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

---

[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.

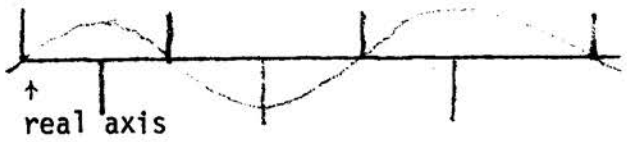
The eigenvalue algorithm is described in a Stanford Report ("Accurate Eigenvalues of a Tri-Diagonal Matrix," Stanford Computer Science Department Report #CS41 (1966)) and in Kahan's Notes on Error Analysis (1968) for the University of Michigan Summer School. The input is a real symmetric tri-diagonal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

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

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.

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

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 $u_{n-1} = \epsilon$ 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 b²ᵢ₋₁)
      :
      U0 = 1.0
      ν = 0
      DO 3 I=1,N
           UI = (A(I)-BB(I)/UI-1)-X
           IF (UI) 2,1,3
1          UI = -ETA                      (if ui = 0)
2          ν = ν+1                        (if ui ≤ 0)
3     CONTINUE
```

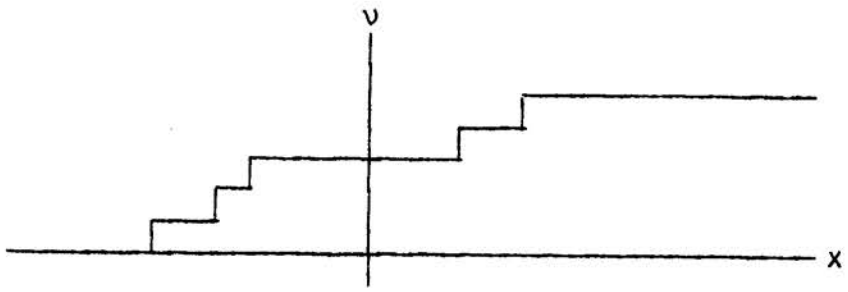
(The subscripts don't actually need to appear in the program.)

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.

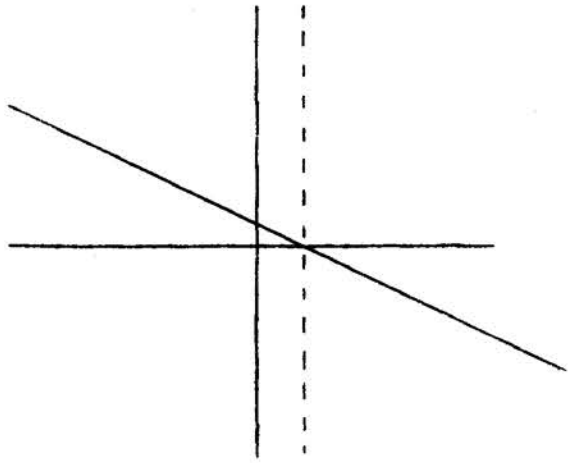
To see this we must plot u as a function of x. We know

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

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

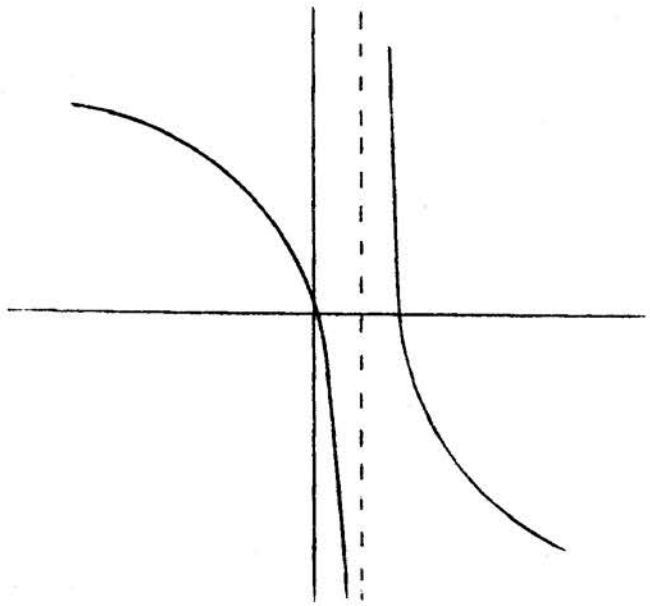
$$u_N = \frac{\det(J_N - x)}{\det(J_{N-1} - x)} .$$

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

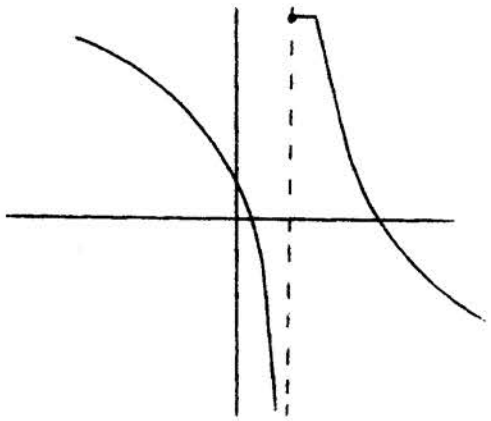


This function is monotonic except at its poles, where $u_1 = 0$. A similar statement, proved by induction, is also true of each of the other u's; they are decreasing except for jumps at poles where the previous u had a zero.

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

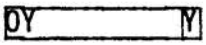


That is, we choose ETA so that no other quotient $\frac{B_{I-1}^2}{u_{I-1}}$ formed by representable numbers in the machine can be larger than B_{I-1}^2/ETA in magnitude. Clearly, then, ETA depends on the machine. On the 6400, for instance, we choose ETA to be the number smallest in magnitude but differing 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 ϵ is in the last multiplication so that

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

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

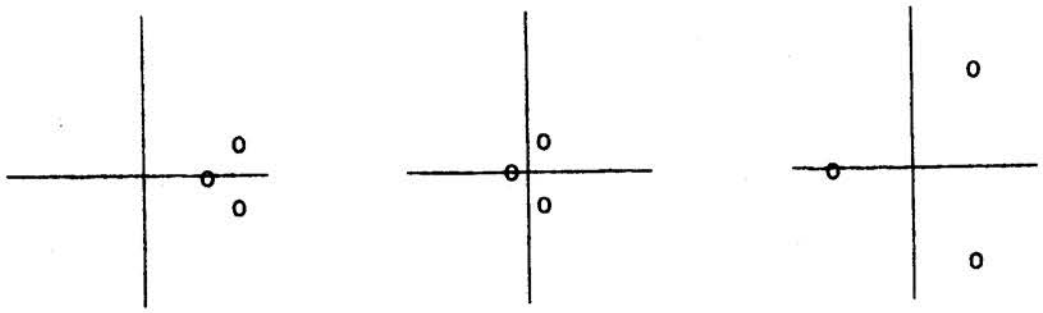
Usual Algorithm

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.



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 $b_1 \neq 0$, 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:

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

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.

	k				ℓ
μ_1	XX	a	b	c	d
μ_2	XX	e	f	g	h
μ_3	XX	i	j	m	n
	μ	0			

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 $\frac{1}{3}k$.

Thesis topic: Discover quickly a satisfactory and rigorous proof of this algorithm.

Examples

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

$\alpha_0 = 353$ $\quad \alpha_1 = -984$ $\quad \alpha_2 = 915$ $\quad \alpha_3 = -284$

$\beta_0 = 353$

$\beta_1' = -655.71$ $\quad \beta_1'' = -327.42$ $\quad \beta_1 = 000.87$

$\beta_2' = 305.1897$ $\quad \beta_2 = 000.6891$

$\beta_3 = -000.173579$

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

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

Then $\mu = .0111$, and

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

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 not 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 never 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

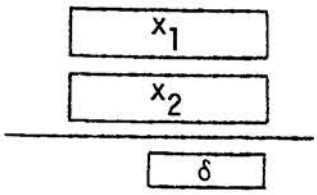
like to know how to design machines which are economical, easy to understand, and have a rich set of nice properties that would allow you to write reasonably efficient programs. Then your programs would run on machines that satisfied these few reasonable rules. It is important to find out what these 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

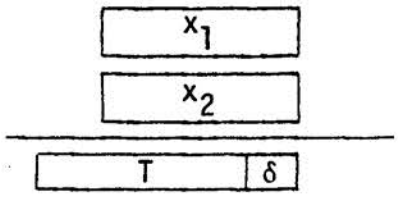
$$\frac{1}{10} \leq 10^k \delta < 1 \quad .$$

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

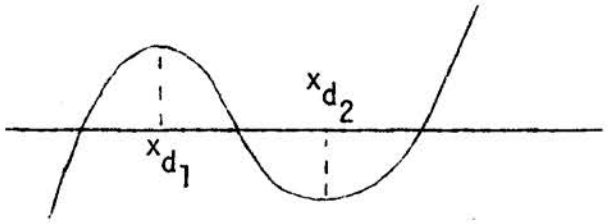
The program first tests to see if the roots are triple or very nearly so. If the roots are exactly triple, there is a relation which holds between the coefficients and the answer is obtained immediately. If the roots are not

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.

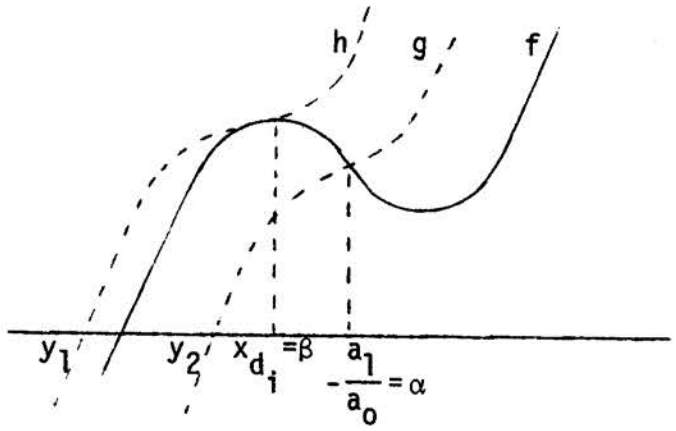
After Roots Are Separated

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

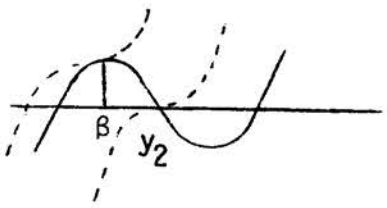
When x_{d_1} and x_{d_2} are real, we construct two functions, one through the point x_{d_i} where the function is largest in absolute value and the other through $-\frac{a_1}{a_0}$, the inflection point where the second derivative vanishes.



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

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.



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

$$x_0 = \frac{y_1 + m y_2}{1+m}$$

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.

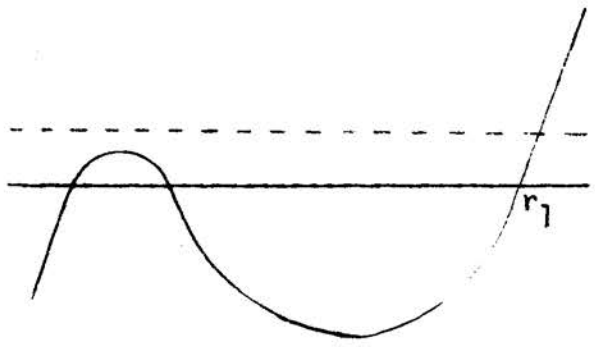
Kahan: This argument is not valid because it depends upon some cancellation occurring that you say you don't want, whereas actually if the cancellation occurred properly you'd be very happy indeed. The issue is that when you compute

$$f(x) = (x-r_1)Q(x) + f(r_1)$$

$$\uparrow$$

quadratic

$f(r_1)$ 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.

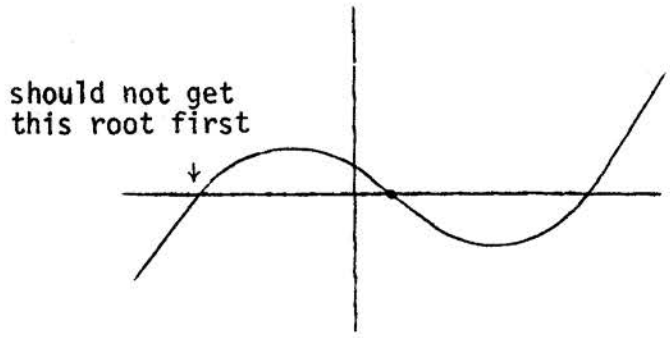
Answer: Yes, but a unit in the last place of a small root is a very small number so you have a small change. But then the other two roots, which are badly blighted by this change, will be blighted by almost any change in the coefficients of an ulp, so they are not well determined. The argument I gave 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, $\epsilon^{1/3}$ does not appear, but rather $\epsilon^{1/2}$ 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.

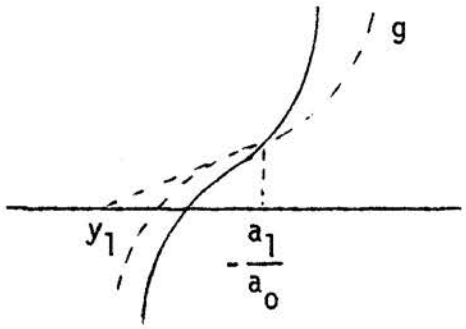


The strategy of the program will give the righthand root (which is the biggest) and then you invert the problem. If you had tried to divide out the factor for the lefthand root, you'd destroy either the largest or the

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:

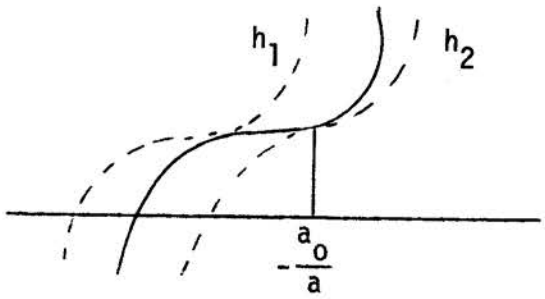


We still compute the function g through $-a_1/a_0$, which guarantees a point to the left (in this picture) of the root and a suitable starting value for the Newton-Raphson method. Another suggestion was to take the derivative at $-a_1/a_0$ and extend the line to the axis and see which point is closer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

The Derivative May Have Close Roots

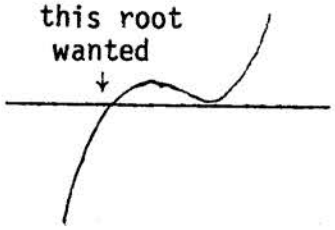
Problems arise when the quadratic has two real roots that are very close. The center of the graph becomes nearly horizontal. If we try to use the point for which the function is larger, we could make the wrong choice. However, the linear combination of points will give us a point to

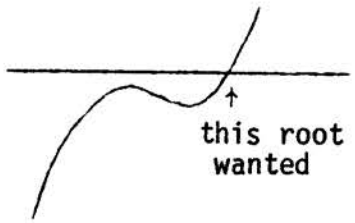
the left of $-a_1/a_0$, even if we made the wrong choice.



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

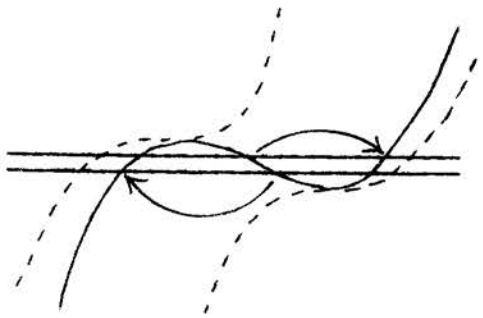
Kahan: The issue is to distinguish these two cases:





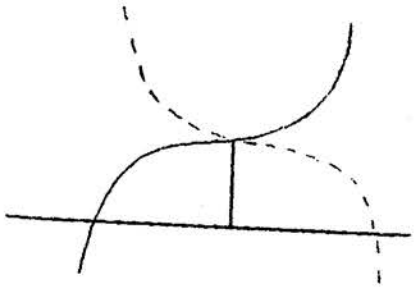
It is easy in the cases drawn here.

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.

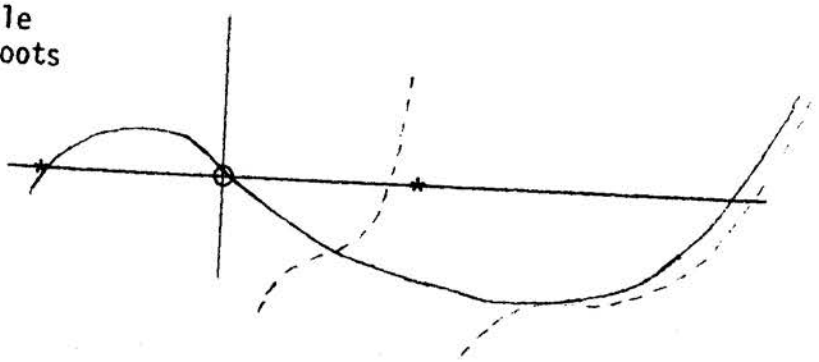


You need a formula such that, if you don't know if h should be to the right or left of the inflection, it will not matter which one you construct. You'll still get a satisfactory approximation to the biggest or the smallest root.

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:

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

or

$$f(x) = 3x - 1$$

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.

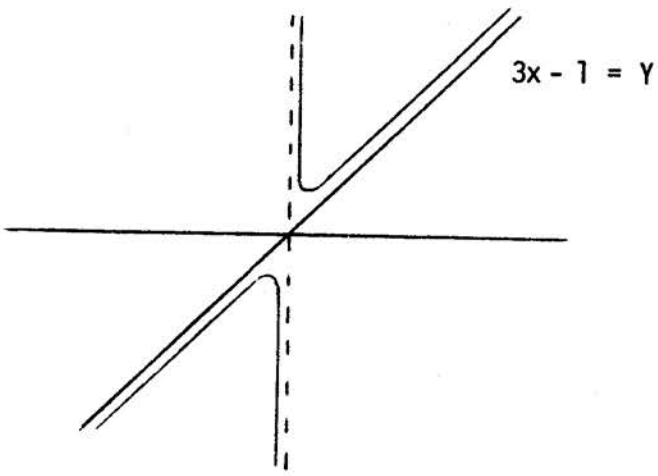
1) You cannot compute the function you'd like to have vanish exactly.

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:

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



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 ϵ ($\approx$ -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 ϵ such that

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

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

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

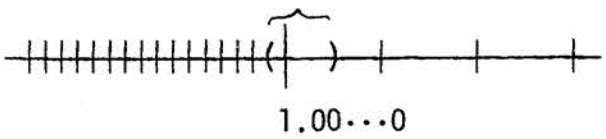
Therefore

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

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

$$F(X) = \&(X,X,\ldots,X) = \&(X_1,X_2,\ldots,X_n)$$

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

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

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 $\&(X,X,\ldots,X)$ is at least as bad as $\&(X(1+\xi_1),X(1+\xi_2),\ldots,X(1+\xi_n))$. In each case, $|\xi_j X|$ < 1 ulp of X or so.

How Does & Vary

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

I'm using the notion that each one of the X's is an independent variable and I can differentiate & 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 $\sum_j \left|\frac{\partial \&}{\partial X_j}\right| |\xi_j X_j|$, because roundoff will include some things we haven't taken into account (the $(1+\eta)$ and truncation error). We can also write this as

$$\sum_j \left|\frac{\partial \&}{\partial X_j}\right| |\xi_j X_j| \leq \sum_j \left|\frac{\partial \&}{\partial X_j}\right| * (1 \text{ ulp of } X) \leq \underset{\substack{\uparrow \\ \text{roundoff error}}}{e(X)}$$

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

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

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

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

$$\&(X) = (\cdots((a_0X_1 + a_1)X_2 + a_2)X_3 + \cdots) + a_n \qquad \text{the expression}$$

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

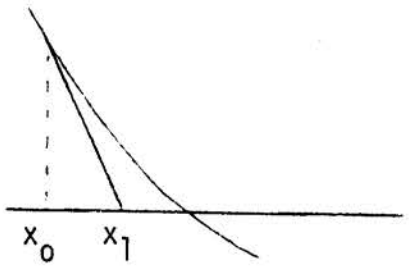
We now have some idea of the sorts of equations we could hope to solve. Now we need to consider what type of method to use to accomplish that solution.

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.

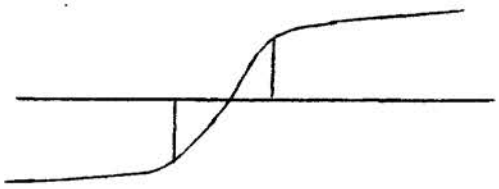


$$x_1 = x_0 - f(x_0)/f'(x_0)$$

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.

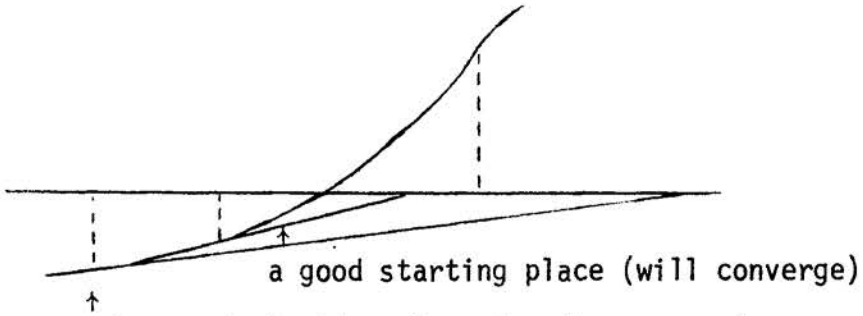
Suppose your function looks like this (like arctan):



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

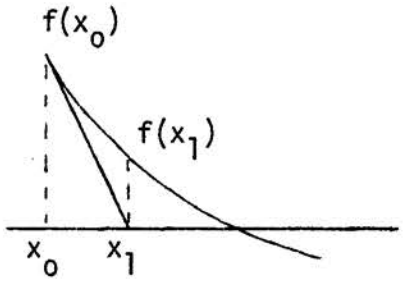
You'd like some sort of theory that tells you that if you use Newton's method in this case, it will always work. That takes some fairly strong global statements about your function, such as if the function is convex in some neighborhood of a root, then anywhere in that neighborhood, you can expect Newton's 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, $f(x_0) > f(x_1)$.

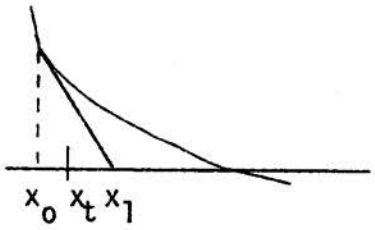


The direction that Newton's method tells you to go is in a sense a downward direction for the magnitude of the function.

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

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



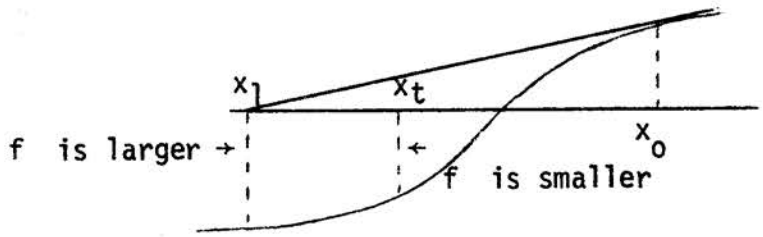
x_t moves in the direction Newton's method points, but not so far

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.†

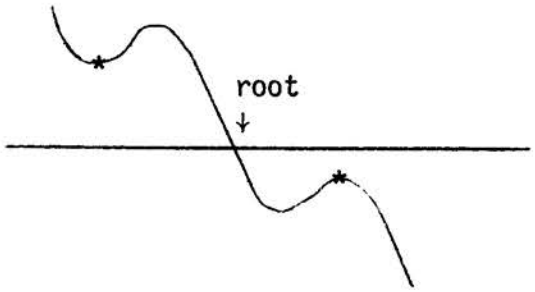
So people have attempted to guarantee, by the selection of t, that the value of f will always decline. That means $t = 1$ may not be such a good choice; t is some fraction that makes $|f|$ decrease. That will greatly improve convergence in the following case:



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.

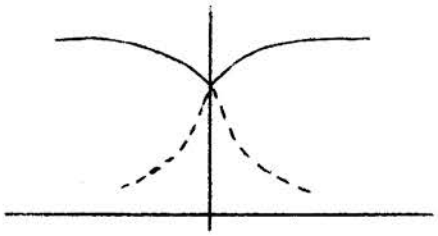
But normally, you do have a way of detecting this situation and then you do something else.

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.



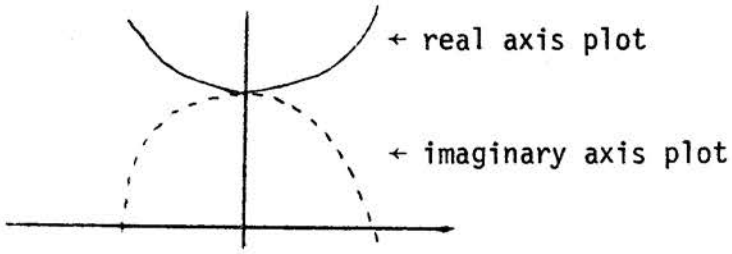
complex plane
$1 + Z^{1/2}$

$1+ Z^{1/2}$ 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.

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



†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.

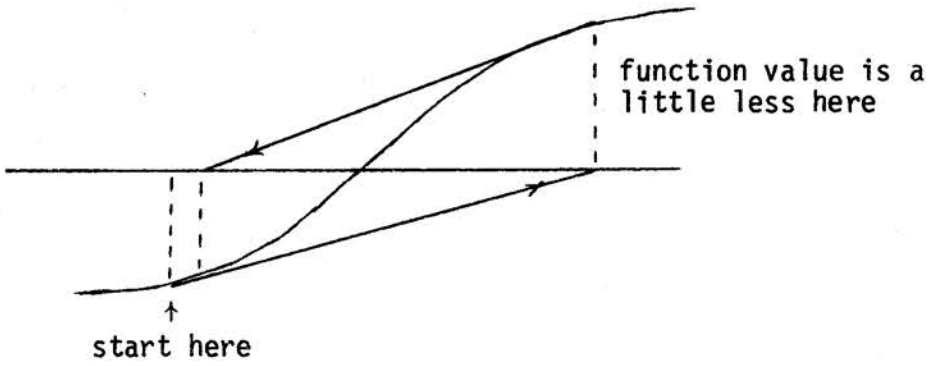
(2) So people are obliged to discover that they are at a minimum of the magnitude of an analytic function with respect to variation along a line which is necessarily a saddle point. Then they must turn the problem through an appropriate angle which depends on how many derivatives vanish.† You either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

---

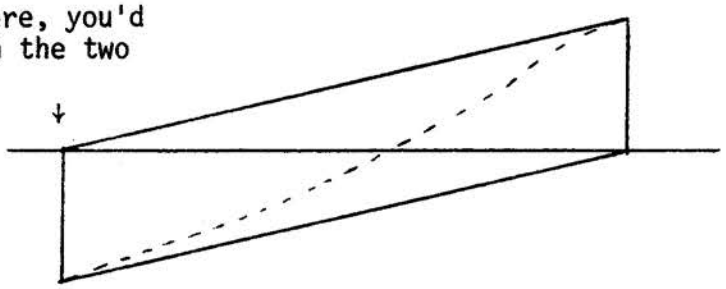
†If $f' = 0$ and $f'' \neq 0$, turn through 90°. If $f' = 0$ and $f'' = 0$ and $f''' \neq 0$, turn through 60° or 120°. If $f' = f'' = f''' = 0$, $f'''' \neq 0$, turn through 45°.

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.

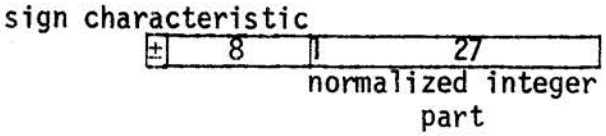
For an interesting algorithm of this type, see that of T.J. Dekker in Proceedings of the Symposium on Constructive Aspects of the Fundamental Theorem of Algebra.

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:

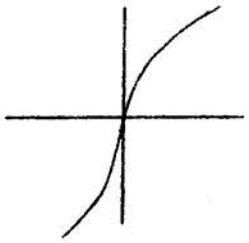


Specification for the SQRT routine:

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.

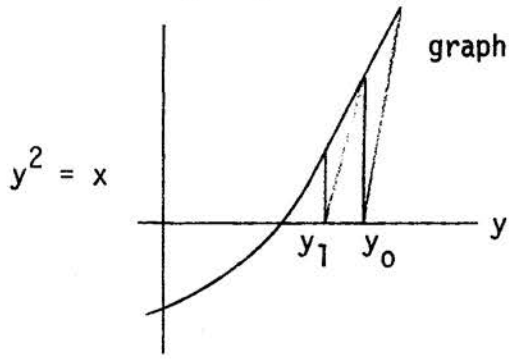
Answer: That is a problem that will have to be looked at by the people who do that project. But I'm talking about a 7090, and on it if two numbers are different their difference is nonzero, unless it underflows, and then you

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.



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

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

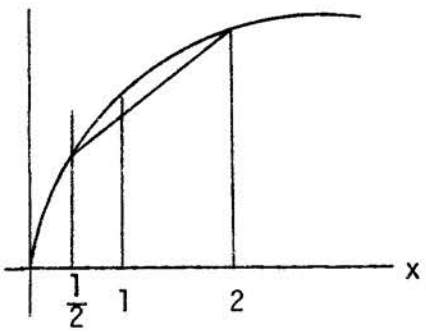
$y_1 = \frac{1}{2}\sqrt{x}\left(\frac{1+\delta_0}{1-\delta_0} + \frac{1-\delta_0}{1+\delta_0}\right) = \sqrt{x}\left(\frac{1+\delta_0^2}{1-\delta_0^2}\right) = \sqrt{x}\left(\frac{1+\delta_1}{1-\delta_1}\right)$ so $\delta_1 = \delta_0^2$ (quadratic convergence)

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

To Get the Approximation

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:

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

$J = 0 \text{ or } 1$

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

$$x_{n+1} = \phi(x_n)$$ 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: $\phi \le x \le \phi^{-1}$ (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.*

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

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

$$\pm \ell n(f/F) \quad .$$

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,

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

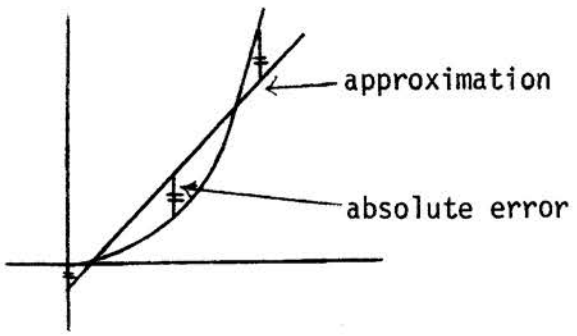
I would get different coefficients. And the coefficients might have been harder to compute.

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.

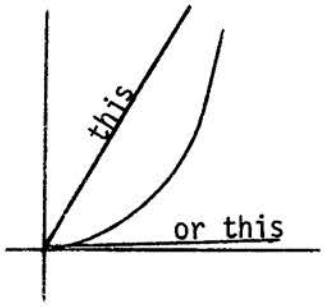
Question: I'm still worried about the fact that errors where the approximation is greater than $\sqrt{x}$ are treated differently from the other side. Is there very much difference there?

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.



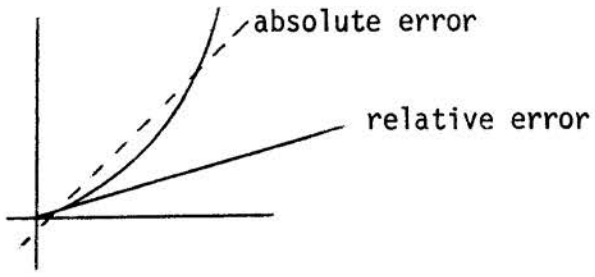
The best linear approximation will have the maximum error in each section equal, using absolute error.

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:

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

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.

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

$J = 0$ or 1

$y_0 = 2^I(C + \frac{F}{2} - \frac{J}{4})$ $\qquad$ 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.

What Happens in the Code

	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.

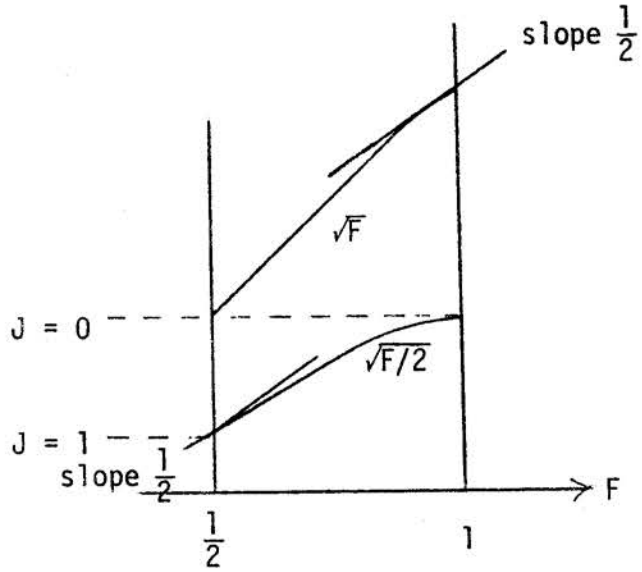
Choosing C

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:

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

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

Do a transformation for $J = 1$

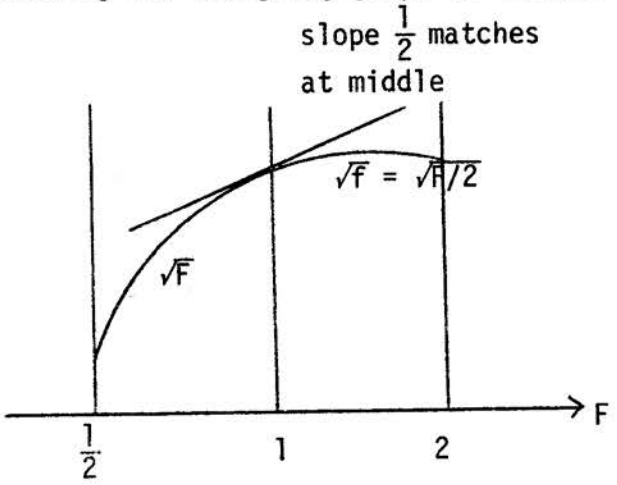
$$F \equiv \frac{1}{2}f$$

$$y_o \equiv \frac{1}{2}Y_o \doteq \frac{1}{2}\sqrt{f}$$

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

$$1 \leq f < 2$$

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:



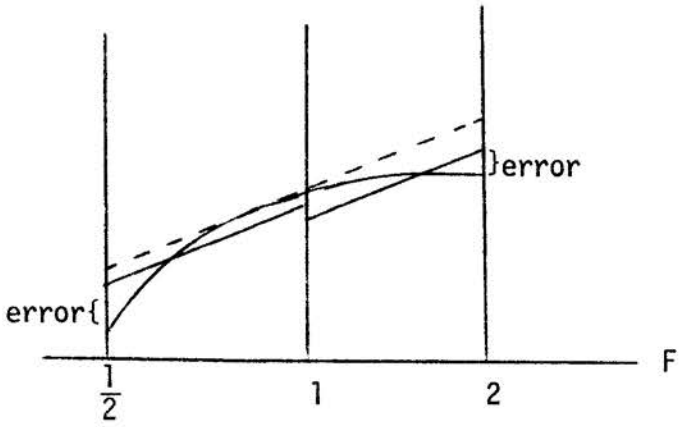
It is just a matter of scaling, so the relative error is still the same.

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_o and C_1 so that the two constant expressions would be the same.

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

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:

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

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

$$\delta_1 = \delta_0^2$$

†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.

What If You Use a Less Accurate C

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.

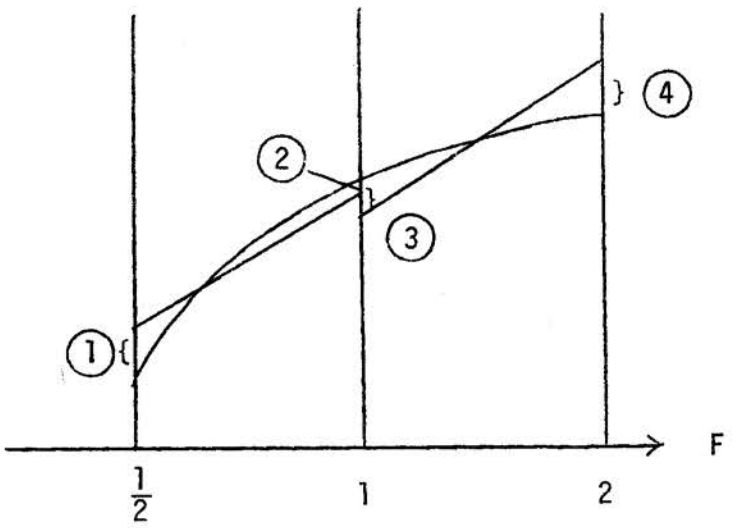
If you believe that everybody who writes programs does this sort of thing, or should do this, you've missed the point. The issue is to see how well we could do and how much it would cost.

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

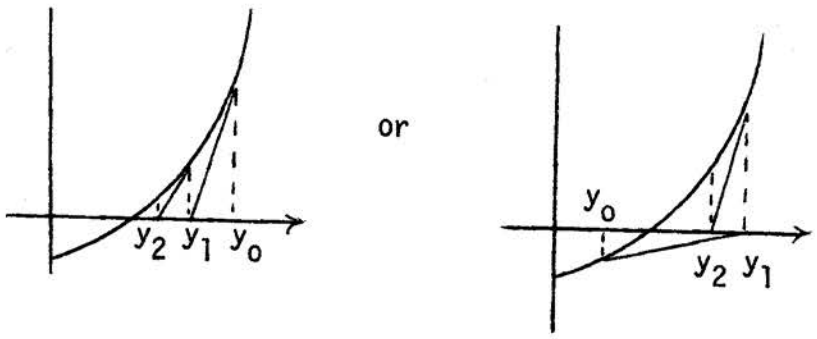
Now, let us look and see why rounding errors, up to the third iteration, do not make things appreciably worse. Here is the code again.

```
          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 = ½(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.

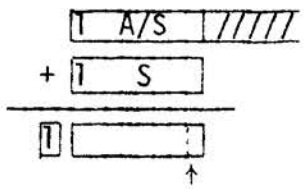
In a double length register called the AC and MQ, we will have:

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

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}).

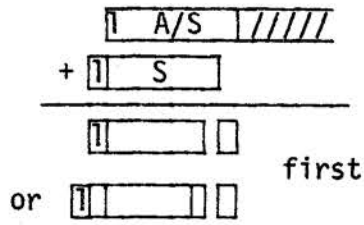
Case 1: exponents equal



↑
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



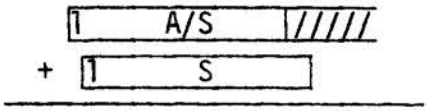
Again, you can see that the digits lost from truncating A/S would not have been used.

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:

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

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

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:

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

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

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

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 27^{th} 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

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

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?

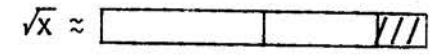
Answer: My estimate will be sufficiently close so that getting those results will be easy. We've got that now. I don't really need to find those 29 numbers. I get what I want by computing the square root almost to double precision; it's a little too big by some numbers near the end of the second word. 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:

$$SQRT(X) = \sqrt{x}(1+e) \qquad \text{rounded to 27 bits}$$
$$|e| < 2\times 10^{-14}$$

$\sqrt{x}(1+e)$ 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.

$$X = 2^{26}M \qquad \text{case 1}$$
$$X = 2^{27}M \qquad \text{case 2} \qquad 2^{26} \leq M < 2^{27}$$

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:

$$N - \frac{1}{2} \leq \sqrt{X} < N + \frac{1}{2}$$
$$2^{26} \leq N < 2^{26}\sqrt{2} \qquad \text{case 1}$$
$$2^{26}\sqrt{2} \leq N < 2^{27} \qquad \text{case 2}$$

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; $(2N \pm 1)^2$ is also an integer; therefore C must also be an integer. The values used for e are limited in that $(1+e)^{-2}(2N \pm 1)^2$ 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	$2^{26} \le N < 2^{26}\sqrt{2}$	$(2N \pm 1)^2 \equiv C \bmod 2^{28}$
Case 2	$\sqrt{2}2^{26} \le N < 2^{27}$	$(2N \pm 1)^2 \equiv C \bmod 2^{29}$

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.

$$(2^M - Z)^2 \equiv Z^2 \bmod 2^m \quad \text{(is really } (-Z)^2\text{)}$$

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:

$$0 < Z < 2^{M-1}$$

$^{\dagger} C = (2N \pm 1)^2 \bmod 2^{28} \text{ or } 2^{29}$

$C = 4N^2 \pm 4N + 1 = 4N(N \pm 1) + 1$

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:

$$Z,\ 2^{m-1}-Z,\ 2^m-Z,\ 2^{m-1}+Z$$

Only Four Solutions

I've shown there are four. Are there any more? No, and let's see why:

Suppose $Z^2 \equiv Y^2 \equiv C \bmod 2^m$

$0 < Y \leq Z < 2^{m-2}$ $\quad C \equiv 1 \bmod 8$

Can 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:

$Z-Y = 2^i p$ $\quad i \geq 1$ $\quad p$ is odd or zero

$Z+Y = 2^j q$ $\quad j \geq 1$†† $\quad q$ 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 mod 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 $i = 1$

$\Rightarrow j \geq m-1$

$2^{m-1} \leq 2^j q = Y + Z < 2^{m-1}$ †† hard to understand

Anything that's hard to understand can't happen. So this doesn't happen. The last case to try is $j = 1$

$\Rightarrow i \geq m-1$

$2^{m-1} \leq 2^i p = Z - Y < 2^{m-2}$ or $P = 0$

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: $C_m = C \bmod 2^m \qquad 0 \leq C_m < 2^m$

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.

$C_3 = 1$ (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

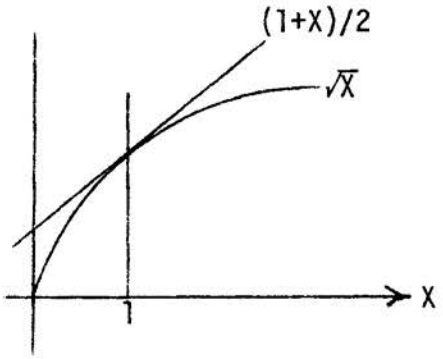
Now let us look at some other aspects of the SQRT program. We shall see what difficulties arise when we try to carry out a similar analysis of a simple routine like SQRT on another machine.

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:

$$\text{SQRT} = Y - (Y - X/Y)/2 \quad .$$

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

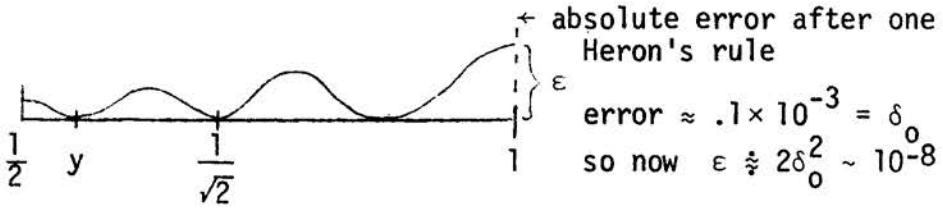
$$f(x) \doteq \sqrt{x}$$

$$f(\alpha x) \doteq \sqrt{\alpha x}$$

$$\frac{1}{\sqrt{\alpha}} f(\alpha x) \doteq \sqrt{x}$$

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.

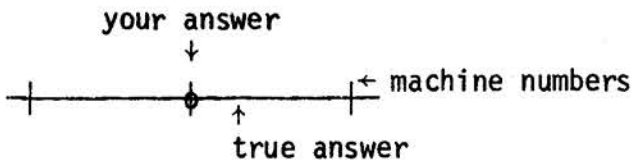
This routine was computed to take 52.8 μsec; the RUN library version takes 80.3 μsec. We did not look at any other versions. (The RUN version was optimized for the 6600, so it may not be so good on the 6400; it uses a strange formula.) The RUN version claim is that out of 200,000 random numbers one answer 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.

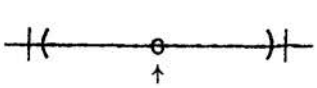
Two accuracy principles:

(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 $x = 3$, $y = 4$.



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 $\delta_0 = .16 \times 10^{-3} < 2^{-12}$. We got this result by knowing where, on the δ error graph, the error would be maximal.

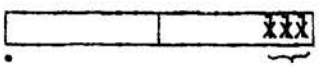
Errors $\delta_0 < 2^{-12}$

$\delta_1 < 2^{-24}$

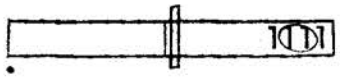
$\delta_1' < (\ ) < 2^{-24}$ (remember the S subtracted)

$\delta_2 < 2^{-48}$ 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 $\boxed{xxxxx00000\underset{1}{0}\cdots\cdots\cdots\underbrace{xxx}_{\epsilon}}$ If $\epsilon > 0$, rounding gives the correct result.

$\boxed{xxxxx11111\underset{1}{1}\cdots\cdots\cdots\underbrace{xxx}_{\epsilon}}$ If $\epsilon < 0$, rounding propagates carries and you recover the correct answer.

Monotonicity

$$x \leq y \Rightarrow \sqrt{x} \leq \sqrt{y}$$

Assume $A = 2^{47}$. $\sqrt{A} = 2^{23} + \cdots$ $(\sqrt{A} = \sqrt{2} * 2^{23})$

$$(A+1)^{1/2} = A^{1/2} + \underbrace{\frac{1}{2}\frac{1}{\sqrt{A}}}_{\frac{2^{-24}}{\sqrt{2}}} - \frac{1}{8}\frac{1}{(\sqrt{A})^3} + \cdots$$

$\boxed{1\qquad\underset{01}{1}\quad xxxx}$

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

$$\sqrt{x} \doteq \frac{ax+b}{cx+d} \qquad [\underline{\frac{1}{x}}, \bar{x}]$$

There is a theorem that says there is a best rational approximation to a function on a given interval. This approximation should hold if $x \leftarrow \frac{1}{x}$.

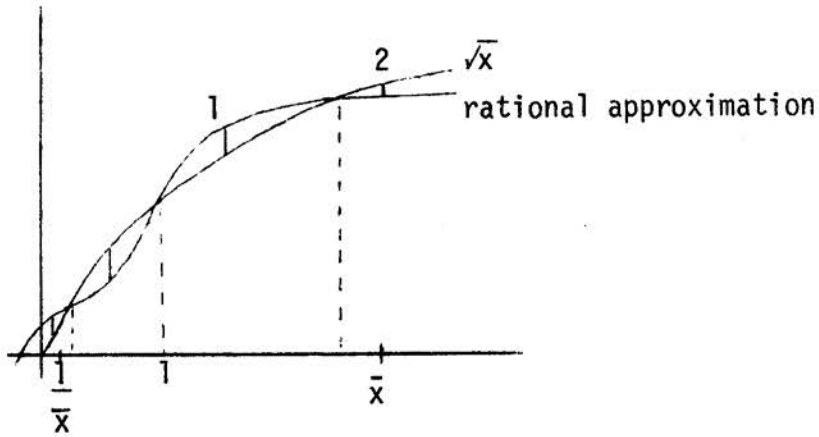
$$\frac{1}{\sqrt{x}} \approx \frac{\frac{a}{x}+b}{\frac{c}{x}+d} \approx \frac{1}{\frac{dx+c}{ax+b}} = \frac{1}{\sqrt{x}}$$

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.

Theorem. 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 diddle 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 4^{th} order while one step of the other method (8 operations) is 3^{rd} 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.

CABS

$$Z = \text{CABS}(X,Y) \qquad |Z| = \sqrt{X^2+Y^2}$$

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.

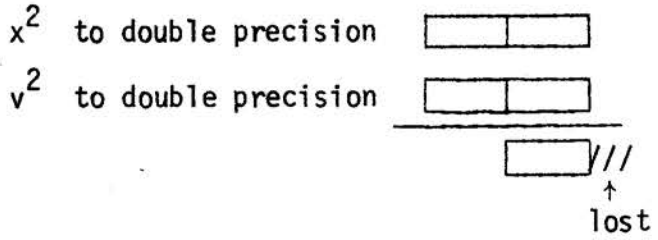
We avoid the division by scaling instead.

$$X = x \cdot 2^n$$

$$Y = y \cdot 2^m \qquad V = y \cdot 2^{m-n}, \quad n > m$$

$$Z = 2^n \sqrt{x^2+v^2}$$

The only problem then is how exact x^2 and v^2 can be.



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)

$\sqrt{Z}$ = (U,V)

a = CABS(Z)

b = $((a + |X|)/2)^{1/2}$

c = Y/2b

if $X \geq 0$, U = b, V = c

if $X < 0$, U = sign(Y)*c, V = sign(Y)*b

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?

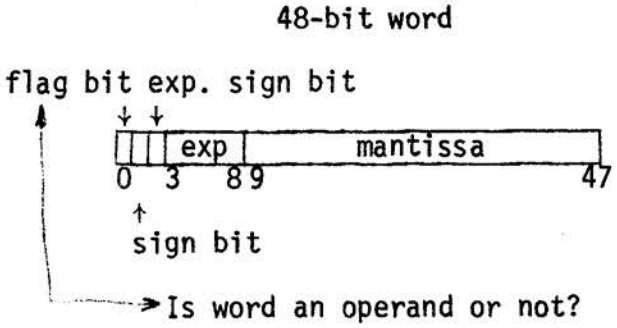
I. STUDENTS' REPORTS ON MACHINE ARITHMETIC

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.

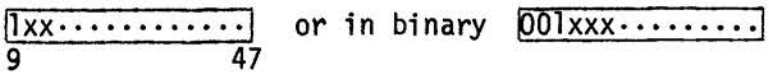
Burroughs B5500 Machine

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 (exp) 39 (integer•)	± 7 (char) 24 (•mantissa)	± 8 (char) 27 (•mantissa)	± 11 (char) 1.xx 36 (mantissa)	± 11 (char) 48 (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 an 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 14^{th} 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 Kahan 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 $77\cdots7\times8^{63}$?

Answer: $((8^{13}-1)\times8^{63})^2 \sim 8^{26}\times8^{126} = 8^{13}\times8^{139} = 8^{13}\times8^{11}$

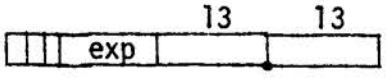
So they should save two characteristic overflow bits. You do not still have the operands. They were on the stack and they have been destroyed. You have irrevocably lost a binary digit. It could have been saved in the exponent sign bit since you know overflow could only occur for positive exponents. You 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 25^{th} 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.

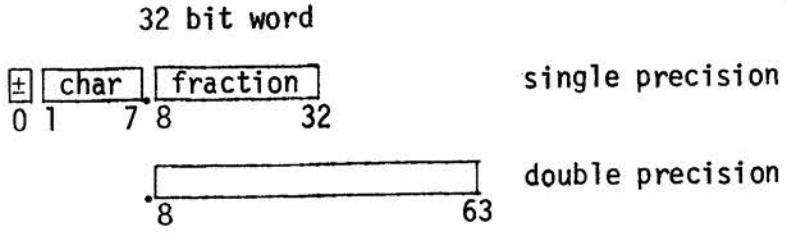
Question: How do ±0 compare in logical operations?

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 $-64 \leq exp \leq 63$

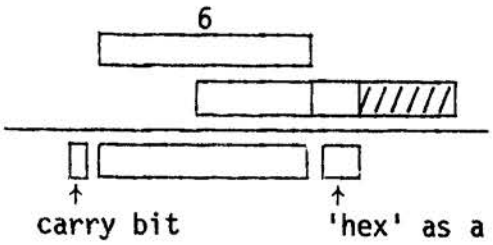
number is fraction $\times 16^{exp}$

$5.4 \times 10^{-79} \leq f \leq 7.2 \times 10^{75}$ Representable numbers expressed in sign magnitude.

The rounding rule is: Do operation to infinite precision. Then round to 6 or 14 hexadecimal digits.

When {adding / subtracting} magnitudes, the rule is

$$Z = TRUNC(X \pm TRUNC(Y))$$



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

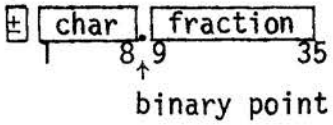
$$\frac{16^{-6}}{16^{-1}} = 16^{-5} \text{ in the last place (almost)}$$

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.

IBM 7094

floating point representation

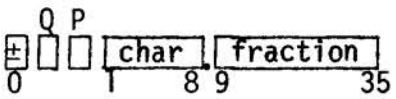


binary machine

characteristic biased by 128, $-128 \leq exp \leq 127$

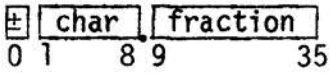
27 bits in fraction

Arithmetic is done in an accumulator



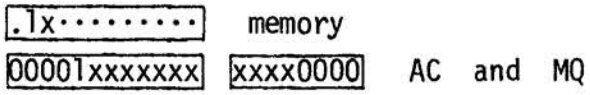
Q, P overflow bits

Another register which sometimes acts like a right hand extension of the accumulator is called the MQ.



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.



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.

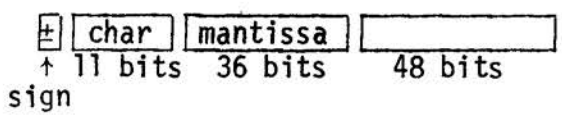
Question: Is any information lost on over/underflow?

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

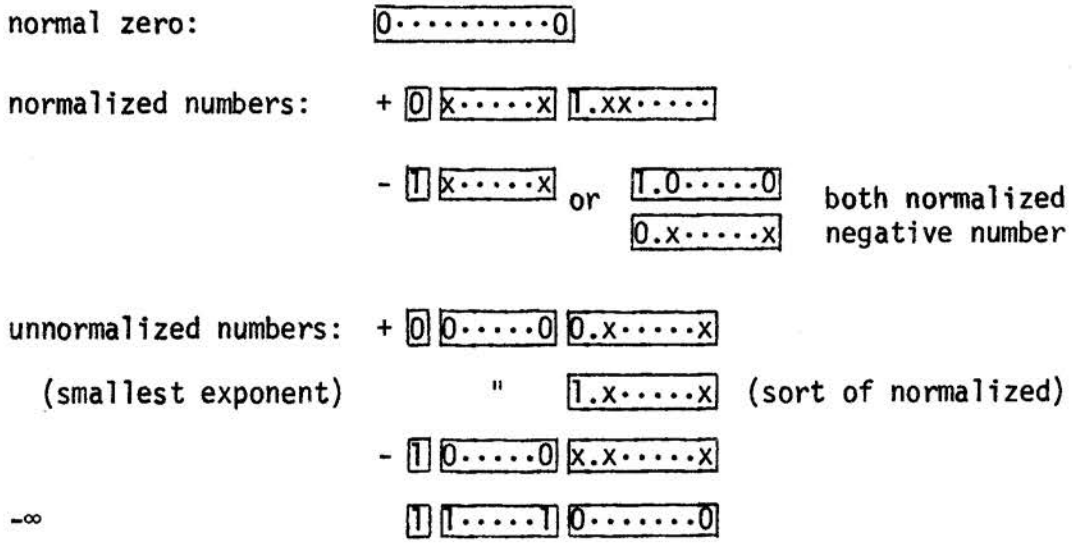


Double precision adds 48 bits to the mantissa

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

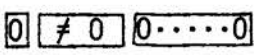


Hardware for machines exists, but not all of the microcoding for floating point and none of it for double precision or for handling $-\infty$ exists.

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



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.

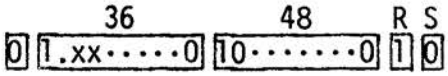
The result is rounded to 36 bits only when you do a store operation in single precision mode.

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.

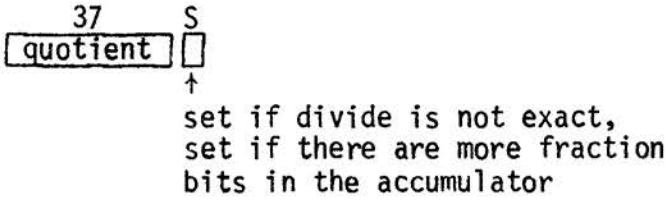
Example where double precision arithmetic and then rounding don't work properly (according to the manual):



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):



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 0|0···0|1.0·····0 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

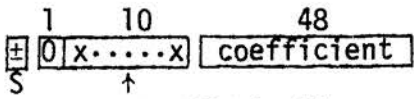
numbers represented as: $\text{sign} \times 2^{\text{exp}} \times \text{coefficient}$

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

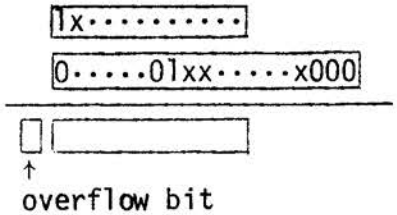
We will principally discuss single precision, normalized numbers. Since 1's complement numbers are isomorphic to sign magnitude numbers, we will only talk about magnitudes.

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



The smaller number is put into a 96 bit register and right-shifted. Bits fall off the right end.

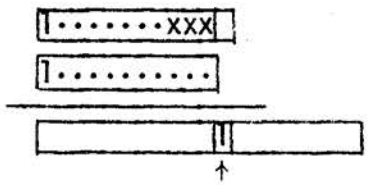
In truncated add, right bits are dropped. If the sum overflows, one

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 $A \oplus B = A(1+\epsilon_1) + B(1+\epsilon_2)$ $\quad |\epsilon_i| \leq 2^{-47}$

error in $A \ominus B = A(1+\epsilon_1) + B(1+\epsilon_2)$ $\quad |\epsilon_i| \leq 2^{-47}$

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 ϵ_1 maybe different from ϵ_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.

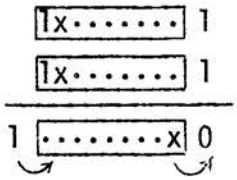
On the 650 (to two digits), 100 - 99 = 10. The ϵ_i's could be made equal, but they'd be huge and you wouldn't want to use them.

Here you violate the rule that says if the answer could be represented exactly in the machine, it should be.

Rounding Add

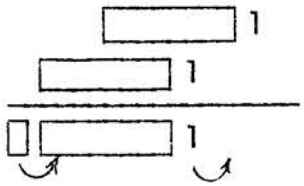
Add a 1 bit at the end of each normalized operand (try to normalize the operands), unless you have a 0. (Zero is not normalized as far as this operation is concerned.)

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 50^{th} bit instead of 49^{th}, 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.

$$A \oslash B = A/B(1+\epsilon) , \quad |\epsilon| \leq 2^{-47}$$

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.

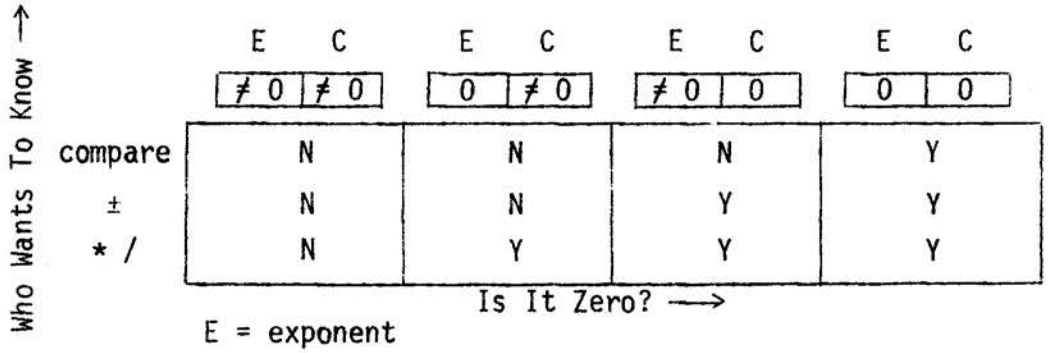
$$A \oslash B = A/B(1+\epsilon) , \quad |\epsilon| \leq \frac{2}{3} \times 2^{-47}$$

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?



E = exponent

E = 0 means smallest possible exponent

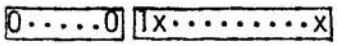
C = coefficient

Column 3 disappears when normalized numbers are demanded. If the ceofficient 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 2^{nd} column type.

There is a strange number on the CDC that is treated differently by different units.



It is normalized since the leading bit is a 1.

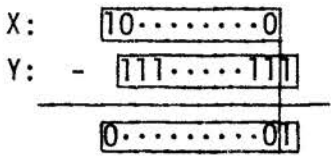
As far as the add unit is concerned, this number is legal. But the

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:



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 ≠ Y.

Overflow and Underflow Peculiarities

If you overflow, there is no trap, but a certain bit pattern is produced, 3777x···x, 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
2.5    NX6   X6      NORMALIZE UPPER SUM
3      FX7   X6+X0   ?
```
(II.3)

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 the result is now normalized.

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.

Case II. 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.

Case III. 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 96 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   X1±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:

```
DXO   X1*X2   DOUBLE MULTIPLY
FX6   X1*X2   FLOATING MULTIPLY                    (II.7)
RX6   XO+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 constant 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
PX6   X6      START TO FLOAT IT
NX6   X6      NORMALIZE
```
(II.9)

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
NX6   X6      NORMALIZE THE RESULT
```
(II.10)

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 not 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 exact. 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 compilation 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).

SNGL, 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 XO+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.