find out how accurate the final result is, we have to examine the coding for Herons' rule.

```
STO   X
STO   S        ≈ approximation
CLA   X
FDH   S        to get X/S in accumulator
XCA
FAD   S        floating add (could have done fixed add if charac-
                             teristics lined up, which they usually
                             did)
ADD   -1       division by 2 (subtract 1 from the characteristic)
STO   S
```

This is the setup for two of the three Heron's steps.

We have done

$$S \leftarrow \frac{1}{2}(S + X/S)$$

Truncation has occurred in doing X/S, and in doing +. This has introduced roundoff.

The third Heron rule puts in a round instruction after the ADD -1.

In the first two cycles of Heron's rule, the error is going to be smaller than the numbers quoted for the δ's, even taking rounding errors into account. Rounding errors actually make the approximation better than it otherwise would have been. The only possible exceptions would be if the original approximation were sufficiently close to the root that rounding errors could make things worse. But from the graph, that only happens for a very few numbers (where the straight lines cut the graph). For all the other numbers, rounding errors actually help.

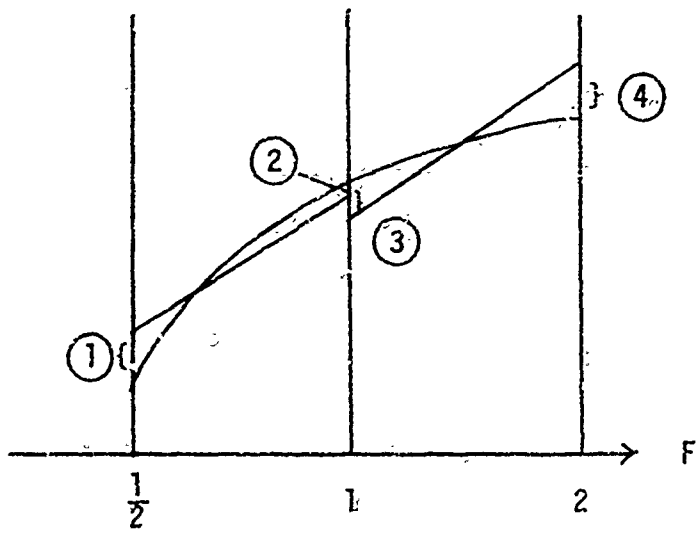
If you believe that everybody who writes programs does this sort of thing, or should do this, you've missed the point. <u>The issue is to see how well we could do and how much it would cost.</u>

Question: Why did you write out Heron's rule three times?

Answer: The index register instructions take 3 cycles for testing, 3 for setting and 2 for restoration. Why bother when the loop is so short? Kuki used a loop; that's why his takes longer.

Review

In getting the first approximation to the square root, we make linear approximations on each of two intervals, $[\frac{1}{2},1]$ and $[1,2]$, where the second interval is really a translation of the situation when $J = 1$. We get two different, but parallel, line segments because we insist on using the same value of C for both graphs.



The values of δ_0 that you would compute at points 1, 3, and 4 would all be the same, although at 3 it has the opposite sign. At point 2, δ_0 would be a little bit better.

C is chosen to minimize the maximum of the relative errors as it happens in terms of the δ's. The reason that you want the minimum for δ_0 is that for each step of Heron's rule, $\delta_{i+1} = \delta_i^2$.

Once we know what the first error is, we can work out what the next several will be. Then we can tell how many steps of Heron's rule are needed. For example: $\delta_2 < 9.841 \times 10^{-8}$. This is somewhat larger than we want the relative error to be; $2^{-26} \approx 1.5 \times 10^{-8}$; so we can't stop with only two iterations.

$$\delta_3 < 9.685 \times 10^{-15} < 10^{-14} \quad \text{(without rounding)}$$

Therefore: $0 \leq$ rel. error in $y_3 < 2 \times 10^{-14}$ (error before rounding)

Rounding Errors Don't Hurt Sometimes

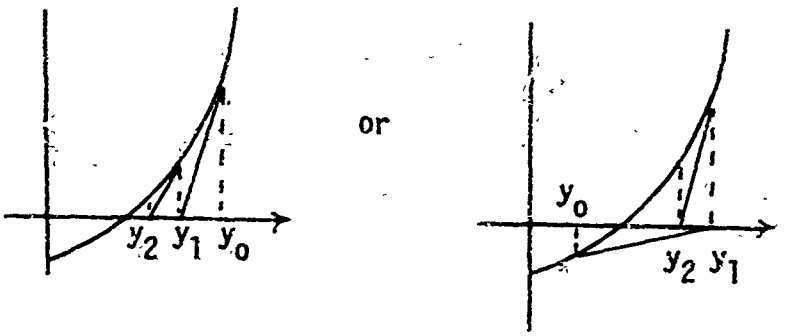
Now, let us look and see why rounding errors, up to the third iteration, do not make things appreciably worse. Here is the code again.

```
          STO   S ≈ √A
          CLA   A
          FDH   S   A/S in accumulator
exchange  XCA
          FAD   S
          SUB   = 0001000000000  divide by 2 by subtracting from exponent
          STO   S = 1/2(S - A/S)
          DO this again
```

At the end of one iteration, the error, δ_1, would be < .000313, if we had committed no rounding errors. What I will show is that the error actually is no worse than δ_1, even though there have been rounding errors.

We are only interested to know if the error is appreciable and not just a few units in the last place of the square root. So say that the error is close to the computed bound, that is, about twice δ_1. Then it looks like a rounding error or two could conceivably make things worse. But they don't. And that is because in Heron's rule, the iterates, except possibly for the first one, decrease toward the square root. The approximations are successively decreasing.

Heron's rule is just Newton's method applied to the graph $S^2 = A$. You draw tangents at the points y_i.



You can also see that they decrease by showing that

$$\tfrac{1}{2}(S + A/S) - S > 0 \quad .$$

You do have to have $S > \sqrt{A}$ for this to work, but after the first iteration and maybe before, it will be.

The only effect rounding errors will have on this monotonically decreasing sequence is to maybe speed things up a little bit, if you're far away. Why? You compute A/S and truncate so you throw a little bit away. The division by 2 doesn't affect anything. Then you do a floating add and throw away a bit more. The net affect is to make your approximation a bit smaller than it would have been without rounding errors. But you could only object if you were so close to the root that throwing those bits away took you below the root. We aren't anywhere near that close when the error is big. So the error bounds I quoted are certain to be valid, in spite of rounding errors. This is a rare circumstance, when the rounding errors help.

Rounding the Third Time Only Is Sufficient

The third application of Heron's rule includes a round instruction.

```
  CLA   A
   :
→ FAD   S
  SUM
→ FRN
  STO   S
```

If the machine had a rounded add you could use that instead of FAD and FRN. To see why the FRN is all that is needed, we need to look at what is in the registers at this point.

In a double length register called the AC and MQ, we will have:

$$(S + A/S)/2 \quad .$$

A/S has been truncated. After doing the add, there may be some bits in the MQ (the lower half of the word). FRN will round the double length word and put the single length result in the AC. It adds half in the last place of AC (which is adding 1 in the first place of the MQ).

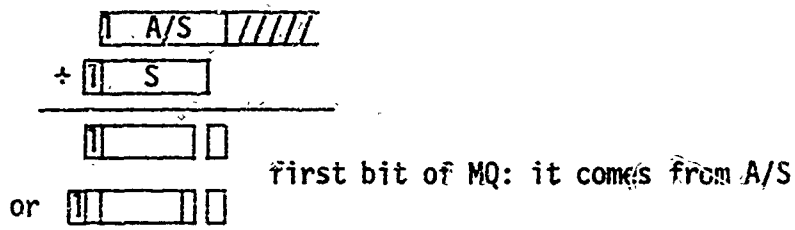
The truncating error in A/S is of no consequence. Why? Normally, at this stage (the third application), $S > \sqrt{A}$ by a little. Hence $A/S < \sqrt{A}$ by a little, and $A/S < S$. Now, when I add A/S to S, there are two possibilities: the exponents are the same, or the exponent of A/S is 1 less than that of S. (Remember, the error at this point is roughly δ_2 or 10^{-7}).

Case 1: exponents equal

```
  [1  A/S  |/////]  digits thrown away
+ [1   S   ]
-------------------
 [1][          ]
      ↑
      becomes first bit of MQ because of the overflow when adding
```

All of the other digits that have been lost from A/S would have played no role anyway, because when I round, I add 1 to the first bit of the MQ and I do have something there. The truncation didn't matter because I merely threw away digits I would not have looked at even had I had them.

Case 2. exponents differ by 1



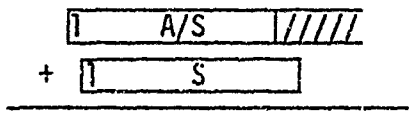
Again, you can see that the digits lost from truncating A/S would not have been used.

Question: What if your machine obeyed the rule 'round to nearest even'?

Answer: Then I might want to know what those other digits were. But remember, here I'm talking only about a 7094. I guess this is the first situation I've seen in which rounding to nearest even might be less than advantageous.

S Cannot Be Much Too Small

You should remember that by this time our approximation S is extremely good. Its relative error is down to around 10^{-7}. So my assumptions are valid, unless our approximation was so good that a bunch of things happened that couldn't happen, so that A/S is too big ($S < \sqrt{A}$). Then the situation is:



Then the lost digits would matter. But this could only happen if S is

appreciably smaller than it should be; one or two units in the last place won't do, because that won't happen. The total error in each iteration is less than a unit in the last place. How could I possibly jump past the square root and be too small by a unit in the last place?

Question: Could that happen if your initial approximation was too small?

Answer: However bad the initial approximation was, I've done a couple steps of Heron's rule. They tend to make the approximation too big unless I was already so close that the rounding error dropped me down. But the total error is less than a unit in the last place -- 1 unit from the division and 1 from the add, but there is the factor of 2.

If the rounding error were exactly a unit in the last place, the situation above (with exponent of A/S > exponent of S) could occur only if I were dropped on the wrong side of a power of 2. But as we will see later, things work out even in this case. I claim that this will never happen.

Thus the digits lost in truncating A/S don't matter. On the last application of Heron's rule we round normally, by adding half in the last place to a number whose relative error is 2×10^{-14}.

Incorrect Rounding Can Happen

We run a certain risk, in that if we looked at the correct square root just before rounding it, the root would have, in the MQ (second word), a 0 and then a long string of binary 1's and some garbage. Because our number is in error by 2×10^{-14} (more than a few units in the last place for double precision), it is possible that what is in the MQ is actually bigger than that, namely a 1, a bunch of zeros and then some bigger garbage. Then you see we would round up instead of down.

The question is, how often does this happen? This is the only way we can get an incorrectly rounded result. In all other circumstances, we have everything that anyone could want.

It is actually possible to discover how close we come to always returning the correctly rounded result. Of course, you can't do this for many functions, but we'll do it for this one.

Playing With Last Digits

We will digress to consider some examples of playing around with last digits, to get a modest feeling for the digits and the way they behave. That is, I'd like you to get used to the integer theoretic approach to rounding errors. When I write it on the board, it will seem much more complicated than it really is, simply because I have to write it down. Once you get used to it, you will be able to follow, fairly easily, calculations of this kind on any machine where the calculations have any value.

I will consider what happens to Heron's rule for a certain set of approximations, namely numbers very close to 1, whose square roots we want. Some interesting things happen.

So let us look at numbers of the form:

$A = 1 + 2^{-26}n$ $\qquad$ n is a small integer > 0

$S \approx \sqrt{A} = 1 + 2^{-27}n - 2^{-55}n^2 + \cdots$

just the power series expansion for $(1+2^{-26}n)^{1/2}$

Now, how do we round root A correctly on our 27 bit machine? If the leading digit is 1, the last digit is 2^{-26}. $\sqrt{A}$ has a $2^{-27}n$ in it, so there is one digit to the right of what can be held in 27 bits. And then there is another term $(2^{-55}n^2)$ to be subtracted off.

If n is odd, there is an extra half sticking out into the MQ, but then some small number gets subtracted:

$$\begin{array}{rl} & \boxed{10\cdots\cdots\cdots} \\ - & \boxed{0\cdots\cdots\cdots 0xxx} \\ \hline & \boxed{011\cdots\cdots 1xxx} \quad \text{MQ} \end{array}$$

When n is even, $\sqrt{A}$ is really a multiple of 2^{-26} and we have:

$$\begin{array}{rl} & \boxed{1}\boxed{0\cdots\cdots\cdots} \\ - & \boxed{0\cdots\cdots\cdots 0xxx} \\ \hline & \boxed{111\cdots 11xxx} \quad \text{MQ} \end{array}$$

Therefore, to round $\sqrt{A}$ correctly, we should use:

$$\sqrt{A} = 1 + 2^{-26}[\tfrac{n}{2}] \qquad [\text{largest integer in } \tfrac{n}{2}]$$

If n is even, $[\frac{n}{2}] = \frac{n}{2}$, and we have

$$\sqrt{A} = 1 + 2^{-26}\,\frac{n}{2} = 1 + 2^{-27}n$$

If n is odd, the extra half that sticks out into the MQ will get subtracted away, so the rounding is correct as stated.

However, our S may not look like the power series. We may have:

$$S_2 = 1 + 2^{-26}\{[\tfrac{n}{2}] + k\} \quad \text{where } k \text{ is a modest integer} \geq -1$$

(S_2 could be small by a unit in the last place.) Now what happens in Heron's rule?

$$A/S_2 = (1+2^{-26}n)(1 - 2^{-26}([\tfrac{n}{2}]+k) + 2^{-52}([\tfrac{n}{2}]+k)^2 - \cdots)$$

using a power series for $\frac{1}{S_2} = (1+2^{-26}([\frac{n}{2}]+k))^{-1}$

$$A/S_2 = 1 + 2^{-26}(n - [\tfrac{n}{2}] - k) + 2^{-52}([\tfrac{n}{2}]+k)([\tfrac{n}{2}]+k-n) + \cdots$$

That's the quotient, but of course that is not what will happen when you truncate. What will happen when you truncate depends on whether the term in 2^{-52} is positive or negative. So there are some conditions on k to check.

We've had an argument already that showed that whether A/S_2 is truncated or not is irrelevant. If you don't believe that, run through the cases when k is a small integer, and see what happens.

Unfortunately, if k is a negative integer equal to $[\frac{n}{2}]$, or $([\frac{n}{2}]+k-n) = 0$ so that the term in 2^{-52} vanishes, you have to look at the next term in the series to find out what is going to happen.

Question: When you say truncated do you mean truncated in the sense that numerical analysts use it?

Answer: No, I mean machine chopped.

Question: Then, why do you care about things far to the right?

Answer: The quotient is exact if you write out the whole series, but the machine only writes out the first 27 bits of it. Those 27 bits will have contributions from later terms. If the 2^{-52} term is positive, the bits simply get thrown away. But if that term is negative, 1 will be subtracted from the 27th bit and this will alter the truncated result. So you have to look at the sign of all the terms after the first two.

Now we add S and divide by 2.

$$A/S_2+S_2 = 1 + 2^{-26}(n-[\tfrac{n}{2}]-k) + 2^{-52}(\ )(\ ) + \cdots + 1 + 2^{-26}([\tfrac{n}{2}]+k)$$

$$= 2 + 2^{-26}(n-[\tfrac{n}{2}]+[\tfrac{n}{2}]-k+k) + 2^{-52}(\ )(\ ) + \cdots$$

$$= 2 + 2^{-26}n + 2^{-52}(\ )(\ ) + \cdots$$

$$(A/S_2+S_2)/2 = 1 + 2^{-26}[\tfrac{n}{2}] + 2^{-53}(\ )(\ ) + \cdots$$

There are two possibilities for the 2^{-53} term. It could be positive

or zero. Then, if n is odd, adding half in the last place will bump up the sum.

Specific Example of Incorrect Rounding

$$n = 1 \qquad k = 1 \qquad [\tfrac{n}{2}] = 0$$

So $A = 1 + 2^{-26}$ and $S_2 = 1 + 2^{-26}$

$$A/S_2 = 1$$
$$(A/S_2 + S_2)/2 = 1 + 2^{-27}$$

$$\boxed{10\cdots\cdots0}\ \boxed{10\cdots\cdots0}$$

This then gets rounded up to $1 + 2^{-26}$ or $\boxed{10\cdots\cdots01}$, which reproduces S_2. That is bad because the square root of A is actually a little less than $1 + 2^{-27}$, and so the result should have been rounded down to 1. It is easy to do this analysis for numbers A a little bigger than 1. If $A = 1 - 2^{-27}n$ (a number is a little less than 1, the leading bit represents a half and the last bit 2^{-27}), similar, but more interesting things happen. The only way to see where the bits go is to work with these numbers with pencil and paper. You should verify for small odd n, that for small $k > 0$, you round up when you should round down.

So you see that there are cases when the error in the square root will be more than a half unit in the last place, that is, when the root is rounded up instead of down.

Now I want to study these cases systematically. If you thought that this problem arises only when taking the root of a number near a power of 2, you are in for a surprise.

Decimal Example of Incorrect Rounding

Here is an example in 4 digit decimal arithmetic. This problem can arise with digit patterns that look essentially random.

$$\sqrt{23790000} = 4877.4994\cdots$$

We are using Heron's rule, and all we have to do is make an error of .0006 in the third application (almost correct to single precision before the last application), to get an incorrect result.

Heron's rule would have given us:

$$4877.5$$

which would be rounded to 4878; it should have been 4877.

We will study this phenomenon systematically mainly because it can be done and not because it has some overriding commercial value. It really is an example of what you can do if you are determined. Having done this analysis, we can contemplate doing analyses for other functions, should they become necessary.

Question: What if someone published a routine similar to yours and stated that the error was no more than 2 ulps? Would you accept that?

Answer: It would be a true but terribly pessimistic estimate. He has overestimated by a factor of 4.

Question: What if he said 1 ulp?

Answer: Then I would ask him if the square root was monotonic and he might not be able to prove it from that estimate. Remember that one of the specifications of the program was that if you increase the argument the square root will not decrease. Or that the square root of a perfect square recovers the number.

Question: Your estimate will be more exact and give you those results?

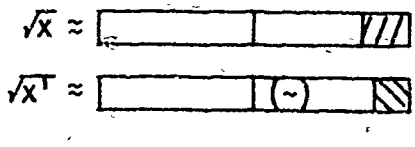
Answer: My estimate will be sufficiently close so that getting those results will be easy. We've got that now. I don't really need to find those 29 numbers. I get what I want by computing the square root almost to double precision; it's a little too big by some numbers near the end of the second word. If I take the square root of a perfect square, the square root fits into the top word; the extra digits from Heron's rule go away when I round.

Question: That ignores truncation errors from subtracting and dividing.

Answer: No, I showed that truncation during division is irrelevant. So I have computed the correct square root plus some garbage far to the right and then I round. Square roots of perfect squares come out.

Monotonicity holds because if I increase an operand by one unit in the last place, I increase its square root by roughly half in the last place, and that increase, in the upper part of the MQ, is not affected much by what's in the lower part of the MQ. Even if the garbage decreases, there is enough increase at the upper part so that the result of rounding will not be to decrease the square root. The root may fail to increase,

but it won't decrease after rounding.



$x' > x$
may have smaller garbage but is bigger than $\sqrt{x}$ by ~1/2 ulp

So the last step of Heron's rule for $\sqrt{x}$ will give me something bigger than the result of the last step for $\sqrt{x}$. Certainly not smaller. Then I round. And monotonicity is preserved. I can only prove this with an error bound of half in the last place.

We want to find out what is the ultimate accuracy in our square root routine. We have:

$$\mathrm{SQRT}(X) = \sqrt{x}(1+e) \qquad \text{rounded to 27 bits}$$
$$|e| < 2\times 10^{-14}$$

$\sqrt{x}(1+e)$ is the number that sits in the registers if you keep the digits that were truncated during division. It is what you have before you round.†

Enumerating the Wrong Roundings

Now we shall enumerate those cases in which the rounding is done incorrectly. Instead of having X a fraction, I will consider X to be an integer of the form:

---

†This program was one of the earliest that was proved to satisfy a certain set of reasonable specifications. There are others that have these properties, like zero finders.

This algorithm will work on the 6400, as the CDC has essentially the same structure as the 7094. Division takes about as long as multiplication, so there is no reason to prefer multiplication. The analysis will then involve 2^{-47} instead of 2^{-26}. On the 6600, the situation is quite different; division takes 3 times as long as multiplication, and multiplications can be overlapped; so this algorithm would not be the most efficient.

$$X = 2^{26}M \quad \text{case 1}$$
$$X = 2^{27}M \quad \text{case 2}$$
$$2^{26} \leq M < 2^{27}$$

M is a 27 bit integer.

Once you have written down this 27 bit integer, changing it by a factor of 2 could drastically change the square root, whereas multiplying by 4 doesn't change the root except by a factor of 2 (which doesn't matter on a binary machine).

There have to be two cases, differing by $\sqrt{2}$; the characteristic is either even or odd.

Define N as the root of X and if you have done your job properly:

$$N - \frac{1}{2} \leq \sqrt{X} < N + \frac{1}{2}$$
$$2^{26} \leq N < 2^{26}\sqrt{2} \quad \text{case 1}$$
$$2^{26}\sqrt{2} \leq N < 2^{27} \quad \text{case 2}$$

N will just barely fit into a single precision word. N is the correct value you would get for the square root of X and then rounding it.

You write X = integer + fraction, where the fraction is less than a half; then things are correct.

If you write X = integer + fraction bigger than a half, then X = integer + 1 - fraction less than a half.

X could be halfway between two integers; then you use a convention for rounding. But that won't happen.

The interesting cases occur when X is near one bound or the other; then you could easily make a mistake. Let's try to see just how close:

If $\sqrt{X} \approx N \pm \frac{1}{2}$, then

$$4X \approx (2N \pm 1)^2 \quad .$$

But I have to know what I mean by approximately equal ($\approx$).

I will have trouble when $4X(1+e)$, which is, after all, what I will have computed, is approximately equal to, or indistinguishable from, $(2N\pm1)^2$. I will have problems when the set of numbers, $\{4X(1+e)\}$, $|e| < 2\times10^{-14}$, is included in $(2N\pm1)^2$.

$$\{4X(1+e)\} \supseteq (2N \pm 1)^2 \qquad |e| < 2\times10^{-14}$$

As I vary e, I will pass back and forth through the division points between the two cases. Those are the points where you decide to round one way or the other.

The situation can only be interesting when:

$$(1+e)^{-2}(2N \pm 1)^2 = 4X = (2N \pm 1)^2 - C$$

C can be positive or negative. While e runs through the values -2×10^{-14} to 2×10^{-14}, through what set of values will C run? Now notice: $4X$ is a big integer; $(2N\pm1)^2$ is also an integer; therefore C must also be an integer. The values used for e are limited in that $(1+e)^{-2}(2N\pm1)^2$ must be an integer also.

How big is C? By looking at the two extremes and at the bound on e, we see:

$$|C| \lesssim 4\times10^{-14} \cdot (2N \pm 1)^2 \quad ; \quad \text{it can't be any bigger than this.}$$

We have the following relations:

$$(2N \pm 1)^2 \equiv C \bmod 2^{28} \qquad \text{Case } 1^{\dagger}$$
$$\equiv C \bmod 2^{29} \qquad \text{Case } 2$$
$$|C| \lesssim 4 \times 10^{-14} \cdot 4X = \cdots < 2000 \quad \text{in case } 1^{\dagger\dagger}$$
$$< 4000 \quad \text{in case } 2$$

There really aren't very many values of C. There are about 4000 in case 1 and 8000 in case 2; that's as many as I've got. A few thousand is like nothing for programming. The situation is actually not as bad as this.

It looks like all I have to do is to let C take all those values, solve the equation for N, find out what X is, and compute the square root and see if I get N. That's all. But it is not at all clear how you'd solve that equation. It wasn't clear to me for quite a while.

Solve By Recurrence

A man by the name of Heilbrand, a renowned number theorist, said isn't there a recurrence for things like that. The one he gave me didn't work, but there was one.

We will pursue the recurrence by which you can solve equations of that kind. We want to solve:

$$(2N \pm 1)^2 \equiv C$$

Case 1	$2^{26} \leq N < 2^{26}\sqrt{2}$	$(2N \pm 1)^2 \equiv C \bmod 2^{28}$
Case 2	$\sqrt{2}2^{26} \leq N < 2^{27}$	$(2N \pm 1)^2 \equiv C \bmod 2^{29}$

Given C, find N. That's the problem.

---

†This means $(2N \pm 1)^2 = C$ times some integer multiple of 2^{28}

$^{\dagger\dagger}|C| \lesssim 4 \times 10^{-14} \cdot 4X$ because $4X$ very nearly equals $(2N \pm 1)^2$.

Then we observe that

$$C \equiv 1 \bmod 8^{\dagger}$$

That cuts down on the interesting numbers; there are only $\frac{1}{8}$ as many.

We are down to about 1000 numbers in case 2 and 500 in case 1; that is so small as to be almost negligible.

All I have to do now is solve the equations for C in the class 1 mod 8. The recurrence involves solving those equations 28 or 29 times. It just takes shifts and a few logical operations, so it isn't very expensive. You could make the program take less time than a division.

We now write the problem as

$$Z^2 \equiv C \bmod 2^m \quad \text{when} \quad C \equiv 1 \bmod 8$$

Z will be $2N \pm 1$; you take your choice.

Bounding Z

The first question is: how many solutions have we got. There are 4 solutions, or there are none.

$$(2^M - Z)^2 \equiv Z^2 \bmod 2^m \quad (\text{is really } (-Z)^2)$$

If Z is negative, compute $2^M - Z$ instead; that doesn't change the square. If the value of Z is rather big, bigger than $\frac{1}{2} \cdot 2^M$, then compute $(2^M - Z)$ and get an answer that is smaller than half of 2^M.

So I might as well assume:

$$0 < Z < 2^{M-1}$$

$^{\dagger}C = (2N \pm 1)^2 \bmod 2^{28}$ or 2^{29}

$C = 4N^2 \pm 4N + 1 = 4N(N \pm 1) + 1$

Either N is even, or $N \pm 1$ is even. Therefore $4N(N \pm 1)$ is a multiple of 8; therefore $C \equiv 1 \bmod 8$.

Z cannot be zero or 2^{M-1}, because Z is odd. Notice that C is odd $\Rightarrow Z^2$ is odd $\Rightarrow Z$ is odd.

I can go farther and observe:

$$(2^{M-1}-Z)^2 \equiv Z^2 \bmod 2^m$$

This happens because of the factor of 2 that appears in the square.†

Thus, Z can be further reduced to:

$$0 < Z < 2^{M-2} \qquad \text{Z can be transformed to this range}$$

The four solutions are:

$$Z,\ 2^{m-1}-Z,\ 2^m-Z,\ 2^{m-1}+Z$$

<u>Only Four Solutions</u>

I've shown there are four. Are there any more? No, and let's see why:

Suppose $\qquad Z^2 \equiv Y^2 \equiv C \bmod 2^m$

$$0 < Y \leq Z < 2^{m-2} \qquad C \equiv 1 \bmod 8$$

there be two solutions in this interval (would give 8 total solutions)? Notice that if $0 < Z < 2^{m-2}$, none of the other solutions is in that interval. Z and Y must be odd (their squares are odd).

$$0 \equiv Z^2-Y^2 \equiv (Z-Y)(Z+Y) \bmod 2^m$$

I can factor 2^m times some integer into those two factors. This is saying that:

$$Z-Y = 2^i p \qquad i \geq 1 \qquad p \text{ is odd or zero}$$

$$Z+Y = 2^j q \qquad j \geq 1^{\dagger\dagger} \qquad q \text{ is odd}$$

---

† $(2^{M-1}-Z)^2 = 2^{2M-2} - 2\cdot 2^{M-1}Z + Z^2 = 2^M(2^{M-2}-Z) + Z^2 \equiv Z^2 \bmod 2^M$

†† j must be ≥ 1; both Z and Y are odd so their sum is even.

When you multiply these two numbers together you must get a number congruent to $0 \bmod 2^m$. This means that:

$$i + j \geq m$$

I will show that this implies Y and Z are equal. First I can't possibly have both $i > 1$ and $j > 1$. If that were true, there is at least a factor of 4 multiplying p and q. Then when I solve these equations, Z and Y come out even.† That can't happen.

Therefore $i > 1$ and $j > 1$ is ruled out. So we try $i = 1$

$$\Rightarrow j \geq m-1$$

$$2^{m-1} \leq 2^j q = Y + Z < 2^{m-1} \quad ^{††} \quad \text{hard to understand}$$

Anything that's hard to understand can't happen. So this doesn't happen. The last case to try is $j = 1$

$$\Rightarrow i \geq m-1$$

$$2^{m-1} \leq 2^i p = Z - Y < 2^{m-2} \quad \text{or} \quad P = 0$$

Therefore, we must have $P = 0$, or $Z = Y$ and there is only one solution.

The Recurrence

Now we need the recurrence. I solve it for $m = 3,4,\ldots,28,29$.

Set: $C_m = C \bmod 2^m \qquad 0 < C_m < 2^m$

We are using the lower m bits to represent C. If C is negative, I go through 2's complement and take the bottom m bits.

$$C_3 = 1 \qquad (\text{since } C \equiv 1 \bmod 8)$$

Let $Z_3 = 1$. Suppose for any m we have

---

† $Z - Y = 4 \cdot 2^{i'} p$

$Z + Y = 4 \cdot 2^{j'} q$

$2Z = 4(2^{i'} p + 2^{j'} q) \Rightarrow Z = 2(2^{i'} p + 2^{j'} q) \Rightarrow Z$ even $\Rightarrow Y$ even

†† $Y < 2^{m-2}$, $Z < 2^{m-2}$; $Y + Z < 2 \cdot 2^{m-2} = 2^{m-1}$

$$Z_m^2 \equiv C \bmod 2^m \qquad \text{(true when } n = 3\text{)}$$

Assume $0 < Z_m < 2^{m-2}$ (can always be done). If $Z_m^2 \equiv C_{m+1} \bmod 2^{m+1}$ then set $Z_{m+1} = Z_m$. Else set $Z_{m+1} = 2^{m-1} - Z_m$.

It is a minor exercise to verify that in fact for a solution at stage m, satisfying this bound, we end up at stage m+1 with a solution satisfying the corresponding bound. The computation involves nothing more than shifts and complementation. We do this, up to 29, for each value of C congruent to 1 mod 8. From the values of Z we get values of N, from N's we get X's; we feed these to the subroutine SQRT and see what it computes. If it computes N, good. If not, we've found a number for which the error was bigger than half in the last place and it rounded up.

This was done, and on 29 occasions these numbers popped up. Actually, it happened a varying number of times depending on C. C was adjusted by diddling the last couple digits to minimize the number of cases.

Review

I've shown how you could hope to prove claims of accuracy for a SQRT program on a machine of a certain structure. There was an integer theoretic equation whereby you could hope to generate all the arguments for which the machine might be expected to be less accurate than to within half a unit in the last place, and inspect them. There were only a couple thousand to look at; on the 7094, only 29 gave errors that were too big. Of course, that example is rather special. Normally you cannot analyze a program as accurately as that. And even if you could, normally you wouldn't; there is a limit on how much people are willing to pay to know everything.

Question: How many other programs have been analyzed like this? It seems that the SQRT is more susceptible to it.

Answer: Oh, infinitely so. I've done analogous things for the cube root, exp, log and trig functions. The last three require a very different point of view and they are much harder to cope with.

Question: What kind of error bounds do you get?

Answer: Around .52 ulps. If I get .513 ulps, I'm happy.

Question: Did you find a similar 29 cases?

Answer: Oh, no. In log, etc., the number of arguments that approach the error bound would be substantial, although none would actually reach the bound.

Other Aspects of SQRT

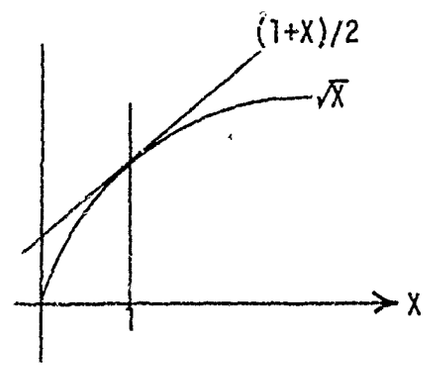
Now let us look at some other aspects of the SQRT program. We shall see what difficulties arise when we try to carry out a similar analysis of a simple routine like SQRT on another machine.

The code will look essentially machine independent, but it is not

because it will not function on some machines. Then we will try to see how the code could be changed to make it as nearly machine independent as possible.

```
      FUNCTION SQRT(X)
      IF (X .LT. 0)          complain (see [6])
      SQRT = 0.
      IF (X .EQ. 0) RETURN
      Y = (1. + X)/2.        first approximation
1     SQRT = (Y + X/Y)/2.
      IF (SQRT .EQ. Y) RETURN
      Y = SQRT
      GØ TØ 1
      END
```

Notice that this uses a poor first approximation to $\sqrt{X}$. Where did it come from?



$(1+X)/2$ is tangent to the graph of $\sqrt{X}$ at $X = 1$ and is bigger than $\sqrt{X}$ everywhere else. So it is an acceptable approximation to use to start Heron's rule. Remember, from now on, applications of Heron's rule produces a descending sequence.

Question: What about rounding errors in Heron's rule?

Answer: In the absence of rounding errors, which we'll consider for now, you would hope to get a descending sequence.

Termination Test

The test for termination is a simple one. On any machine, there are a finite number of representable numbers and therefore, sooner or later you must run out of numbers. And then it stops.

However, arguments like this have a certain fatuity on machines like the 6400 where the number of distinguishable numbers is $2^{60} \approx 10^{18}$. Since doing anything costs about a microsecond, it would take 10^{12} secs to examine them all. Isn't that rather long?

But we know that for Heron's rule, the argument is reasonable. As soon as you have 1 correct digit, 7 more applications of Heron's rule will give you 64 correct digits, and that's more than the CDC can hold. Getting 1 binary digit is not hard. If your first approximation is terribly large, Heron's rule will bring it down roughly by a factor of 2; since no machines have an infinite exponent range, convergence will eventually get faster. It could conceivably require a thousand iterations to get 1 correct digit; then 7 more are enough. So it is machine independent.

We can see that this will work on a machine for which we know neither the base nor the precision. It may be slow. But people used programs like this until they discovered that users liked to take square roots of numbers not very close to 1.

I still want to analyze this program; getting the first approximation is a technical detail that necessarily depends on the structure of the machine. The rest of the code is more interesting.

This code would probably not run on an IBM 650. It wouldn't because the test IF(SQRT .EQ. Y) is too demanding. On machines of this type, the sequence that should be monotonically decreasing isn't. (Try the example of taking $\sqrt{30}$ in 2 decimal, truncated arithmetic.) After a while, the approximations will oscillate around the correct result. So the test always fails and the program never stops.

Weaker Termination Test

The first thing you learn, then, is that you have to put in a weaker test on SQRT and Y.

```
IF(SQRT .GE. Y) RETURN        (this will do)
```

Why will this work? Your first approximation will be too big, or be only too small by 1 ulp. Too small by a unit in the last place is a very good approximation, so you would accept it.

Using Heron's rule, you expect to decrease monotonically toward the square root. But a rounding error may throw you past the root, but only by a unit in the last place. There is a unit error in X/Y. You add and the right shift necessarily reduces the error to $\frac{1}{2}$. Rounding brings the total to 3/2 units. Division by 2 on a nonbinary machine may give another 1/2. So the total is 2 in the last place; that could be considered quite respectable.

We Can Do Better

It is possible to get slightly better accuracy, on most machines by the following dodge:

$$SQRT = Y - (Y - X/Y)/2 \quad .$$

This code takes advantage of the fact that on most machines, subtraction of numbers very close together is done exactly.

X/Y is still in error by 1 ulp; but X - X/Y will be precise and be a very tiny number. Now division by 2 can also be done precisely, even on a nonbinary machine.† The only other error comes from the other subtraction; this normally occurs satisfactorily. There are exceptions, though, say on the CDC; but then division by 2 introduces no error. What you lose on the swing you gain on the roundabout.

On hexadecimal machines, this trick is crucial. On the 360, with its guard digit, the subtraction is done precisely and the division by 2 no longer causes a rounding error. This is the trick used to code SQRT on the 360. The first approximation is better, of course.

The error has thus been reduced from approximately two ulps, to at most 1 ulp, for hexadecimal machines. You only commit one rounding error.

By using a similar dodge on other truncating machines, we can get the error down to 1 unit in the last place. Knuth used this when he wrote the SQRT for the B5500, an octal machine with rounded arithmetic.††

The point of the FORTRAN code was to show that you could do the job in a machine independent way, insofar as you can do anything in a machine independent way, if you are willing to wait long enough.

---

† Y - X/Y is tiny, so it has lots of zeros. Dividing by 2 on any even base machine can at worst add one digit to the number and that can still be represented exactly.

††It is interesting that people who have published analyses of SQRT routines did not use this trick and obtained even for binary machines error bounds of 3/4 ulp; it's a bit hard to see how they got that. This is in a book by Householder on Numerical Analysis, 1953, and by John Todd in some numerical analysis notes that he's been using for the past 40 years.

20. STUDENTS' REPORT ON CDC 6400 SQRT, CABS AND CSQRT

This lecture was a report by the group of students who worked on improving the CDC RUN FORTRAN library versions of SQRT, CABS and CSQRT.

Our basic method is stated by scaling the argument to lie between 1/2 and 2, getting a rational approximation, and then applying three Heron's rules.

The scaling was done by writing $x = F*2^{2I+J}$, $J = 0$ or 1. The rational approximation to $F = \bar{x}$ was

$$\sqrt{\bar{x}} \doteq \frac{a\bar{x}+b}{c\bar{x}+d} = \frac{a\bar{x}+b}{b\bar{x}+d} \quad \text{because the interval is symmetric } \frac{1}{\sqrt{2}} \leq x < \sqrt{2}$$

$$= c + \frac{1-c^2}{x+c} \qquad \text{where } c = a/b$$

The intervals to be approximated over are

$$\frac{1}{2} \leq x < 1 \quad \text{and} \quad 1 \leq x < 2$$

We have an approximation on $\frac{1}{\sqrt{2}} \leq x < \sqrt{2}$. So notice

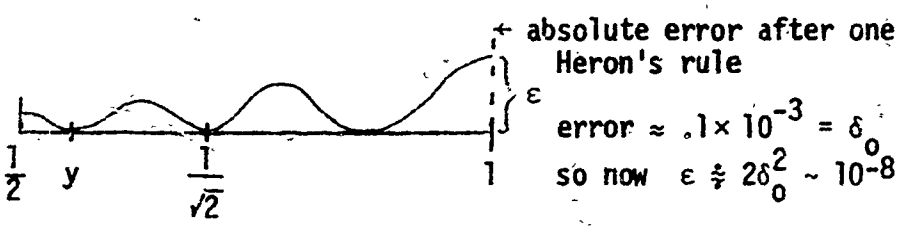
$$f(x) \doteq \sqrt{x}$$

$$f(\alpha x) \doteq \sqrt{\alpha x}$$

$$\frac{1}{\sqrt{\alpha}} f(\alpha x) \doteq \sqrt{x}$$

Now we have $\frac{1}{\alpha^2} \leq x < 1$ or $1 \leq x < \frac{1}{\alpha^2}$. If we pick $\alpha_1 = \sqrt{2}$, $\alpha_2 = \frac{1}{\sqrt{2}}$, we get the needed ranges above. We took an approximation to the square root on one range and mapped it into an approximation on another range; it is easy to compute on the first range; we intend to apply it on the second range.

Now that we have an initial approximation, we will apply Heron's rule. Recall that after the first Heron's rule the error is greater than 0.



A rounding error at this point would be 10^{-14}. Comparing this to the error of 10^{-8}, we see that a rounding error cannot change the graph much.

After one Heron's rule, we wanted to drop that error graph by some absolute value and still keep the relative error nice. We want to drop the graph such that the relative error at y is the same as the relative error at 1; at y, the relative error changes most; at 1, the error improves the least. Say we drop the graph by S

$$\frac{S}{\sqrt{y}} = \frac{E-S}{1}$$

You make the worst errors the best possible by choosing S in this way.

In doing Heron's rule, division by 2 is accomplished by subtracting 1 from the exponent, but as long as you are subtracting something anyway, why not subtract S from the integer part as well; it doesn't cost you anything except the memory reference to get the constant.

You may get an unnormalized number when you subtract S if the argument is close to 1, but it won't be less than half of the number you'll divide it into.

$$y_2 = \frac{1}{2}(y_1 + \frac{x}{y_1})$$

y_1 may be unnormalized by 1 bit.

x will be near 1, y_1 is a little less than 1; so you won't have

problems with the division giving you zero. Then when you do the add, x/y_1 has a leading 1 bit and so nothing is lost there.

Heron's rule is applied normally two more times with the last operation a rounded add. The rounded add will work properly because the exponents are the same (the exponents might differ by 1 if you are very near 1).

Empirically the only case in which this routine gives the wrong answer is when you have 1.0···01 as the argument.

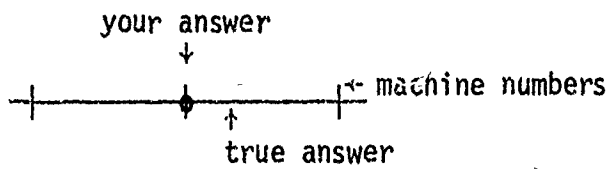
This routine was computed to take 52.8 μsec; the RUN library version takes 80.3 μsec. We did not look at any other versions. (The RUN version was optimized for the 6600, so it may not be so good on the 6400; it uses a strange formula.) The RUN version claim is that out of 200,000 random numbers one answer was off by 3 ulps and all the rest were right.

Accuracy and Tests

Now we'll discuss the accuracy of this routine and the tests that were made on it, how the initial approximation was found and briefly some other possible approaches.

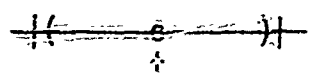
Two accuracy principles:

(1) If you have an answer that is not machine representable and if your program calculates to within $\frac{1}{2}$ ulp of that answer, you have the correctly rounded result.



(2) If the result of a long calculation is machine representable and if your program calculates to within 1 ulp, then the answer you get is that

machine representable number that is the precise result. This second principle is relevant when you discuss complex absolute value, $z = \sqrt{x^2+y^2}$; if z is precisely representable, can you guarantee you'll get that number, say if $x = 3$, $y = 4$.



actual result here

Our result is within 1 ulp so it must be the same.

Question: I'm confused about what you mean by 1 ulp, especially near where the exponent changes.

Kahan: Take the number 2. If you add 1 ulp, that takes you to the next machine number, but if you subtract 1 ulp, you drop down two machine numbers.

Answer: We are talking about numbers between 2^{47} and 2^{48}.

Kahan: So take 2^{48}. You say your result will be correct to within 1 ulp in 2^{48}. So you are talking about $2^{48} \pm 2$. But between 2^{48} and $2^{48} - 2$ there is another representable number, $2^{48} - 1$. So that second principle is in doubt.

The issue is: If the answer should be an integer do you get that integer? To prove that it would suffice to show that before you committed the last rounding error, the result that you rounded was within $\frac{1}{2}$ ulp of what you'd like to get. Then the rounding can't bump you to the wrong place. But that argument needs to be made more precise, especially near the exponent changes. This problem will not arise in CABS because an integer times a power of 2 cannot be an answer.

Question: I still don't see within half an ulp of what, the correct answer or the computed answer?

Kahan: It would be the answer before rounding which is neither correct nor computed.

We tested the routine on 30,000 random numbers on $[\frac{1}{2},2]$ and compared it to the double precision result correctly rounded to single precision. We found no difference between our routine, the RUN version, and the correctly rounded result.

I'll show that our maximum error is $.5 + 2^{-46}$ ulp $\approx .5\ 0^{12}\ 1$ ulp.

$$\begin{aligned}\sqrt{x}\left(\frac{1+\delta}{1-\delta}\right) &= \text{approximation}\\ &= \sqrt{x}(1+\delta)(1+\delta+\delta^2+\cdots)\\ &= \sqrt{x}(1+2\delta+2\delta^2+\cdots)\\ &= \sqrt{x}(1+\epsilon) \qquad \text{relative error} \gtrsim 2\delta\end{aligned}$$

Using Heron's rule

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{X}{x_n}\right)$$

$$\delta_{n+1} = \delta_n^2$$

We found $\delta_0 = .16\times10^{-3} < 2^{-12}$. We got this result by knowing where, on the δ error graph, the error would be maximal.

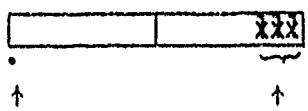
<u>Errors</u> $\delta_0 < 2^{-12}$

$\delta_1 < 2^{-24}$

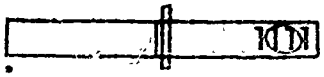
$\delta_1' < (\ ) < 2^{-24}$ (remember the S subtracted)

$\delta_2 < 2^{-48}$ or 2^{-47}

$\delta_3 < 2^{-96}$ or 2^{-94} relative error $\approx 2\delta_3 \approx 8$ units in double precision



binary point last 3 or 4 bits may be in error after three Heron's rules

 error could be $.5 + 2^{-44}$ ulp

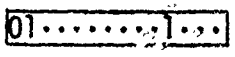
You never actually write down all of X/x_n; you do a truncated division so that all the double precision digits including those four in error never appear. Yet you can claim that the result you get is as good as if you had rounded the whole double precision number [19].

Question: Are you prepared to state for how many arguments your routine will not give the correctly rounded results?

Answer: Not yet.

Kahan: Well, there are at most 6.

Where could you round incorrectly? You could if the actual result was, in the double precision part:



1··· error part would cause a carry, then you would round wrong.

Kahan: If you neglect powers of 4, there are only two different numbers that could cause problems.

Answer: That's if you have ±1 in the last bit of 1. Of those, the +1 was incorrect, the -1 was correct.

Kahan: You should test all numbers I called C, where $C \equiv 1 \bmod 8$ and smaller than 8, so that's +1 or -7.

Good error bounds are needed to show that the routine recovers square roots of perfect squares and preserves monotonicity.

To Show that Square Roots are Recovered

square fits in 48 bits $\boxed{1x\cdots\cdots\cdots x}$

square root $\boxed{xxxxx00000\underset{1}{0}\cdots\cdots\underbrace{xxx}_{\epsilon}}$ If $\epsilon > 0$, rounding gives the correct result.

$\boxed{xxxxx11111\underset{1}{1}\cdots\cdots\underbrace{xxx}_{\epsilon}}$ If $\epsilon < 0$, rounding propagates carries and you recover the correct answer.

Monotonicity

$$x \le y \Rightarrow \sqrt{x} \le \sqrt{y}$$

Assume $A = 2^{47}$. $\sqrt{A} = 2^{23} + \cdots$ $(\sqrt{A} = \sqrt{2} * 2^{23})$

$$(A+1)^{1/2} = A^{1/2} + \underbrace{\frac{1}{2}\frac{1}{\sqrt{A}}}_{\frac{2^{-24}}{\sqrt{2}}} - \frac{1}{8}\frac{1}{(\sqrt{A})^3} + \cdots$$

$$\boxed{1 \qquad \underset{01}{\boxed{1 \quad xxxx}}}$$

If you increase an argument by 1 in the last place, its square root will increase by very much more than the sum of the errors you'll have made in computing the square root before the last rounding. You look at the two numbers before the last rounding, and while there is trash in the last 3 or 4 bits of double precision, the numbers will differ in the right direction by an amount much bigger than that trash. The rounding operation will not destroy the monotonicity.

Reducing the Degrees of Freedom From 4 to 1

$$\sqrt{x} \doteq \frac{ax+b}{cx+d} \qquad [\frac{1}{\bar{x}}, \bar{x}]$$

There is a theorem that says there is a best rational approximation to a function on a given interval. This approximation should hold if $x \leftarrow \frac{1}{x}$.

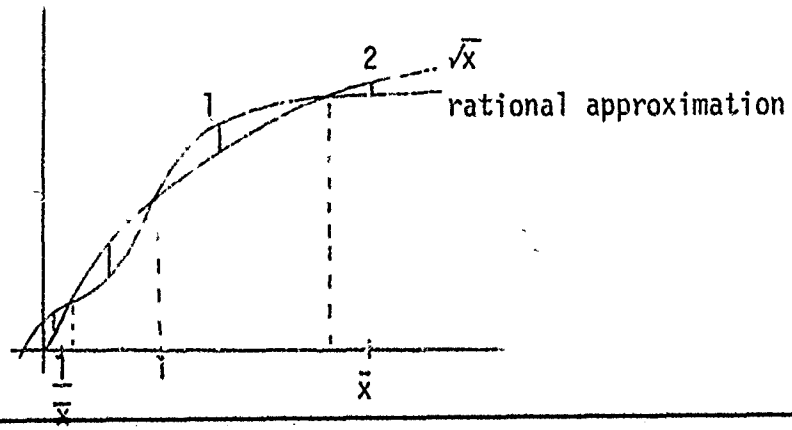
$$\sqrt{\frac{1}{x}} \doteqdot \frac{\frac{a}{x}+b}{\frac{c}{x}+d} \doteqdot \frac{1}{\frac{dx+c}{ax+b}} = \frac{1}{\sqrt{x}}$$

Kahan: $\frac{ax+b}{cx+d}$ is a best approximation in that its relative error has been minimized. It must also hold for $\frac{1}{\sqrt{x}}$ which is in the same range. If we take the measure of error, $\left|\ln \frac{ax+b}{cx+d} - \ln \sqrt{x}\right|$, it is a function of x. The maximum value depends on a,b,c,d.

<u>Theorem.</u> There exists a unique set $\{a,b,c,d\}$, except for a common factor, which minimizes the maximum taken by the relative error. If we compute the relative error, $\left|\ln \frac{1}{\frac{dx+c}{ax+b}} - \ln \frac{1}{\sqrt{x}}\right|$, it is the same kind of function of x, with a and d swapped, b and c swapped. It can also be minimized by apt choice of a, b, c, d. Since the choice is unique, the two functions must really be the same function.

If the function you wish to approximate has a symmetry that is preserved by the way you measure error and by the interval, then you should be able to use the symmetry to decrease the number of independent constants.

So we can say that $y = \frac{ax+b}{bx+a}$.



$$\sqrt{x}\left(\frac{1+\delta}{1-\delta}\right) = \frac{ax+b}{bx+a} = c + \frac{1-c^2}{x+c}$$

We solved for $\delta(x)$, took $\delta'(x) = 0$; had to solve numerically to give points of maximum error; had maximum error at end points also.

The two interior δ's are equal, the two end δ's are equal, regardless. So we set the errors 1 and 2 equal to determine c. It is not possible to divide with c, or with the S, so that that one wrong square root comes out correct. It is wrong because you use Heron's rule.

It was suggested that we look at a third order method like $X^P(X^2-A)$. The resulting P would require 8 operations to compute the function, whereas Heron's rule takes 3. Two steps of Heron's rule (6 operations) is 4th order while one step of the other method (8 operations) is 3rd order.

We tried linear initial approximations, minimizing the absolute error (that's what the RUN version does). The linear approximation has 1 multiply and 1 add; ours has 2 adds and 1 divide, so it is not much more work. We looked at Professor Kahan's initial approximation (for the 7094), but the 1's complement exponent would be almost as much work to unravel as doing the other approximation. Also four Heron's rules would be needed.

<u>CABS</u>

$$Z = \text{CABS}(X,Y) \qquad |Z| = \sqrt{X^2+Y^2}$$

In the RUN compiler, they compute

$$Z = |X|\sqrt{1+(Y/X)^2} \quad , \quad X \geq Y$$

They do this to avoid overflow, but introduce a rounding error in doing the division.

We avoid the division by scaling instead.

$$X = x \cdot 2^n$$

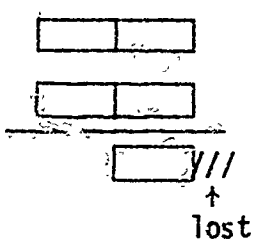
$$Y = y \cdot 2^m \qquad V = y \cdot 2^{m-n}, \quad n > m$$

$$Z = 2^n \sqrt{x^2+v^2}$$

The only problem ther is how exact x^2 and v^2 can be.

x^2 to double precision

v^2 to double precision



First add the two lower halves together. Then add that sum to v^2 (smaller of two numbers), truncated because when we add to x, the truncated part will get lost anyway. We did our own rounded add. Add the sum to the upper and lower halves of x^2, then multiply lower sum by two and add to the upper sum.

So now we have (x^2+v^2) to $\frac{1}{2}$ ulp. We call our SQRT routine and we'd like to get $(x^2+v^2)^{1/2}$ to $\frac{1}{4}$ ulp, but because we may skip over a boundary we get

$$(x^2+v^2)^{1/2} \text{ to } \frac{\sqrt{2}}{4} \text{ ulp} \sim .35 \text{ ulp}$$

plus the error from taking the SQRT which is ~ .5 ulp.

So the error in the final answer is

$$2^n(x^2+v^2)^{1/2}\left(\frac{\sqrt{2}}{4} + \frac{1}{2}\right) \text{ ulp} \sim .854 \text{ ulp}$$

This method takes longer than the RUN version which takes 85 μsec; ours takes ~100 μsec.

In testing on random numbers, we found ours to differ by .8 ulp from

the correctly rounded double precision result, while the RUN version differed by $1\frac{1}{2}$ ulp many places and by 2.4 ulp for one example.

If Z is an integer, will we get that? We do get (x^2+y^2) to within $\frac{1}{2}$ ulp. You don't run into boundary problems (as in principle (2) earlier) because all hypotenuses of Heronian triangles have an odd factor bigger than 1. $z^2 = x^2+y^2$; we're only in trouble if z is a power of 2. Cancel all powers of 2, so at least one of x^2, y^2, z^2 is odd. If z^2 is even, we must have x^2 and y^2 both odd (can't both be even). An odd number squared is congruent to 1 mod 8. You add two such numbers. The sum is congruent to 2 mod 8, but z^2 is congruent to 4 mod 8 or 0 mod 8.

CSQRT

Z = (X,Y)	if $X \geq 0$, U = b, V = c
$\sqrt{Z}$ = (U,V)	
a = CABS(Z)	if X < 0, U = sign(Y)*c, V = sign(Y)*b
b = $((a+\lvert X\rvert)/2)^{1/2}$	
c = Y/2b	

This is what RUN does and is about as accurate as you can do on our machine. If a or b overflows, X or Y was very close to overflowing. The time necessary to do all the checks is not worthwhile. The user should be scaling if he is that close. The only number that could underflow is c, but then you deserve it.

Kahan: You could have avoided overflow by imbedding your CABS routine in this one and deferring the scaling up until later. You might have avoided over/underflow without excessive cost.

(That would even save on RJ to call CABS; RJ is rather slow.)

Kahan: On your SQRT, you said after two Heron's rules your result was good to about 2^{-48}. It seems that with some careful trimming of your constants and trickery, that result before rounding could be good to 2^{-50}. Then your final error would have been something like .505 ulp. It would still preserve monotonicity and recover square roots of perfect squares. But you could not say there was only one operand whose SQRT was wrong. Shouldn't you save 8 μsec and gain a program as good as any others around?

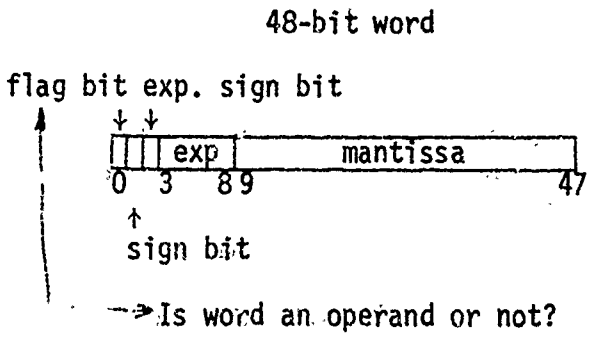
I. STUDENTS' REPORTS ON MACHINE ARITHMETIC

Groups of students investigated various interesting machines in order to determine how numbers are represented, what kind of model of arithmetic could be applied, how overflow and underflow worked, and which of the rules [2] were followed. The students' presentations to the class and the discussions they generated are transcribed below.

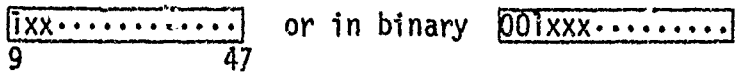
Burroughs B5500 Machine

Information from the manual did not always agree with that given by Professor Kahan or by the Burroughs people in Oakland. So some workings had to be guessed at.

If some of the things told us were right, the manual is visibly wrong.



Representation is in powers of 8 handled by the machine as shown. The exponent is two octal digits and the mantissa is 13 octal digits. The octal point is to the right of the mantissa. A normalized mantissa could be like this:



Exponent is not biased. It is also in sign magnitude, as is the mantissa.

$$-77_8 \leq \text{exp} \leq +77_8$$

	B5500	360/40	7094	BCC Model 1	CDC 6400
Base/representation	8/ sign magnitude	16/ sign magnitude	2/ sign magnitude	2/ 2's complement	2/ 1's complement
word organization	[ ][±][±][6][39] exp integer•	[±][7][24] char •mantissa	[±][8][27] char •mantissa	[±][11][1.xx 36] char mantissa	[±][11][48] char integer•
exponent range	$-77_8 \leq e \leq +77_8$	$-64 \leq e \leq 63$	$-128 \leq e \leq 127$	$-1022 \leq e \leq 1024$	$-1022 \leq e \leq 1024$
usual arithmetic rules	Add 1/2 in last place; normalize results first (guard word)	Normalize, then truncate (guard digit)	Normalize, then truncate (guard word)	Normalize and round using a round and 'sticky' bit (guard word as well)	Chop answer without normalizing (even though it has a guard word)
double precision	Normalized and truncated (do not form full product)	Like single precision	Same as single precision	Same as single precision except for divide	Same as single precision
over/underflow	Characteristic in error, integer correct on overflow	Characteristic off in high order bit, mantissa correct	Special overflow bit for characteristic, mantissa correct - an interrupt to tell you what happened and what kind of operation was being performed	Characteristic off in high order bit, mantissa correct - can choose to trap	Overflow - special number put in register, attempts to use cause abortion Underflow - result set no zero, no warning issued

±0 are presented, but all operations treat any zero as a true zero; only test for zero is on the mantissa, regardless of sign or exponent. Integers are represented as floating point numbers with exponent equal to zero.

Arithmetic operations work on a stack. The two top elements are registers A and B. Operations are performed on these two registers and the result is left in B-register. Another register X is used to hold shifted out digits and the extension of the result of multiplication and in division. X is not mentioned in the manual and its contents are not available to the programmer.

You can't see X but it affects you. There is also a exponent register N.

The rounding rule (not from the manual but from the users so it may not be correct) is that you always round up by $\frac{1}{2}$ in magnitude in the last place. You have a 'bias' up.

How Addition and Subtraction are Performed

If one argument is zero, then the other is the answer.

If the operands have the same exponent, they are added (subtracted) and the answer is rounded up to 13 digits.

Question: To what extent are single precision arithmetic operations characterized by the rule that Knuth uses -- do the operation correctly, take leading 13 digits and round the 14th up?

Answer: It is followed for normalized numbers in single precision (+ - * /).

Question: Is there any case when this isn't true?

Answer: It is not true if the larger operand is not normalized; this can lead to an error of 10% according to the manual.

You can generate unnormalized numbers in subtraction as the result is not normalized after the operation.

According to the Manual

If the shifting to align octal points is by 14 digits or more, the larger operand is taken to be the result, say

$$\left.\begin{array}{r} 0 \cdots 1 \times 8^0 \\ +\ 7 \cdots 7 \times 8^{-14} \end{array}\right\} \text{ result} = 1 \times 8^0$$

The correct result is $1.077\cdots7 \times 8^0$ which rounds to $1.100\cdots0 \times 8^0$ with more than 10% error.

According to Professor Kahan

If the larger operand is unnormalized, shifted out digits are kept in the X-register (you lose only the last 7 in the example above). Addition is performed in 26-bit adders; the answer is shifted left into the B-register until X is empty or the B-register is normalized. Then the result is rounded up. Now you only have difficulty when you are subtracting. If the 1 (in the example above) is unnormalized, the result may be wrong by one unit in the last place in the stored register.

This is the extent to which the B5500 results will vary if you use unnormalized operands, assuming that the manual is wrong.

Comments on the Rules in [2]

1. If -x is representable then x is.

2. The representation of a number is unique except for unnormalized numbers and for ±0. There is no normalize instruction.

If you generate an unnormalized number, whether it remains unnormalized

or not depends on the next operations.

It was intended that you should get the same result from normalized or unnormalized operands. If Professor Chan is right, there is a small discrepancy. If the manual is right, the error is too large to believe.

3. Exact answers are given when possible.

4. Overflow gives correct answer with exponent correct only to modulo 64; overflow toggle is turned on.

Question: That would be okay if the only exponents to cause an overflow were 64 to 127. Then the fact that overflow had occurred would tell you the true exponent. But what happens when you square $77\cdots7 \times 8^{63}$?

Answer: $((8^{13}-1) \times 8^{63})^2 \sim 8^{26} \times 8^{126} = 8^{13} \times 8^{139} = 8^{13} \times 8^{11}$

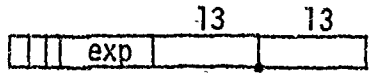
So they should save two characteristic overflow bits. You do not still have the operands. They were on the stack and they have been destroyed. You have irrevocably lost a binary digit. It could have been saved in the exponent sign bit since you know overflow could only occur for positive exponents. You cannot tell if you overflowed by a little or a lot.

The Burroughs people were not willing to make the change to use the exponent sign bit for overflow.

5. Rounding is okay except for unnormalized numbers.

6. You drift up in the sequence $x_{n+1} = (x_n+y) - y$ [2].

Double Precision Representation



Results in double precision are always normalized but truncated.

On multiply there is a problem only when you get 25 and not 26 digits. They keep only 27 digits as they multiply (they don't form the 52 digit

product). There can be an error of 1 unit in the 25th digit.

In subtract, they keep only 26 digits.

Question: If you have (A-B)*C and (A-B) yields an unnormalized result, is it normalized before multiplication?

Answer: No, multiplication is performed, then the 13 most significant digits are normalized and rounded if possible.

If you multiply integers together, the answer tries to stay an integer, i.e. with zero exponent. There are special rules for rounding them.

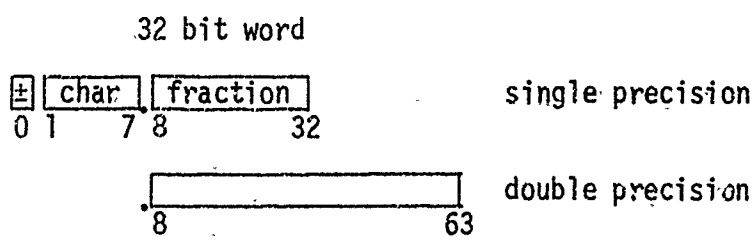
Question: How do ±0 compare in logical operations?

Answer: They would be different.

Question: Suppose I compare (x-y) with -(y-x)?

Answer: You have to distinguish relation operations comparing numbers and Boolean operations on bit strings. In comparisons, ±0 is always zero. There are all the tests >, ≥, etc. There is no test just for sign except by a Boolean operation.

IBM 360/40



It uses true hexadecimal representation:

biased exponent by $64_{10} = 40_{16}$

characteristic $0 \leq C \leq 127$ so $-64 \leq \exp \leq 63$

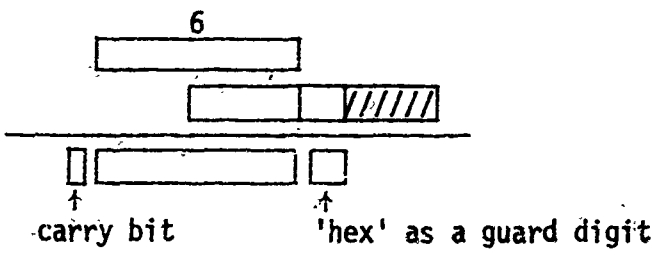
number is fraction $\times 16^{\exp}$

$5.4 \times 10^{-79} \leq f \leq 7.2 \times 10^{75}$ Representable numbers expressed in sign magnitude,

The rounding rule is: Do operation to infinite precision. Then round to 6 or 14 hexadecimal digits.

When $\left\{\begin{matrix}\text{adding}\\ \text{subtracting}\end{matrix}\right\}$ magnitudes, the rule is

$$Z = \text{TRUNC}(X \pm \text{TRUNC}(Y))$$



The result is left shifted until the answer is normalized. It could be off almost one hexadecimal digit in the last place.

If you get an overflow on adding

the result is right shifted by one 'hex' (4 bits)

so you lose 4 digits.

Say the bit lost was F ($= 15_{10}$). Then the answer is truncated.

We might have

1FFFFF|F

Then the answer is just $1FFFFF_{16}$, not 200000_{16} as might be more reasonable.

If the machine is given unnormalized operands, it first normalizes them.

In multiplication, it produces a 14-hexadecimal-digit result (the last two are always zero) (28 in double precision). Actually in double precision,

it does a curious thing. It gets the 28 digits, truncates to 15, normalizes by left shifting and truncates to 14 digits.

(A long time ago they used to truncate to 14 hex digits first, then left shift if there was a high order zero. The answer was already truncated.)

You had $1.0*X \neq X$ because the last hex digit was truncated.

Actually, they never keep more than 15 digits while multiplying. (In the larger model 360's, they do this chopping -- they generate the whole product and throw a big chunk away.)

In division, truncation is done properly.

Overflow/underflow: In exponent overflow, the exponent is small by 128, the fraction is correct.

In exponent underflow, the exponent is large by 128 and the fraction is correct.

Division by zero is suppressed.

The 360 has a strange thing -- over/underflow can cause an interrupt, but it can run with the interrupt turned off. Then a condition code is set, except for multiply and divide. You can get around all this by letting the interrupt work and code to prevent it.

In FORTRAN there is not much hope of recovering from overflow. IBM says use PL/1 to find or go around your error.

No information from the exponent is lost on overflow because it can only overflow by 1 bit. The number range (normalized) is not so asymmetric as on the B5500.†

---

†As an exercise, verify that if the B5500 exponent were biased differently, you'd not lose that extra bit. You do not lose a bit on the 360.

Rules satisfied:

1. Have x, -x
2. There is a -0, but it acts like the normal zero in compare operations.
3. Exactness of answers is preserved.
4. One can say they don't waste information unnecessarily, but they make it hard to recover.

Question: What about this truncating business?

Answer: In a sense that is not wasted because you had nowhere to put it.

5. The answer won't necessarily be set to the nearest representable number because of truncation, but it is one of two numbers on either side.

The error is less than 16^{-5} for single precision. You see this by looking at

$$.100000|FFF\cdots F$$

If you throw away that (those) F's, the relative error is

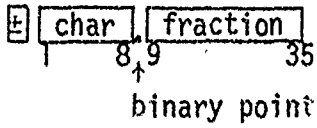
$$\frac{16^{-6}}{16^{-1}} = 16^{-5} \text{ in the last place (almost)}$$

6. Sign symmetry is preserved. You do get drift because of truncation.

Double precision is just like single precision (they carry a guard digit) except for 14 instead of 6 hex digits.

<u>IBM 7094</u>

floating point representation



binary machine

characteristic biased by 128, $-128 \leq exp \leq 127$

27 bits in fraction

Arithmetic is done in an accumulator

```
     Q P
±[ ][ ][ char ][ fraction ]    Q, P  overflow bits
0    1     8 9          35
```

Another register which sometimes acts like a right hand extension of the accumulator is called the MQ.

```
±[ char ][ fraction ]
0 1    8 9         35
```

Integers use all 35 bits and have their own operations because they are not set up like floating point numbers with zero exponent.

Add or Subtract operation -- different exponents. It puts the number with the smaller exponent into the accumulator. Then it right shifts it into the MQ.

```
  [.1x........]   memory
± [0000.xxxxxxx] [xxxx0000]   AC and MQ
```

It performs + or -, puts the answer into the accumulator and normalizes, then doesn't round the answer, although the information is sitting in the MQ. In normalizing, it brings in bits from the MQ. So you have 54 bits when you add or subtract. There is a round instruction; it examines the highest bit of the MQ and rounds up if it is a 1. You could use Professor Kahan's rounding scheme here by examining other bits of MQ, but this isn't done.

For multiply, you have the 54 bit result in AC and MQ, but you only get the truncated (or rounded) result, as in adding.

In division, the quotient is in the MQ and the remainder is in the AC. Correct rounding here would require another division to determine the ratio of remainder to divisor. So division is simply truncated.

Question: How well does single precision follow the rule that says get the exact answer and truncate to so many significant figures?

Answer: This is followed except in one case because if a result must be left shifted, bits are brought in from the MQ. The one exception is when an add or subtract shift sends the smaller number out of the MQ. Then the rule is not followed as the larger operand is the result.

This case could lead to some sequences to not be monotonic that should be.

Double Precision

AC has high order bits, MQ has low order bits -- two words in memory have the other double precision word.

Add and Subtract

If you have to right shift one operand, shifted bits are simply lost. So there are no guard digits.

Multiply

Say words are A and B (in AC and MQ) and C and D (in memory). The leading digits of the answer come from $[A \times C]_{\text{high order}}$. Lower order digits come from $[A \times C]_{\text{low order}} + [A \times D]_{\text{high}} + [B \times C]_{\text{high}}$. This can lead to an error of 6 units in the last place. The machine truncates on taking $[A \times D]_{\text{high}} + [B \times C]_{\text{high}}$. So it could lose -2 in the last place. It will lose another 1 (down) in the last place because you ignored $[B \times D]_{\text{high}}$.

Then if the entire answer needs to be left shifted to normalize the result, these -3 would become -6 units in the last place. An instance of an error of 5.94 ulps was discovered at the Jet Propulsion Laboratory.

Double precision divide can lead to -4 units error in the last place.

Question: If you subtract two different double precision words, can you guarantee that the result is non-zero? Except for underflow.

Answer: Yes, although it's not easy to find out.

Question: What if one number is zero and the other gets shifted way to the right?

Answer: It can't happen as the machine would detect the zero and give the other number as the result.

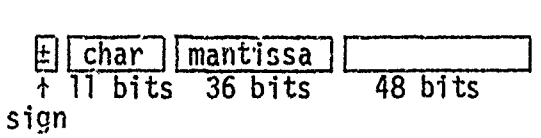
Question: Is any information lost on over/underflow?

Answer: No, registers are not cleared on over/underflow. Also, you get an interrupt and are given the following information: whether overflow or underflow occurred, in which register this happened, AC or MQ, what operation was being performed, +, -, *, /; the address +1 is stored.

You can do arithmetic with P and/or Q nonzero where they act as part of the characteristic. But the manual warns that this may give wrong results.

You can get drift because of truncation.

BCC Model 1



Double precision adds 48 bits to the mantissa

The exponent is biased by 2000_8; the mantissa is in 2's complement. The exponent is not changed for negative numbers. This is true for both

normalized and unnormalized numbers.

4 Different Kinds of Floating Point Numbers

normal zero:	0·····0		
normalized numbers:	+ 0 x·····x 1.xx·····		
	- 1 x·····x	or 1.0·····0	both normalized
		0.x·····x	negative number
unnormalized numbers:	+ 0 0·····0 0.x·····x		
(smallest exponent)	" 1.x·····x	(sort of normalized)	
	- 1 0·····0 x.x·····x		
$-\infty$	1 1·····1 0·····0		

Hardware for machines exists, but not all of the microcoding for floating point and none of it for double precision or for handling $-\infty$ exists.

Floating point arithmetic was essentially implemented in microcode from the manual.

Bias was put in the exponent to allow the zero test to be either fixed point or floating point. This actually is unnecessary because there are floating point tests.

Any other number of the form

0 ≠ 0 0·····0

is not a legal floating point number. If you try to use it as a floating point number, you get trapped.

Arithmetic

All arithmetic is done in double precision. The accumulator is 84 bits.

The result is rounded to 36 bits only when you do a store operation in single precision mode.

Add and Subtract are done with a round bit and a sticky bit.

Question: Aren't there two rounding bits?

Answer: No, there's just one rounding bit. You can get around this (for subtract where two bits at the right may be needed) by shifting left 1 at the beginning and using the overflow bit.

<u>Example</u> where double precision arithmetic and then rounding don't work properly (according to the manual):

```
          36          48      R S
[0][1.xx·····0][10······ ·0][1][0]
```

Round the double precision word to the nearest even so R and S are just thrown away. When you try to store this word in single precision, the machine looks only at the 48 bits and rounds to the nearest even in this case (so 48 bits are just thrown away) and your answer is off in the last place. This is wrong in the manual. It uses the 48 bits and the R and S bits to round.

Multiply acts like you expect it to, using the R and S bits.

Divide in single precision (claimed):

```
      37     S
[quotient][ ]
            ↑
            set if divide is not exact,
            set if there are more fraction
            bits in the accumulator
```

There is an anomaly in double precision divide. You can divide 84 bits of accumulator by 84 bits in storage to give an 85 bit quotient. You need one more bit to be set if the division is not exact.

The manual does not treat unnormalized numbers. It just says that they exist.

You can generate them on underflow and choose to trap on underflow, or choose an unnormalized result.

If you trap on over/underflow, the mantissa is correct and the exponent is off by 2^{11} (overflow) or -2^{11} (underflow).

There are five different rounding and double precision modes. You can select a different mode for one operation and then the program reverts to the standard mode (discussed earlier).

Discussion Comments

1) There should be no discernible difference in the way single and double precision rounding are done. Thus, division is a mistake in the design. One bit is left out.

This allows a user to discover if his program is not working because of a flaw in the program or because of rounding errors in the machine. He changes his single precision program to double precision and looks to see if his errors have moved to the right or not.

2) The extra rounding modes give users access to interval arithmetic by allowing users to specify round to the next larger or round to the next smaller in value or magnitude as wanted [12].

3) Default rounding is to the nearest number.

But rounding can take as long as an ordinary add. So do your rounding when you use an operand and not when you generate it. Then you can overlap operations in the machine and not waste time.

On the 7094 for example the Round instruction takes 2 cycles, ordinary Add takes 3, so Add and Round take 5 cycles.

Question: On the 7094, is it possible to get the same result in two different ways (multiplying) because the binary point is to the right of the first digit? On overflow, that is.

Answer: No, you can only get one overflow from mantissas because their product, no matter how big they are, is less than 4. Say $m = 1.1\cdots1$ then $m^2 < 4$.

4) Normalized and unnormalized numbers: Normalized numbers all have reciprocals. But the number $\boxed{0}\boxed{0\cdots0}\boxed{1.0\cdots0}$ does not (it is the only one that doesn't), so it is unnormalized. Actually, it is called both.

5) Why is there a $-\infty$? ∞ isn't dealt with; it is just used to tell that overflow has occurred.

In interval arithmetic, you think of numbers as being on a circle and every interval that is representable is an interval on that circle. Its complement is also representable, so that includes only one point as ∞ [12].

This machine could have been ideal, depending on if the details were implemented conscientiously.

Question: Why are numbers in complement form instead of in sign magnitude?

Answer: Because they said it was easiest to run the registers that way and besides it doesn't matter.

Question: Doesn't the double rounding cause problems?

Answer: It could, if you did (A+B)*C without storing and if you stored (A+B), then fetched and multiplied by C. They get around this by always having A+B stored temporarily (by compiler) with store and fetch operations overlapped by others, so no time is lost in doing this.

When a number is rounded, right hand bits are cleared. But a double precision word ought not to be rounded until it is stored, so somebody missed the point.

Question: Is it economical to have double precision hardware, or is it a luxury?

Answer: Once it was decided that double precision was a good thing, you only pay a small penalty by doing all operations to double precision, namely a small time penalty for carries to propagate.

Question: But why do most machines not have built-in double precision? Why was it done in the BCC?

Answer: Usually, you can program double precision almost as well as it can be handled by hardware. But in machines of small characteristics, you run into serious problems with underflow and overflow. The characteristic of the second word is down from the first, so there is a nasty tendency to underflow and the system may be cleared to zero. But you might say the problems get worse with higher precision.

That's true, but most people are content with double precision. Those who want more are willing to sacrifice efficiency.

CDC 6400

floating point number range: $3 \times 10^{-293} \leq f \leq 2 \times 10^{321}$
rather large compared to other machines

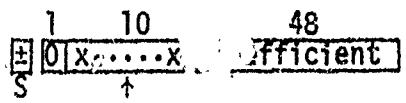
numbers represented as: $\text{sign} \times 2^{\text{exp}} \times \text{coefficient}$

coefficient is considered as an integer

$-1022 \leq \text{exp} \leq 1022$

$0 \leq \text{coef} \leq 2^{48} - 1$ unnormalized

$2^{47} \leq \text{coef} \leq 2^{48} - 1$ normalized



exp is 11-bit 1's complement number
then complement the first bit to bias the exp

If the number is to be negative, the whole word is 1's complemented.

The word is packed so complicatedly in order that you could take the floating point representations and compare them using fixed point compare, as long as the numbers are normalized and none are indefinite. But you can't do this by using fixed point subtract (it's not really faster anyway).

Floating point operations

Add Subtract Multiply	truncate, round, double precision
Divide	truncate, round
Normalize	normalize and round

We will principally discuss single precision, normalized numbers. Since 1's complement numbers are isomorphic to sign magnitude numbers, we will only talk about magnitudes.

Question: What does a normal zero look like?

Answer: It depends on who wants to know and we'll go into that in some detail later. The best answer is that anything that is fed to the normalizer that it thinks is zero is cleared to all zeros.

The RUN version of FORTRAN makes sure everything is normalized. It must normalize after add and subtract.

Addition

```
 [1x··· ··· ··· ·]
 [0····01xx·····x000]
 ------------------------
 [ ][              ]
  ↑
 overflow bit
```

The smaller number is put into a 96 bit register and right-shifted. Bits fall off the right end.

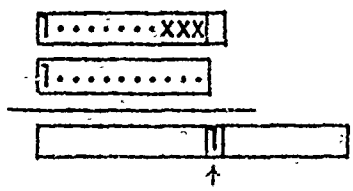
In truncated add, right bits are dropped. If the sum overflows, one

right shift is done.

Actually the leading 48 bits in the register are taken as the result, not the leading 48 significant bits.

Subtraction

Same sort of thing. Exact subtraction of magnitudes, truncated to leading 48 bits of the register. The answer may be very small.



exponents differ by 1

This may be the only significant bit and it would be thrown away.

error in $A \oplus B = A(1+e_1) + B(1+e_2)$ $\quad |e_i| \leq 2^{-47}$

error in $A \ominus B = A(1+e_1) + B(1+e_2)$ $\quad |e_i| \leq 2^{-47}$

Question: Why is that register 96 bits?

Answer: It is used in multiplication and double precision operations. You can get at the right hand part.

Question: Why must you have e_1 maybe different from e_2? Is it because you can get a zero answer undeservedly?

Answer: No, for example, on the 650 where right shifted digits are lost immediately (and you don't get zero undeservedly), the same error analysis would have to be used. Non-zero answers can still have a high relative error.

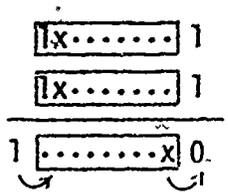
On the 650 (to two digits), 100 - 99 = 10. The e_i's could be made equal, but they'd be huge and you wouldn't want to use them.

Here you violate the rule that says if the answer could be represented exactly in the machine, it should be.

Rounding Add

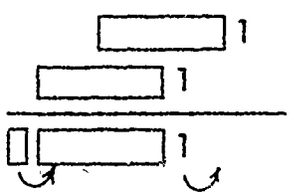
Add a 1 bit at the end of each normalized operand (try to normalize the operands), unless you have a 0. (Zero is not normalized as far as this operation is concerned.)

case of equal exponents



This would add $\frac{1}{2}$ in the last place.

case of unequal exponents



This could still overflow. Then adding only $\frac{1}{4}$ in the last place makes an error of $\frac{3}{4}$ in the last place.

$$A \oplus B = A(1+\epsilon_1) + B(1+\epsilon_2)$$

$$\text{no overflow} \Rightarrow |\epsilon_i| \leq 2^{-48}$$

$$\text{overflow} \Rightarrow |\epsilon_i| \leq \frac{3}{4} \times 2^{-47}$$

This doesn't lead to so pronounced a drift as in truncation.

Multiplication

It forms an exact 96-bit product (may have only 95 significant bits).

If it has a leading zero, the result is normalized while still in the 96-bit register, then truncated to the 48 high-order bits.

$$A \otimes B = A \times B(1+\epsilon) \ , \quad |\epsilon| \leq 2^{-47}$$

Rounded multiply

When the machine was first built they would add a 1 to the end of one operand before multiplying, so that you could have $A \times B \neq B \times A$.

Now they add a 1 in the 50th bit instead of 49th, then left shift 1 if necessary.

So they round by $\frac{1}{4}$ or $\frac{1}{2}$, depending on if there is no or 1 left shift.

Actually this adding is done at the beginning of the multiply operation (done by add, shift, add, shift, etc.). Instead of adding to zero in the first cycle, they add to

$$\boxed{010\cdots\cdots0}$$

$$A \otimes B = A \times B(1+\epsilon) \ , \quad |\epsilon| \leq \frac{3}{4} \times 2^{-47}$$

Not much better than truncated multiply.

Division

It truncates the exact answer to 48 bits.

$$A \oslash B = A/B(1+e) \ , \quad |e| \leq 2^{-47}$$

Rounded Division

This appends to the numerator the series 010101··· so they compute $\frac{N+\frac{1}{3}(1-2^{-48})}{D}$ and take the most significant 48 bits.

$$A \oslash B = A/B(1+\epsilon) \ , \quad |e| \leq \frac{2}{3} \times 2^{-47}$$

This assumes that the operands are uniformly distributed between 2^{+47} and $2^{+48}-1$ and that the part thrown away is also uniformly distributed. Then you get that the mean error, after rounding, is zero (by hocus-pocus).

Question: Are the operands uniformly distributed?

Answer: I have no idea, but there may be a tendency for smaller numbers to appear.

Who Wants to Know if It Is Zero?

Who Wants To Know ↑	E ≠ 0, C ≠ 0	E 0, C ≠ 0	E ≠ 0, C 0	E 0, C 0
compare	N	N	N	Y
±	N	N	Y	Y
* /	N	Y	Y	Y

Is It Zero? ⟶

E = exponent

E = 0 means smallest possible exponent

C = coefficient

Column 3 disappears when normalized numbers are demanded. If the coefficient is zero, the whole thing is set to zero by the normalize instruction.

In column 2, it will be set to zero unless it is already normalized. That is, if the normalize box has to shift left, it can't because the exponent is already as small as it could possibly be, so all zeros are entered.

The add box would be happy to add in a number of the 2nd column type.

There is a strange number on the CDC that is treated differently by different units.

0·····0	1X·····X

It is normalized since the leading bit is a 1.

As far as the add unit is concerned, this number is legal. But the

multiplier looks at the zero exponent and says the number is zero. Thus you can get

$$A*1. = 0 \quad \text{when} \quad A \neq 0 \quad .$$

If you divide by this number, you get an indefinite answer.

An example of two different numbers whose difference is 0:

```
X:   [10·····0]
Y: - [111·····111]   exponents differ by 1
     ----------
     [0·····001]
```

When this answer is truncated, you lose 100% of the answer. So if you test for X = Y, the result is TRUE, according to the machine.

Why is this so bad, since the numbers differ by only 1 in the last place? Because you are losing all of your answer. This could cause a problem in the following way:

Say $X = 1.0$, $Y = 1.0 - 2^{-50}$ are FORTRAN variables. You test for $X = Y$ and get TRUE, implying that $X = Y$. If you had tested for $(X - .5) = (Y - .5)$, you would get FALSE, implying $X \neq Y$.

Overflow and Underflow Peculiarities

If you overflow, there is no trap, but a certain bit pattern is produced, [3777xxx], called ∞. When you try to use this number, you are trapped. There is a test for this number, but you would have to do it after every multiply and divide.

On an overflow, the coefficient would be correct, but there's no way to get at it. And the exponent is put to 3777 (it is not correct modulo anything).

On underflow, the result is cleared to zero and there is no message. Thus, things like the following can happen:

$$\frac{Ax+B}{Cx+D} = 1.0 \qquad \frac{A+B/x}{C+D/x} = \frac{2}{3} \quad ,$$

A, B, C, D, $x > 0$ and normalized.

In one case, the numerator and denominator underflow, in the other nothing happens. The point is that on the CDC you have no way of knowing if there was underflow.

II. THE RUNW.2 COMPILER FOR CDC FORTRAN†

Introduction

This paper is a description of another revised FORTRAN IV compiler derived from the CDC RUN compiler. The modifications were performed by the author on the RUNW.1 compiler, which is in turn a modification of the University of Washington RUN compiler, November 1970 version. Basically, RUNW.1 is a modification of the CDC RUN compiler which produces somewhat more efficient code, largely through improved use of temporary space.

The purpose of the new revisions was to "fix up" real single precision arithmetic. This has been done by modifying some in-line functions and one library function, changing the order of evaluation of relational expressions (such as X .GT. Y+Z) and, as a user option at the subroutine level, to provide properly rounded single precision real arithmetic instead of the somewhat undesirable arithmetic currently compiled.

This paper is a description of the new revisions. It is divided into four sections: Section I is a theoretical discussion of rounded arithmetic on CDC 6000 machines and numerical analysis, and is meant to describe and explain the defects in the CDC RUN code and provide solutions. The improvements made in the compiled code are discussed.

Section II is a description of the options available to RUNW.2 users, and is oriented toward the somewhat sophisticated user. Some examples of COMPASS code are given, but they are mostly for completeness; it is not essential that the reader understand COMPASS in order to use Section II.

Section III is devoted to compiler internals and is not reprinted here.

Section IV gives some wishes for the future.

---

†By David S. Lindsay.

Most of the new floating point algorithms generated by RUNW.2 were suggested by Professor W. Kahan, who supervised the compiler modifications. We state at the outset that our purpose is to implement correctly rounded arithmetic. Over/underflow problems still exist and there is just no sensible economically feasible solution on these machines.

Section I: Rounded Arithmetic on CDC 6000 Machines

In order to understand the options available on a 6000 series machine, it is necessary to be familiar with the floating point hardware, as described in the machine reference manual, CDC publication #60100000.

We first consider addition and subtraction, which may make use of the F, R, or D type add and subtract and the N and Z (normalize, and round and normalize) instructions. However, the Z instruction does not appear to be useful in this context.

We will adopt the convention, used by the RUN compiler, that operands are assumed to be normalized and, if they are, then results are guaranteed to be normalized also.

The "obvious" way to perform an add of X1 and X2 into X6 is:

```
FX6   X1+X2   FLOATING ADD
```

However, normalization in the 98 bit accumulator does not occur. Its upper 48 bits are simply packed with the appropriate exponent into X6. Thus, as the reference manual points out, the result may not be normalized. Hence, if we are to adhere to our convention of guaranteeing normalized results, a normalize is required. As we shall see, the lack of an "automatic" normalize causes most of the problems associated with addition and subtraction.

We thus arrive at the following code:

```
FX6   X1+X2   FLOATING ADD
NX6   X6      NORMALIZE
```

Similarly, a complete floating subtract would look like:

```
FX6   X1-X2   FLOATING SUBTRACT
NX6   X6      NORMALIZE THE RESULT
```

In fact, the RUN compiler compiles all of its single precision real adds and subtracts in this way (although, of course, not necessarily with the registers we used).

This arithmetic has an ugly feature. It is possible for two normalized real numbers Y and Z to be such that Y-Z computed in this way yields zero, but (Y - 1.) - (Z - 1.) does not!

For example, let

Y = 1721 4000 0000 0000 0000 B value = exactly 2.0
Z = 1720 7777 7777 7777 7777 B value = $2.0 - 2^{-47}$ exactly

Now Y-Z, following the recipe in the reference manual, would be computed as follows:

Put the number with the smaller exponent (Z) into the 98 bit accumulator, and right shift it by the difference between the exponents (1). Then perform the indicated operation (-), yielding:

```
          upper                  lower
- 3777 7777 7777 7777 / 4000 0000 0000 0000  -Z
+ 4000 0000 0000 0000 / 0000 0000 0000 0000   Y        (II.1)
  0000 0000 0000 0000 / 4000 0000 0000 0000  result
```

where the / marks the division between the upper and lower 48 bits of the accumulator.

Thus after the F subtract, the result register will contain:

1720 0000 0000 0000 0000

which, when normalized, is of course zero.

But now consider Y - 1.:

1. = 1720 4000 0000 0000 0000

Thus Y - 1. = 1720 4000 0000 0000 0000 = 1.0

and Z - 1. = 1717 7777 7777 7777 7776

Now compute the difference between these two numbers. The accumulator will then contain

```
  - 3777 7777 7777 7777 / 0000 0000 0000 0000  -(Z-1.)
  + 4000 0000 0000 0000 / 0000 0000 0000 0000   Y-1.     (II.2)
  ----------------------------------------------------
    0000 0000 0000 0001 / 0000 0000 0000 0000  result
```

which is not zero!

We therefore also note that the results obtained from this kind of arithmetic depend not only on the exact answer, but also on the operands.

We would like to present code which does not have these defects. The availability of the lower 48 bits (by use of the D instruction) is the way out.

Consider the following code:

```
1    DX0   X1+X2   LOWER 48 BITS WITH THEIR EXPONENT
2    FX6   X1+X2   UPPER 48 BITS WITH EXPONENT 48 LARGER
3    FX7   X6+X0   ADD (1) AND (2)
```

Let us consider step 3.

The coefficient of the smaller exponent (X0) is entered into the

accumulator and right shifted by the difference between the exponents (48). This puts it entirely in the lower half of the accumulator, which is of course where it came from to begin with in step 1.

The coefficient of the larger exponent is then added. It is of course of the same sign as the coefficient just entered, but lies wholly in the upper 48 bits. Thus the effect is exactly that of concatenating the two coefficients viewed as bit strings. This is all obvious enough. But something interesting occurs if we insert a normalize between 2 and 3, thus:

```
1      DX0   X1+X2   LOWER SUM
2      FX6   X1+X2   UPPER SUM                    (II.3)
2.5    NX6   X6      NORMALIZE UPPER SUM
3      FX7   X6+X0   ?
```

If n left shifts were performed in step 2.5, then the exponent of X6 would be decreased by n (assuming the coefficient $\neq 0$). Thus in step 3, the coefficient of X0 would be right shifted $48-n$ places before placing it in the accumulator. Its leftmost n bits now lie in the upper part; X6's lower n bits were cleared by the normalization, so again the effect is just that of concatenating the bit strings, but <u>the result is now normalized</u>.

The only exception to that statement arises if X6 were originally zero. Then the result of step 3 is exactly the lower part of the original sum, and so it still must be normalized to guarantee a normalized result. (But in fact, it is easy to see that normalization is only really necessary when the answer is zero.)

We now have a method for getting more accurate chopped arithmetic. The only case in which the result obtained is not the chopped representation of the exact result is when

1) The add or subtract results in the algebraic sum of two non-zero numbers of opposite sign, and

2) One is so small in magnitude with respect to the other that it is entirely right shifted out of the accumulator when it is loaded.

The computed result equals the operand with the larger magnitude, but the exact chopped result is smaller (in magnitude) by 1 bit in the last place. (However, this is a lot better than we were doing before.)

Replacing the floating add in step 3 by a rounded add will result in properly rounded arithmetic. The complete algorithm is:

```
1    DX0   X1±X2   DOUBLE ADD/SUBTRACT
2    FX7   X1±X2   FLOATING ADD/SUBTRACT
3    NX7   X7      NORMALIZE UPPER PART                (II.4)
4    RX7   X0+X7   GET ROUNDED RESULT
5    NX6   X7      AND NORMALIZE
```

To see why the rounded add works right, note that the operands in step 4 will be of the same sign, so a round bit will be attached to the right of the larger (and to the right of the smaller if and only if it is normalized). The presence or absence of the round bit on the smaller operand is irrelevant, as a consideration of the cases shows:

Case I. The exponents are the same.

How could this happen? The exponent of the upper part is 48 greater than the exponent of the lower part before normalization (step 3); after normalization, the exponent is decreased by at most 47, unless the coefficient is zero, in which case its exponent becomes -1777B, corresponding to a characteristic of 0000. However, fortunately, the hardware treats this as a special case. It will not append a round bit to a zero operand.

Thus the result of the addition will be exactly X0, as desired.

<u>Case II</u>. The exponent of X0 is greater than that of X7.

The analysis in Case I shows that this can only happen when X7 is zero. But then we get the right answer, namely X0.

<u>Case III</u>. The exponent of X7 is greater than that of X0.

A round bit is then attached to the right of X7 before addition. If X0 is not normalized, no round bit is attached to it; if it is normalized, a round bit is attached. But since X0 is right shifted at least by 1, its round bit cannot generate a carry. Figure II.5 makes this clear. It assumes positive operands; the case of negative operands is similar. Note of course that both X0 and X7 have the same sign.

Figure II.5

```
0xxx ··· xx / ··· x100 ··· 00   X0  right shifted at least 1
1yyy ··· yy / 10000000 ··· 00   X7  normalized
              ↑   x100 ··· 00
              |   ↑
              |   cannot generate a carry
              |
              round bit
```

We now know that the presence or absence of the round bit to X0 has no effect. If $X7 \neq 0$, then it will have a round bit attached. Will this always give the correct rounded result?

Since X7's coefficient will always have a round bit appended, if no overflow out of the accumulator takes place, the round will be $\frac{1}{2}$ in the last place. This is exactly what we want. But what if overflow does occur? There will then be a right shift by 1 to compensate for it, so the round is only by $\frac{1}{4}$. But in fact, the right answer still obtains, as follows:

We noted previously that instruction 3 of II.3 effectively just reproduced the accumulator as it was at the conclusion of the adds (or subtracts)

in 1 and 2, except that it is now normalized. Thus the only time an overflow can take place in 4 of II.4 is when it is caused by the round bit. But that can occur only when the 49 leading bits are all 1 The result is then to clear all of them to zero and set the overflow condition, which will increase the exponent by 1 and produce a coefficient of 4000 0000 0000 0000 B. But that is the correctly rounded result.

We conclude then, that the R add (instruction 4 of II.4) will always produce the correctly rounded sum of its operands. Will that always be the exact answer to the original add (of X1 and X2) rounded to 48 bits? Surely it will be if the exact answer lies wholly within the 98 bit accumulator. But even if it does not, the result will still be correct, as the following argument shows:

Suppose that the exact sum X1+X2 does not lie wholly within <u>96</u> bits. This can only happen if the magnitudes of the operands are so different that there is at least 1 bit separating them when they are placed in the accumulator. Suppose first that they are separated by at least 2 bits:

```
1xx ··· xx / 00000 ··· 00        larger operand
000 ··· 00 / 001yy ··· yy / y    smaller operand partially (or wholly)
                                 shifted off
            ↑
            at least
            2 zeros
```

If the operands are of the same sign, the round bit in step 4 of II.4 has no effect, and the result is exactly equal to the larger operand, which is the correctly rounded result; when the signs are different, we have in absolute value:

```
zzz ··· zz / 11ww ··· ww   result of the subtract
             ↑
             at least 2 ones
```

Even if the result is not normalized, a left shift of 1 must normalize it. Thus the round bit will generate a carry, and rounding will be by 1. This restores the larger of (X1,X2) as the final result, which is of course the correctly rounded result, except when the coefficients are separated by 1 bit:

Example

Suppose the accumulator looks like:

```
  1xx ··· xx / 0000 ··· 00
- 000 ··· 00 / 01yy ··· yy / y
-----------------------------
  zzz ··· zz / 1www ···
```

If the accumlator overflows, the larger coefficient must have 47 trailing zeros. To make the rounding come out wrong, we must have the one lost bit alter the upper part of the DP accumulator. The following example gives a wrong answer:

```
  100 ··· 00 / 0000 ··· 00 /
- 000 ··· 00 / 0100 ··· 00 / 1
-----------------------------
  011 ··· 11 / 1100 ··· 00        computed result
  011 ··· 11 / 1011 ··· 11 / 1    exact result
```

Now normalize and round the computed result:

```
  111 ··· 10 / 00 ··· 0
  000 ··· 01 / 10 ··· 0
                 1          round bit
-----------------------
 1000 ··· 00 / 00 ··· 0     computed result
```

But if we normalize and round the exact answer:

```
 11 ··· 10 / 00 ··· 0
 00 ··· 01 / 01 ··· 1
            1          round bit
 11 ··· 11 / 11 ··· 1   exact bit, rounded
```

Thus the computed answer is not correct.

We have thus concluded that the scheme in II.4 will almost always yield the exact answer rounded to 48 bits, provided that at each step the operands and results were in range of the floating point hardware.

This means that our scheme now guarantees that the result is dependent only on the exact answer and not on the operands almost always. This almost satisfies one of the conditions set forth in Knuth, Vol. II.

We further note that it is no longer possible to have Y and Z such that Y-Z is computed to be zero, but (Y-X) - (Z-X) for any X, is not, barring underflow problems. For if Y-Z is computed as zero, then either Y = Z exactly or Y-Z underflows. This is true because we compute the most significant 48 bits of the exact difference. If these are zero, then the difference is zero. But since Y = Z, we conclude Y-X = Z-X for all X (once again, barring underflow).

Note that if the exact result of an addition or subtraction is half way between two adjacent floating point numbers, the rounding is always up in magnitude. In some special cases, this can cause problems. For example, if we compute X+Y-Y+Y-Y+··· with X = 1.0 and Y = 2.**-48, then each time Y is added, it will add 1 bit, while each time it is subtracted, it will have no effect. The computed result will then drift upward, while the exact result merely oscillates about 1.0. To avoid such problems, we would actually like the rounding to always be to the nearest even (or odd) number, when the exact result is half way between. The following code will accomplish

this, but it was not felt to be worth putting into the compiler.

```
        DX0   X7±X2    LOWER SUM/DIFFERENCE
        FX7   X1±X2    UPPER SUM/DIFFERENCE
        NX5   X7       NORMALIZE UPPER PART
        RX7   X0+X5    ROUNDED RESULT
        NX6   X7       NEEDED ONLY FOR 0 RESULT
        MX4   1
        DX3   X0+X5    LOOK AT LOWER PART OF ANSWER
        LX4   48       GET 0000 4000 0000 0000 0000                    (II.6)
        UX3   X3       FILL EXP FIELD WITH SIGN BITS
        BX4   X3-X4    SEE IF SPECIAL CONDITION HOLDS
        NZ    X4,DONE  SENSE NOT HALFWAY BETWEEN
        LX7   59       LOWER BIT OF RESULT TO SIGN BIT
        AX7   58       GET 0 IFF RESULT IS EVEN
        ZR    X7,DONE  SENSE EVEN
        FX6   X0+X5    DONT ROUND -- THUS GET EVEN RESULT
DONE    BSS   0        DONT NEED TO NORMALIZE PREVIOUS STEP -- IT CANT BE 0
```

This could be made into a subroutine by anyone who wishes to do that kind of rounding.

Let us now consider the multiply operations available. There are three multiply instructions: types F (floating), D (double), and R (rounded). Assuming normalized operands, the F and R instructions will yield normalized results. This is done by performing an integer multiply of the two 48 bit operands in a 96 bit accumulator, and left shifting the result by 1 if and only if that will normalize the result. In the rounded instruction, the round bit is added <u>before</u> the final left shift, so that the rounding is either by $\frac{1}{4}$ or $\frac{1}{2}$. The rounded result is thus not correctly rounded.

Assume for the moment that we are using floating point numbers which have 5 rather than 48 bits of coefficient. Here is an example of two pairs of operands which yield the same exact product, but different products using

the rounded multiply.

The first pair of operands have coefficients of 18 and 20. Note that they are both normalized; as 5 bit binary numbers, they are:

18 = 10010
20 = 10100

18 * 20 = 360, so the double length accumulator would contain:

```
01011 / 01000 = 360 unnormalized
00000 / 01000   add round bit
-------------
01011 / 10000 = result, unnormalized, after rounding
```

The hardware then left shifts by 1 before packing, thus yielding:

10111 with a suitable exponent

There is a pair of normalized 5 bit numbers which multiply to yield 720 (twice 360). Thus if their exponents are chosen properly (their sum should be 1 less than the sum of the exponents chosen in the above example), then the exact product in each example will be the same. But look what happens in the rounding:

The numbers are 24 and 30; 24 * 30 = 720.

24 = 11000
30 = 11110

```
10110 / 10000   double length product
00000 / 01000   round bit to be added
-------------
10110 / 11000   result
```

Thus the upper part = 10110.

So although the two products are equal (with suitable exponents), their

rounded products are not.

Here is a 48 bit example, which is messier to verify. The reader may show that:

If

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

then

$$X * Y = A * B \quad ,$$

but

$$R_X(A*B) < R_X(X*Y) \quad .$$

The F multiply does not have this problem. It is properly chopped. Furthermore, since 96 bits are enough to hold the exact product of two 48 bit numbers, we can employ the rounded add to provide a properly rounded multiply:

```
DX0   X1*X2   DOUBLE MULTIPLY
FX6   X1*X2   FLOATING MULTIPLY                    (II.7)
RX6   X0+X6   FINAL ROUNDED ADD
```

The result obtained will, as in the case of addition and subtraction, be the properly rounded 48 bit representation of the exact answer. The computed result will thus not depend on the operands, but only on the exact result.

Note that no final normalize is needed, since the F multiply always provides a normalized result, even when the result is zero.

The case of division is much more difficult to do correctly. There are two divide instructions available, F (floating) and R (rounded). The floating

divide gives the exact chopped result, but as always, the rounded operation is not correct. In the R divide, $\frac{1}{3}$ is effectively added to the last bit of the dividend. The divisor is effectively a number between $\frac{1}{2}$ and 1, so the round will be by a number between $\frac{1}{3}$ and $\frac{2}{3}$ in the last place. The rounding is thus dependent on the divisor, and the computed answer is therefore not dependent only on the exact answer. Unfortunately, to compute the properly rounded quotient is a very long process. But the hardware R divide is such that, statistically, the rounding is very close to correct. For those reasons, the compiler was modified to produce R divides in rounded mode rather than the tedious double precision divide. However, we will present here an algorithm which may be coded as a subroutine and called by those who actually wish the correct answer.

The method consists of performing the floating divide, then multiplying back (in double precision) and subtracting (with care), then dividing again to obtain the double precise answer, and finally using it to round the original result. The reader will probably agree that this is not normally worth doing; it reduces the maximum error by $\frac{1}{6}$ bit, but the average error is almost unaffected.

```
FX6   X1/X2   FLOATING DIVIDE
DX0   X6*X2   START MULTIPLYING BACK
FX7   X6*X2   COMPLETE THE MULTIPLY
FX3   X1-X7   BEGIN SUBTRACTING                                  (II.8)
DX4   X1-X7   IN DOUBLE PRECISION
NX3   X3
FX3   X3+X4   OBTAIN EXACT DIFFERENCE OF (DIVIDEND - F MULTIPLY)
FX7   X3-X0   NOW SUBTRACT THE D MULTIPLY
```

The following double subtract (the starred lines) may or may not be necessary. As yet, we have not been able to prove or disprove that it is:

```
DX0   X3-X0   * DOUBLE PART OF SUBTRACT
NX7   X7      NORMALIZE
FX7   X0+X7   * ADD IN DOUBLE PART
NX7   X7      * NORMALIZE IN CASE RESULT IS ZERO
```

We now have the remainder. Note of course that it will fit exactly in 48 bits. We may now divide again, and then use this result (which is the double precision part of the divide) to round the single precision result:

```
FX7   X7/X2   OBTAIN DOUBLE PART OF QUOTIENT
RX6   X6+X7   CORRECTLY ROUNDED QUOTIENT
```

Let us now consider exponentiation (of a real by an integer).

When the compiler sees R**K, where R is a real expression and K is an integer <u>constant</u> between -11 and 1 inclusive, in-line code is compiled to evaluate the result. The only exception is the case of K = -0, for which a function call (to RBAIEX) is made for some mysterious reason.

The in-line code squares the base, then squares that square, then squares that result, etc. At each step, a product is compiled into the answer register (usually X6) if necessary. For example, R**5 would be compiled as:

```
FX7   X1*X1   (ASSUMING R IS IN X1)
FX7   X7*X7   OBTAIN R**4
FX6   X1*X7   OBTAIN R**5
```

If the exponent is negative, the inverse is done last.

The system subroutine RBAIEX, which evaluates such cases when in-line code is not compiled for them, seemed hopelessly inadequate, and was scrapped for rounded mode. It performs its multiplications with R instructions and does the division (in R mode) first. This of course removes the possibility of "spurious over/underflow" which could arise if the divide is not done

until last. But it also means that the two results X and Y below may not be equal:

```
K = 3
X = R**K
Y = R**3
```

For those reasons, when the compiler is in rounded mode (and so the in-line code uses our 3-instruction rounded multiplies of II.6) a different subroutine is called: RBAIEXR (for Real Base Integer EXponent Rounded).

RBAIEXR performs its divides last, and has tests for under/overflows so that it will guarantee to return an answer without a mode error, and will correct for spurious underflows with negative exponents. (Such underflows arise because the smallest floating number is about 2^{96} times larger than the inverse of the largest floating number.) For example, if the correct answer is 10^{+300}, the in-line code first tries to get 10^{-300}, which underflows to zero. Inverting yields ∞. It also guarantees to return infinity and zero of the correct sign, should they be generated. Furthermore, if the real argument is indefinite, it returns exactly this argument to aid in error tracing. However, if the exponent is ± 0, +1.0 is always returned (to agree with in-line code). The old RBAIEX has none of these features.

The in-line code has no traps for infinity or zero, however, so the following could happen:

1) The Fortran program could get a mode 2 or 4 error while performing an exponentiation. If the exponent is negative, this could happen when the correct answer underflows seriously.

2) If the exponent is negative, the Fortran program will produce infinity if the correct answer is within a factor of 2^{96} of machine infinity

(between 10^{294} and 10^{322}, approximately.)

Thus if the user is operating in these ranges, it might be well to force calls to RBAIEXR by writing exponentiation as a real to an integer variable.

No changes were made in chopped mode code generation. It still calls RBAIEX.

A few miscellaneous in-line functions were changed. AINT, the Fortran in-line function to take the floating greatest integer in a floating number, produces the following code under the old RUN:

```
UX6   B7,X1    UNPACK OPERAND
LX6   B7,X6    FORM AN INTEGER                    (II.9)
PX6   X6       START TO FLOAT IT
NX6   X6       NORMALIZE
```

This code has the disastrous defect that if the argument is greater than the largest 48 bit integer (about 10^{14}), it produces garbage. The following trick eliminates this bug:

```
MX0   1
LX0   59       GENERATE UNNORMALIZED ZERO
FX6   X0+X1    ADD IT TO THE ARGUMENT             (II.10)
NX6   X6       NORMALIZE THE RESULT
```

If the argument is small (less than 2^{48}), then it will be right shifted during the add just enough to place its binary point to the right of bit 0. This is because the zero in X0 has an exponent of zero. If, however, the argument is larger than that in absolute value, it will be the zero which is right shifted -- the result will then be exactly equal to the argument. But that seems to be what one would want: The 48 bit representation of a number $\geq 2^{48}$ is the same as the 48 bit representation of its integer part.

AMOD was also changed completely. AMOD(X,Y) used to generate code equivalent to:

```
X - Y*AINT(X/Y)
```

This is <u>not</u> always (or even often) the remainder when X is divided by Y. That is to say, the expression quoted above is of course AMOD -- but the code compiled for it falls far short of accuracy. We completely rewrote the in-line code to use the new AINT algorithm, followed by the remainder computation (as in II.6). The result now generated is <u>exact</u>. Therefore, AMOD can now be used to program multiple-precision divides. Furthermore, if $X/Y \geq 2^{48}$, in which case the old AMOD produced complete garbage, the new AMOD can be iterated as many times as necessary to yield the exact answer.

The floating point comparisons have also been changed. RUN and RUNW compile relational expressions from left to right as one long (almost) equivalent expression. For example

```
X+Y .LT. Z-P
```

is compiled as

```
X + Y - Z + P    ,
```

left-to-right, and then tests are performed on the sign of the result. That gives rise to the possibility of having, for example

```
LOGICAL L1, L2
X = A+B+C
L1 = D .LT. X
L2 = D .LT. A+B+C
```

with L1 ≠ L2. This problem is independent of the rounding problem; it arises from the fact that floating point addition is not associative.

For example, let A be large, B = -A, and C and D at least 2^{49} times smaller than A. Then A+C = A in single precision.

A+B+C will be compiled left to right; B will exactly cancel A leaving zero, and C will be added, leaving C. Thus X = C.

Then L1 = D .LT. C, or .TRUE. if and only if D-C < 0.

But in the compilation of L2, D-A-B-C will be evaluated. Thus D-A will yield -A, then subtracting B will yield 0, and finally subtracting C will yield -C. Thus L2 will bear no relation at all to L1!

To remedy this, it is merely necessary to force compiliation of the left and right sides of a relational symbol separately (using either chopped or rounded arithmetic depending on the compiler's mode) and then subtract them using the appropriate mode of subtract. This is surely what the user wants when he writes down a relation.

This change was implemented in RUNW.2.

Section II: Use of the Compiler's Features

The main new feature is the choice of chopped or rounded mode. This mode is set by a compiler directive between subprograms, and persists until changed by another directive. There are two directives for this purpose: 'CHOP' and 'ROUND'. They are only recognized when the compiler is looking for a subprogram declaration card.

The directives 'ROUND' and 'CHOP' obey the rules for a standard Fortran statement: they must begin in column 7 or later, etc.

The default mode of the RUNW.2 compiler is chopped.

In chopped mode, the modified in-line functions are still available

(discussed below) and the comparisons are still compiled as described at the end of the previous section. However, the standard real arithmetic using only F instructions is compiled. This mode is designed for compatibility with programs compiled under RUN or RUNW (it will give the same wrong answers) or to provide faster and smaller programs for those who do not need (or do not think that they need) good rounded arithmetic.

Under rounded mode, additions, subtractions, multiplications, and divisions are compiled in rounded form, as described in the previous section. Also, a different subroutine for evaluation of (real)**(integer expression) is called.

To make the use of floating constants compatible with rounded mode, the compiler will evaluate combinations of floating constants using the rounded code previously described, in rounded mode only. In chopped mode, it uses the same code that would be compiled in that mode. This is what RUN originally did, but RUNW messed things up by inserting R type multiplies and divides (but not adds or subtracts!). Thus under RUNW, it might be possible to have Y and X come out differently in something like:

```
Y = 100.*.77
L = 100
X = L*.77
```

This cannot happen in either mode under RUNW.2.

Apart from the introduction of rounded mode, there are some modifications and additions to in-line functions. The most obvious need is for a function to return the rounded single precision value of a double precision argument. Such a function, called RND, is available in either mode, and results in the rounded addition:

RX6 X.U+X.L RND FUNCTION

(where X.U and X.L hold the upper and lower parts of a double precision argument).

SNG., yielding the unrounded upper part of a double precision argument has been implemented in-line (before it was a system function, which seemed silly). It compiles perhaps a Boolean to move the operand to a different X register. However if the operand is an expression, the final DX7 X0+X1 which RUN is fond of compiling in double precision is suppressed. In this case, the in-line function actually removes some code!

AINT has been changed to produce good answers for all operands not indefinite or infinite. Before it produced garbage for operands $\geq 2^{48}$ in magnitude.

AMOD, which is supposed to return the remainder upon division of the first operand by the second,† has been changed to produce an exact answer. If the quotient were greater than 2^{48} before, the answer was garbage. If the quotient is that big now, the remainder returned is the exact chopped result. In fact, it can be AMOD'ed again to produce the desired answer. Thus AMOD can be used to program multiple-precision divides.

A few other functions have been optimized, and the double precision in-line functions have been corrected (all of them were wrong, both in RUN and RUNW, but the bugs were different).

Section IV: Wishes for the Future

The comparisons are still somewhat unsatisfactory, since they cannot compare against infinity without producing a mode error. Infinity should be

†The precise definition of AMOD(X,Y) is: remainder when X is divided by Y to produce only those quotient digits left of the binary point; or, if there are more than 48 of them, then only the leftmost 48.

bigger than everything, and -infinity smaller; also, if the machine is running in a mode to ignore indefinite operands, any test against indefinite should fail. The following code would do this, with appropriate coding at 'FAIL' (perhaps a floating add, to abort if the machine is not in a suitable mode, followed by the production of .FALSE.).

```
        IXO   X.L-X.R   INTEGER COMPARE
        ZR    XO,EQUAL  SENSE EQUAL OPERANDS
        ID    X.L,FAIL  SENSE LEFT OPERAND INDEFINITE
        ID    X.R,FAIL  SENSE RIGHT OPERAND INDEFINITE
        BX7   X.L-X.R   ARE THE SIGN BITS DIFFERENT
* HERE WE ARE GUARDING AGAINST INTEGER OVERFLOW
        PL    X7,NOVFL  SENSE NO OVERFLOW POSSIBLE
        BXO   X.L       IF SIGNS ARE DIFFERENT, X.L TO XO
NOVFL   PL    XO,GREAT  SENSE LEFT OPERAND GREATER
        EQ    LESS      SENSE LEFT OPERAND SMALLER
```

Note that, since the comparisons are exact, this coding would only be appropriate in rounded mode.

There are various other shortcomings in RUNW.2.

Relational expressions use only real arithmetic even when comparing double or complex variables.

A case could be made that complex arithmetic should also be done in rounded mode, so that a user would get the same answer if he either used real variables or complex variables with zero imaginary part. Currently, rounded arithmetic affects only single precision reals.

We believe that in coding RBAIEXR, we set a good example by always returning the correct sign of 0 or infinity and guaranteeing not to abort. It would be nice if other arithmetic functions did this also. Since the Cal loader presets core to -indefinite + address of self, it would be useful in

error tracing to have system routines return a copy of their input argument if it is indefinite, rather than just a standard indefinite. Our RBAIEXR also does this (unless the exponent is ±0, in which case it returns 1.0 to agree with in-line code generation).

Invisible system subroutines should have non-Fortran names. Currently, anyone writing his own RBAIEX, or many other names, will mess things up completely without knowing why. But the cure is not just to change all the names at the end of the compiler -- it has a routine somewhere (I do not know where) that removes non-standard symbols from external names. Unless this routine is also changed, it will just turn strange names into Fortran names.