Default Rules for Rounding
Fixed Precision Floating Point Arithmetic
Ignoring Over/underflow.

The following rules would have to be amended only slightly to allow for over/underflow, which is a nearly independent and much more complicated topic. For simplicity here we consider the "representable numbers" to be an infinite discrete subset of the continuum of real numbers.

#1: The representable numbers must include 0, 1 and, if x then $-x$ too.

#2: Each representable number must be represented uniquely by a symbol string that represents nothing else.

#3: Any arithmetic operation* which, when executed without roundoff error, would produce a representable number, must actually be executed without error.

#4: Do not discard information unnecessarily.

#5: Any arithmetic operation which cannot be executed without roundoff error must result in a representable number nearest what would have been produced in the absence of roundoff error.

#6: The preceding rule is ambiguous when two representable numbers are nearest the unrounded result. This ambiguity must be resolved in a systematic way which preserves sign symmetry (e.g. $x-y = -(y-x)$) and is "unbiased" in the sense that "drift" cannot occur; e.g. the sequence $x_0, x_1, x_2, \ldots$ defined for arbitrary x_0 and y by $x_{n+1} := (x_n + y) - y$ has $x_1 = x_2 = x_3 = \cdots$

* The arithmetic operations include $+$, $-$, $\times$, $/$, $|\cdots|$, and conversion; and might be extended to include $\sqrt{\ }$ and other FORTRAN functions if the rules above were slightly relaxed.

W. Kahan
Univ. of Calif. @ Berkeley
Sept. 20, 1969

First Lecture - Tues Oct 6, 1970

note: These notes on the first two lectures were not made using a tape recording. All other notes were made using a tape + class notes.

Consider a FØRTRAN program that contains the following statements:

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

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

Then there's the problem that only a finite no. of numbers can be represented on the machine.

CDC supplies ∞ approximate ∞ and ∞ (indefinite)

Are CDC's rules in handling these reasonable? Is it reasonable that you should get thrown off the machine if you try to use these numbers?

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

You are to write a subroutine that will solve

for the roots of a quadratic equations, given A, B, & C. The equation is $Ax^2 - 2Bx + C = 0$; A, B, & C are single precision, floating point numbers. The roots are to be accurate to within ~~&~~ a few units in the last place, or if a root is out of range, there such be an appropriate message.

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

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

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

and on some machines C may not lie between A and B. [this is on octal or hexadecimal machines).
What if A+B overflows?

Think of ~~something~~ $Z = X + iY$ & wanting to compute $CABS(Z) = \sqrt{X^2+Y^2}$

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

So you try the subterfuge:

```
CABS(Z) = ABS(X) * SQRT ( 1 + (Y/X) **2 )      |X| ≥ |Y|
```

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

Computation of elementary functions: ln, sqrt

A fellow was computing:

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

He got the message that he was trying to take the sqrt of a negative no.

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

Say you ~~fire~~ have a system of linear equations that have no solution. Say:

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

"ill-posed"

Then consider the system:

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

"ill-conditioned"

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

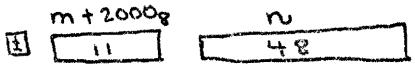
Representation of machine representable numbers

Floating point numbers:

$$\pm B^m \cdot n \qquad 0 \le |n| < B^K$$

B: 2 on binary machine; 48 on CDC 6400

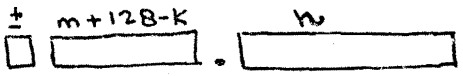
$-1024 \le m \le 1023$ (then biased) on CDC



CDC 60 bit fl. pt. word

on system 360: $B = 16$, $K = 6$ (or 14 in double precision)

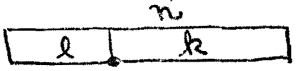
$$-128 \le m - k \le 127$$



Difference in 'n' on 360 & 6400: on 360, it is a fraction, on the 6400, it is an integer.

In a polish machine, numbers are represented by their logs

$$2^{n} \times 2^{-k} \qquad 0 \le n \le 2^{k+l}$$



On such a machine $a(b+c) = ab + ac$ exactly!
Adding & subtracting are more interesting. They use a gauss log. table which has:

$$e^{x} + e^{y} = e^{z}$$

Given x & y, you get z from the table.

The Table-maker's Dilemma, or how to store a possibly infinite number in a finite word length. You want to store a number as close as possible to the actual number you want to represent.

$.762449\cdots9$ → $.7624$

$.762450\cdots0$ → $.762\begin{matrix}4\\5\end{matrix}$ } which one to store?

The table makers says calculate the answer to more digits and then round the answer. But what if you get

$.7624\ 49\cdots9$

Maybe the $49\cdots9$ should really be $50\cdots0$.
In some situations, you cannot tell how many digits are necessary to insure correct rounding. Sometimes an arbitrarily large number of digits are needed, but you can't tell in advance!

Error Analysis

C = A + B FORTRAN statement; a is value stored in cell named A, etc

1-5

In some machines you have:

$c = a(1+\alpha) + b(1+\beta)$ like the CDC have bounds on α & β.

on others:

$c = (a+b)(1+\epsilon)$ $\quad |\epsilon| \leq 10^{-3}$ for 4 decimal digits

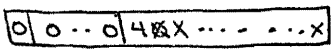
You are to read the CDC manual on floating point operations and see what happens.

Second lecture – Thurs, Oct 8, 1970

On the CDC, it is possible to get

$A * B = 0$ $\quad A, B \neq 0$ (see p. 3-18)

Say you test for $0 < a < 0.1$. This is done in the add box, which tests all digits. But when you say multiply, only the characteristic is tested, & if it is all zeros, the number is considered zero. Thus if

$x =$ 

then $x > 0$, but $x * y = 0$!

Note relation to protection

Why should you object to such treatment?

Well, you may get a wrong answer. But more important, you may be trying to debug a program. How could you tell if this is what happened? It may cost you a great deal of time because you don't trust your machine, as you check through all the funny quirks it has.

When can $C = \frac{A+B}{2} \notin [A, B]$?

Consider using 2 decimal digit truncated arithmetic:

$A = 99$

$B = 100$

$199 \rightarrow \frac{190}{2} = 95$

This can also happen in rounded arithmetic:

$A = 98$

$B = 99$

$197 \rightarrow \frac{200}{2} = 100$

→ Can you find an example in octal & hexadecimal arithmetic? put them in
Can it happen on a binary machine?
Is this serious? ? $A/2 + B/2$

Dodge $A + \frac{1}{2}(B - A)$

→ Consider: is it possible to design a reasonable machine on which:

1. $A - B \neq -(B - A)$
2. $1.0 * X \neq X$

a) in case of overflow
b) when there's no overflow

→ Look into Knuth, vol. II on floating point arithmetic.

Rules: the way I'd like things to be; if they must be abandoned they should be abandoned from the bottom up.

1. The representable numbers must include 0, 1, and if x, then $-x$.
2. Each representable number must be represented uniquely by a symbol string that represents nothing else, at least as far as the programmer is concerned.

   i.e., if you don't have uniqueness then the two can be interchanged (with no loss) for any use - they must act the same in operations.

   e.g. $X = 2 = 20 * 10^{-1} = 02 * 10^{0}$

What happens when you try to use unnormalized operands?

B5500: $X = 20. * 10^{-1}$

$Y = 50. * 10^{-2}$

$Z = X + Y = 25 * 10^{-1}$ works okay

360 oddity:

X: 2.0000×10^{0}

Y: 5.0324×10^{-1}

2.5032×10^{0}

but if $0.00002 * 10^{5}$ unnormalized

$5.0324 * 10^{-1}$

$.00002 | 5 \ldots.$

$\Rightarrow 2.5000 \times 10^{0}$

This means that the two representations for 2 are not equivalent on the 360. The software is designed so that the second 2 is suppressed, but you can use unnormalized arithmetic & then you run a risk.

On CDC, the symbol string -0 represents two things.

for arithmetic, $-0 = +0 = 0$. But for branches $-0 \neq +0$.

3. Any arithmetic operation which, ~~when~~ when executed without roundoff error, would produce a representable number, must actually be executed without error.

e.g. $2.0 * 2.0 = 4.0$, not $3.9\cdots9$

$x/x = 1.0$ if $x \neq 0$

$x - x = 0.0$

[note: 1/10 is not representable on a binary machine, so can't talk about things like $\pm 0 \cdot 10$]

4. Do not discard information unnecessarily.

~~e.g.~~ e.g., don't throw away the remainder in division if you have a place to put it.

if the answer under- or overflows, don't throw away the digits that are left

5. Any arithmetic operation that cannot be executed without error must result in a representable number nearest to what would have ~~been~~ been produced in the absence of roundoff.

e.g., round by half in the last place, don't chop

problems: what happens in the half-way case? Tablemakers say round to nearest even.

6. The preceeding rule is ambiguous ~~in the case~~ when two representable numbers are nearest the unrounded result. This ambiguity must be resolved in a systematic way which preserves sign symmetry,

e.g., $x - y = -(y - x)$

and is unbiased in the sense that 'drift' cannot occur.

e.g., the sequence $x_0, x_1, \ldots, x_n$, defined for arbitrary $x_0 + y$, by:

$$x_{n+1} := (x_n + y) \mp y$$

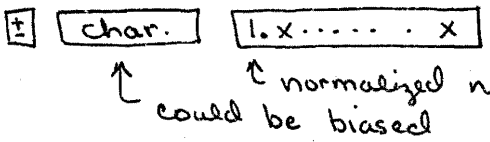
has $x_1 = x_2 = \ldots$ [not absense of x_0; this is different from Knuth].

see papers by DAVID MATULA on base conversions.
you run into this kind of problem by repeated ~~transf~~ transfer of data into various bases. E.g., truncation leads to values drifting down.

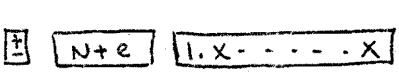
other examples: doing iterations on matrices + transfering into and out of storage ; Jacobi's method may get to the point of adding + subtracting the same no. to the diagonal.

The concept of these rules is that they are to apply to the rules seen by the FORTRAN programmer. The 6 rules do not apply for the systems programmer or to overflow + underflow.

Now, let's look in detail at a binary machine.
The reason for chosing binary will be discussed later.
Machine will be sign-magnitude (equivalent to 1'S complement)



exponent in the range: $-N, -N+1, \ldots, 0, 1, \ldots, N-1$
then the characteristic is $N+e$: and the number is $2^e * 1.x\cdots x$

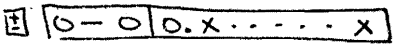


This machine obviously has $1, 0, x, +, -, \div$.
This is designed to have as few numbers as possible whose reciprocals are not representable. In fact, only 1 number doesn't.

$= 1 * 2^{-N}$

Closest you can get to 2^N is $2^{N-1} * 1.11\cdots 1$
When scaling [bringing numbers in a problem closer to the center of the range — if know how it will affect the result ; use power of the base to avoid rounding error]
may not get reciprocal of large numbers.

When can the leading 1 be abandoned?
Numbers will be treated correctly if the ~~exponent~~ characteristic is the smallest possible, namely zero.



Still have unique representation, but don't have reciprocals.
~~The~~ set of numbers is all the normalized numbers (leading 1) & those with a zero characteristic (since no reciprocal, ~~you~~ there is problem in dividing by them).

Third Lecture - Tues Oct 13, 1970

Wrote down the rules again in brief form

1. must have 0, 1; and if $-x$, then x
2. unique representation of numbers
3. preserve exactness when possible
[4. don't waste information]
5. approximate by nearest neighbor
6. resolve ambiguity in #5 by preserving systematic rules like sign symmetry *[by avoiding "bias" or "drift"]

These above are designed to recall the rules in extenso.

1) Rules are designed to be rules as might be understood by one who does not know how the hardware works & doesn't want to know, but wants to use floating point arithmetic to achieve some objective that has nothing to do with the advancement of computer science.

2) Rules are not a set of axioms; they are not independent, for example. Several of the rules are implied by earlier ones. Tried to write them out roughly in the order of priority as he (I?) thinks they should be so that if you did have to abandon some of these rules, you should abandon them from the bottom up.

Should have looked into a chapter of Knuth, Vol. 2. Would have found some interesting material there. By not looking into it, you would have missed perhaps the most important conceptual ~~sy~~ summary of rules of the above kind, namely, that if you succeeded in satisfying all of the rules except #4 (not a rule that

can easily be put into axioms), + the last line of #6 – that is, they are not treated in Knuth's book – but the others are. Knuth observes that the essential fact is that these rules could be implemented easily if you thought of one way to round numbers to a specified field length + then insisted that every arithmetic operation be equivalent to the following:

first perform the operation without error thereby generating a result requiring more digits than will fit in the field length, then call your rounding procedure.

think of some arithmetic operation that could be +, −, *, /. If you wish to know how the machine actually computes, when you ask it to do this operation, first it performs that operation quite precisely (in principle), + then, it invokes a rounding procedure + that's what it stores.

$Z = X \odot Y$ Fortran statement – what you intend to have happen

$z = \text{round}(x \circ y)$ what the machine does

$\odot = \{+, -, *, /\}$

If you can work out an appropriate rounding procedure that applies uniformly to all real numbers, then it would suffice so to design the hardware that this happened in order to be able to satisfy all the rules except the ones bracketed

Knuth shows that if $z = \text{round}(x \circ y)$ is satisfied, proofs of what might otherwise seem to be rather tricky

relations are relatively routine (may still be long).

It has not always been clear how so to construct an arithmetic unit that you would satisfy just this relationship.

Question: what is the distinction between X and x?

X is name (in a FORTRAN statement) of a real number x & it is represented in storage somewhere. You like the machine at execution time to do the operation & store the result & would rather ignore rounding errors. Your intention is expressed by:

$$Z = X \odot Y$$

but the actuality is expressed by:

$$Z = \text{round}(x \odot y)$$

You write a FORTRAN statement & something else happens.

It wasn't always clear that an arithmetic unit could be designed economically to follow the rules (Knuth talks about number of digits needed or size of registers). Kahan's machine from last time showed that you could implement this rule (rounding properly) without having to know $x \odot y$ precisely - without having to store that operation precisely. Remember the Table Maker's Dilemma - to round correctly he might have to know some result exactly.

Interesting property in rule #6 (concerns way in which you round) is not discussed very thoroughly in Knuth's book. Look into what Knuth says.

Will continue to show it is possible to implement this sort of scheme: $Z = \text{round}(x \odot y)$

Question: You said there was a basic ~~com~~ connection between the relation $z = \text{round}(x \circ y)$ + most of those rules. It seems that it has nothing to do with #1.

Answer: Yes, #1 is a separate hypothesis

Question: What about unique representation?
Answer: That's okay, because even if numbers are represented differently internally, it would be impossible by floating point operations to distinguish them.

Question: Three is covered by this.
Answer: Three is obvious. #4 is irrelevant. #5 is the essence of the rounding rule.

Question: So it is #2, #3, + #5.
Answer: Yes, + in rule 6 could preserve ~~things~~ rules. like sign symmetry. Rules ~~that~~ that could be preserved could be characterized in the following way:

consider 3 functions: $f(x)$, $g(x)$, $h(x)$, map representable numbers onto representable numbers.

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

or $g(x) = 2^k x$ (binary machine, k an integer)
maps representables onto representables except for over/under-flow.

To resolve ambiguities in a systematic way, would want to ~~be~~ preserve the following property:

if $h(x \circ y) = f(x) \odot g(y)$

example: ~~h(x)~~ $f(x) = -x$ $g(x) = -x$ $h(x) = x$ $\odot = *$

want machine to preserve that relation too.

If you round $h(x \circ y)$ & round $f(x) \odot g(y)$, ~~the~~ you want the relation to still be true.

This example is preservation of sign symmetry. If reverse sign of $x * y$, sign of product shouldn't change.

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

On another machine, people used its divide subroutine for three years without knowing that you got the wrong result if you divide by a negative number.

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

"Bias" + "drift" not included in systematic rules as that depends on the rounding used.

Show how one can design an arithmetic unit so that without undue increase in expense, you could accomplish those rules insofar as they are applied by doing exact arithmetic + then use a rounding rule.

To Describe such a machine

will use binary arithmetic, for reasons other than those under discussion. binary point will be to the right of the leading digit, which is a 1 for normalized numbers, then a string of other digits. Have a field for the characteristic, which would normally be biased (altho that isn't important right now).

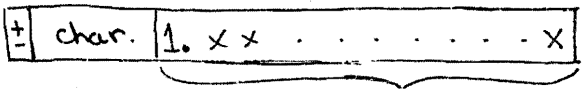
$-n, -n+1, \ldots, 0, 1, \ldots, n-1$ for the exponent

n an integer, normally a power of 2.

characteristic = exp + N so characteristic is a positive number

have bit for the sign of the number - using sign magnitude representation. None of this is really important; just done to make things definite.

so it looks like this.



↑ word length as far as arithmetic unit is concerned

use separater register for ~~character~~ characteristic manipulation
sign goes into separate flip-flop to tell whether to complement or not

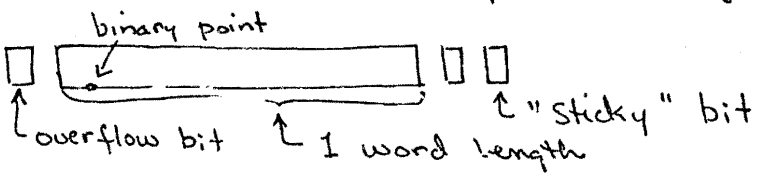
the word length is what can be called 1 single precision word : on CDC, that is 48 bits; 7094, 27 bits; IBM 360, 6 hexadecimal digits

How wide do the registers have to be in order to be able to follow the rules for single precision arithmetic

Question: from ~~q~~way you have written the exponent, does that imply it is a 2's complement number, is -0 not there?
Answer: That's it, there's no -0 for the exponent, so it really is 2's complement, whereas on the CDC the exponent is a 1's complement number biased in a 2's complement fashion. ~~There~~ are no wasted exponents in our machine.

How many more digits will we need? for active circuits that do addition, need only be as long as 1 ~~single~~ single precision word plus 1 bit on the right in which active addition may have to take place plus a "sticky" bit to the right of that + one bit on the left for overflow

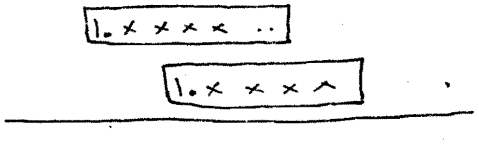
purposes. This is not ~~fraction~~ characteristic overflow (not sending you out of your number range), just intermediate overflow in dealing with significant digits.



It is possible to accomplish what we want with a register of that length. You don't need a double length register. That doesn't mean it would be economical to build a machine this way. In fact, I am of the opinion that the hardware used in arithmetic units has ~~bee~~ become so cheap, compared to the rest of the machine, that ~~we~~ it is unlikely we would want to execute arithmetic in ~~any~~ a way that minimizes the amount of hardware. Hardware is cheap; time is much more expensive & it is storage access that is the most significant contributor to the cost in time. The foregoing is just a way of thinking how to do the arithmetic.

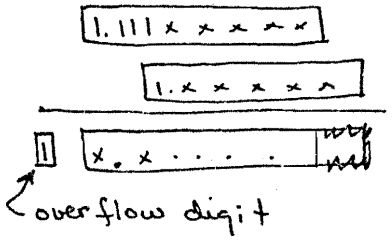
Start with an add operation - adding two floating point numbers & show how the extra digits would be used, ~~for~~ in particular, the "sticky" bit.

In adding two floating point numbers, usually have to shift the significant digits of one of them to the right, by some amount

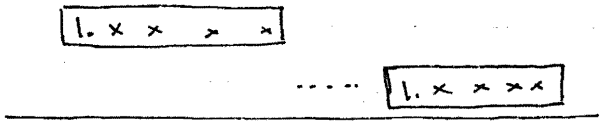


You then imagine that you add & that something would have to be done to extra digits. If the upper number has consecutive binary ~~digit~~ 1's, an overflow

digit of 1 could be generated.



then you'd have to do a right shift of the whole works. Whether that happens or not, the end result is likely to have extra digits on the right. In unusual circumstances, the numbers could be so disparate in magnitude that one of the numbers would have to be right shifted a very great distance.



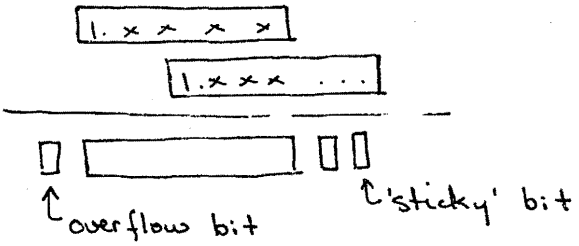
Then when you add, those digits might be several words to the right. If you ever wanted to know what the correct result was so that you could round correctly, you might think you'd need a register several words long. That in fact is obviously unnecessary. All you really want to know is:

1) after normalization of the result, what is the digit next to the rightmost digit? I need that to do my rounding.

2) also have to know if there are any digits farther right of that one. Are there any there at all? That's what the "sticky" bit is designed to tell.

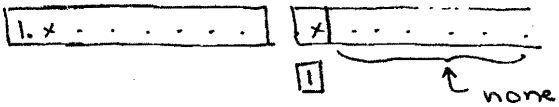
So really only need 2 bits worth of information, to be able to round correctly. One bit is the actual bit that should be there that you'll throw away when you've used it, & the other bit is whether

there are any other bits to the right.



This is enough. Why should this be the case? Consider rounding. What sort of rounding function should you use?

Look at a word - can only store the leading, single precision piece of it. What you store will depend on all the rest of the digits. The usual rounding procedure is to add 1/2 unit in the last place. But if you always add 1/2 in the last place, only the bit directly to the right will have any affect.



But if you always do this (add 1/2 in last place), you will violate that part of rule #6 that says avoid "drift"

If, instead of rounding, you decide to truncate. You will also introduce "drift".

But that doesn't worry you. You may want to be able to say to users of your machine that what you will get is equivalent to having computed the correct result, found so many leading significant digits as will fit into a single precision word, and then you chop off the rest & throw them away.

That does not mean you could ignore what's way out to the right.

Consider a subtraction, a small number from a large one.
e.g., $1 - 10^{-100}$

~~in zero~~ it is common to simply ignore the tiny number, to say it is too far to the right to matter.

Then your result is the larger of the two operands

$1.0 \neq 1.0 - 10^{-100}$ (2 decimal-digit machine)

That is not what you'd get if you'd truncate the true result.

$$1.0 - 10^{-100} = .99\ldots\ldots9$$

then, truncation leads to the answer: .99

In many respects, 1.0 is a rather better ~~answer~~ result, but it is not what you would get by applying the rule:

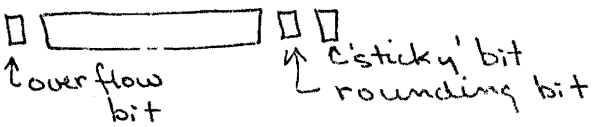
$$z = \text{TRUNC}(x \circ y)$$

so that if you had a proof that depended on this rule, it might not hold.

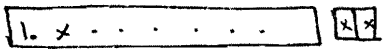
Question: Is what you are explaining ~~Does~~ that the machine does not conform to the specifications ~~in the manual~~ as the manufacturer stated because...

Answer: Not to the most obvious specifications. If you wanted to prove that a program worked & you used relatively simple rules, as Knuth does, your proof would be wrong because in certain instances the arithmetic did not behave the way you thought it should. Your theorem may be still be correct, just your proof was wrong. In events like this the error may be so small that it does not vitiate your theorem. That's a much more delicate, complicated business.

Instead of needing a terribly long register, just need to know if there are any bits in it at all, past the rounding bit.



In an add operation, you keep these two guard bits, As shift bits right through the 'sticky' bit, it stays zero until a 1 is shifted in, then it remains a 1 regardless. Hence the name "sticky" Round bit accepts whatever is shifted into it. Now add. Two extra bits don't change, only those in the single precision word, & overflow bit. If overflow, right-shift 1, "sticky" & round bits preserve their characteristics.

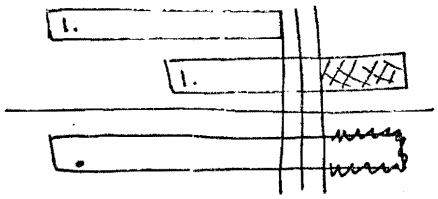


round as if "sticky" bit were just another bit.

1 0	special rounding rule - to nearest even to avoid 'drift'
1 1	round up
0 1	round down (truncate)
0 0	round down (truncate)

Question: What if rounding causes 1 bit overflow?
Answer: That's ok okay, since rest of word will be zeroes, just right shift 1 & adjust exponent. Haven't lost any 1 bits off the right end.

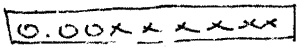
Subtraction of magnitudes



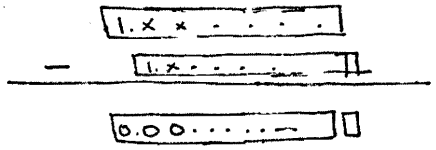
cannot be an overflow
right shift through 'sticky' bit.

Subtract as if sticky bit were a legitimate bit of the number. last two digits are 2's complemented.

What if you get some leading zeroes in the result? Then you'll have to left shift to normalize.



If have to shift left several places, "sticky" bit will travel, & there is something wrong with the "sticky" bit. But the point is that to get cancellation to yield several zeroes, the two numbers must have lined up to begin with, or only needed a shift of one position.



If one number was shifted for to the right, can't have a left shift of more than 1. See this because, number subtracted will be less than 1/2, so answer is ~~begin~~ bigger than 1/2, & to normalize will only need 1 left shift.

~~Exam~~ Even in the case of the special, unnormalized numbers with characteristic zero, there is no problem. Students should think about this, to see why it is so.

When have to shift left 1 place, sticky bit tells you how to round correctly. Sticky bit is now round bit, & it has been complemented. Round correctly without drift. Rules are different for subtracting.

~~('Sticky' bit is 1: 1 to the right of it~~

Question: What about the two cases:

$$1.00\cdots0 \;\Big|\; \begin{matrix} 011\cdots1 \\ 000\cdots1 \end{matrix}\Big\}$$ treated the same as 'sticky' bit holds the 1

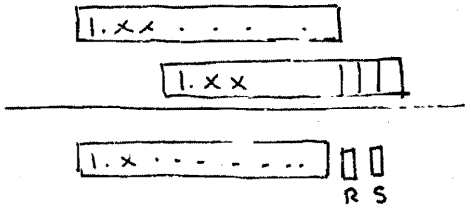
Maybe should distinguish having to left shift or not?

There is a problem if have to left shift only 1. Will have to sit down & work it out again. No problem if no

shifting, or more than one shift (then numbers lined up & or had only to shift right one).

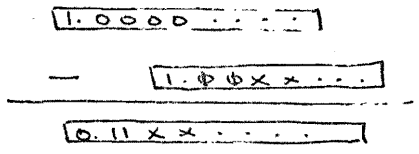
Problem only arises if had to right shift originally 3 or more places. If only shifted 2 places, then "sticky" bit is correct & arithmetic is correct.

If having trouble, it is because smaller operand has been shifted right more than 2 places; 'sticky' bit may be holding a 1 from very far right.



may have to have two round bits

Stupid Question: But if you've shifted right 3 or more places, you won't have to shift left at all.
Answer: consider

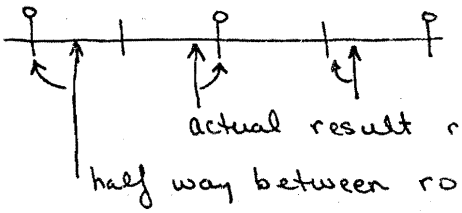


need a left shift of 1 to normalize.

Two round bits will certainly work: so use 2 round bits & 1 sticky bit.

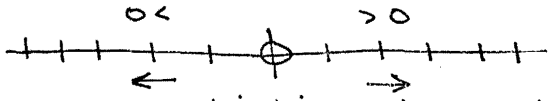
Students should try to work out if there is a trick so that only need 1 round bit.

Question: Do you round a negative number as the absolute value of the number or round it towards zero?
Answer: Round a negative number to its nearest neighbor unless half way between. That has nothing to do with zero.

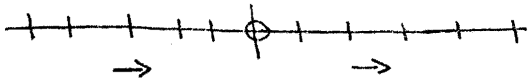


don't care where zero is in rounding

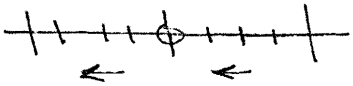
Question: for normal machines that round, they usually just add 1 in the last place. (actually 1/2)
Answer: sign-magnitude machines do just add 1 in ~~the last~~ first bit to be discard.



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



In 1's complement machine, like 6400, same thing happens as in sign magnitude machine.
If you truncate, throwing digits away, you are always moving to the left.



Don't want that to happen in my scheme.

Question on 2's complement rounding.
Answer: example in 2's complement rounding.

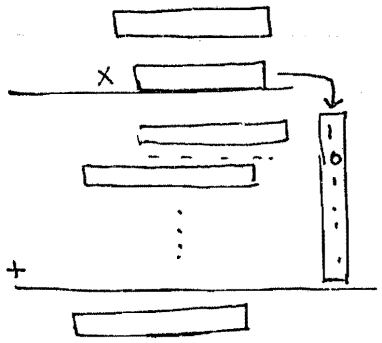
1.001 -7/8 in two's complement
1.00 truncated in two's complement
↑ representation for -1, so magnitude has been increased for a negative number

Need 1 or 2 round bits. But could you get along without the 'sticky' bit? Should verify for yourself that you cannot round to nearest neighbor without a 'sticky bit'!

Should verify for yourself that if you have to left shift answer by more than 1 bit, the answer is exact. Only if 1 left shift is required is there any problem.

Multiplication & division dealt with in principle by thinking of them as iterated addition + subtraction.

multiply:

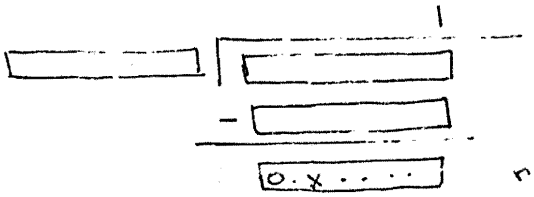


copy down when see a one, shifting as you go. Add from right to left. in all additions is suffices to retain the round bit + the 'sticky' bit

On actual machines, several lines are grouped together & added all at once. 6400 does multiplication in the slow way. The 6600 can do fast on numerous additions simultaneously (to speed up its operation).

On the last addition (to multiply) you inspect the round + sticky bits & apply the rounding procedure.
360 does it essentially this way, without the sticky bit.

division - quotient is generated by trial subtractions, so in ~~principle~~ principle you can always know what should be in the sticky bit.



right shift for next stage of division

generate digits for a quotient ≠ and a round bit, & look to see if there is a remainder. Tells what 'sticky' bit should be. Division can be rounded correctly.

Thus, you can build an arithmetic unit to satisfy

$\not\equiv$ = round $(x \circ y)$ round to nearest even.

Has "bias" or "drift" been eliminated by this construct? Will the following sequence be prevented from "drifting"?

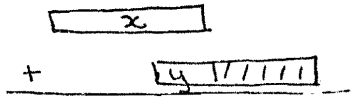
let $x_{i+1} := (x_i + y) - y$ $\quad x_0 = x$, y } arbitrary

is $x_1 = x_2 = x_3 = \ldots = x_{i+1}$?

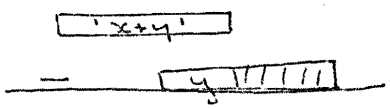
notice that x_0 does not appear in the sequence

Test on arithmetic already known

truncated arithmetic $x, y > 0$



'x+y' is too small



'x+y' - y' . . . . 1 thrown away.

also too small

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

could push x down until it compared with y, then process would settle down

Similar thing can happen if round up in the conventional way— that is, add ½ in the last place & throw away the fraction. But then you drift up instead of down.

example.

x_n = 1.00000 1101 — 9 significant bits

y = .100000001

'x_n+y' = 1.10001101 | 1 ⇒ 1.10001110 stored '$x+y$'

$-y$: .100000001

1.00001110 ⇐ 1.00001101

1.00001110 when stored

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

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

Same example by rounding to nearest even

first time through you get the same result

stored $(x+y)$ = 1.10001110

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

go through the cycle again:

1.00001110

+ .100000001

1.100011101 : rounding ⇒ 1.10001110

1.10001110

− .100000001

1.000011011 : rounding ⇒ 1.00001110

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

changed in first step if x is odd; then the sequence doesn't "drift"

Prove this in general by examining all the different cases.

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

Question: how about other possible sequences?
Answer: if multiply/divide is your sequence, it will also settle down. Arguments are similar to those used by Dave Matula in papers on base conversion. See his papers — usually have 'base conversion' in title, look in ACM

In the multiply/divide case, you can verify the result by observing that the error can't exceed half a unit in the last place, in either multiply or divide. So in one step, the error can't exceed 1 in the last place. Just have to show that if the error was 1/2 in both cases, something peculiar happens. Show that that just doesn't happen.

Question: Why isn't it economical to build a machine that rounds your way?
Answer: I said it has not be thought worth while to do it this way. People who build machines don't see that there is much value in building machines that eliminate the bias. Neither does Knuth as he doesn't discuss it at all.
I'm not sure people appreciate what would happen if you eliminate the bias. Certain iterations would work better, on the average. Certain identities are preserved. It would make it easier to prove certain relations about iterations.

Say, if you want to show that a certain iteration will ultimately converge.

example: $\sqrt{z}$, $z > 0$

$y_0 = (1. + z)/2.$

$Y_N = (Y_0 + z/Y_0)/2.$

if $(Y_N .EQ. Y_0)$ go out

else $Y_0 = Y_N$

on truncating machine, one thing can happen & something else on a rounding machine.

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

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

in reality, one $>$ sign becomes an $=$ sign & you quit. you've gotten as close as you want to $\sqrt{z}$.

if drift does not exist*, easily prove that this terminates.

also prove that it will terminate for rounding machines.

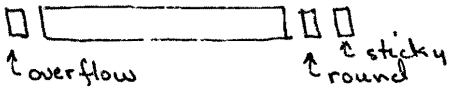
might not terminate for machines which truncate.

Question: Then purpose of sticky bit is prevent drift & that's all?
Answer: Yes, it prevents drift & to make things correctly rounded. If want machine to correctly truncate, would still need a sticky bit. It is not possible achieve the type of rounding desired with only a guard digit. Can add ½ or chop with only a guard digit. Cannot get correctly truncated arithmetic with only a guard digit.

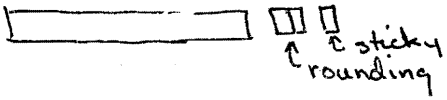
Next time various people will talk about different actual machines & to what extent they satisfy those 6 rules.

Fourth Lecture - Thurs Oct 15, 1970

To tidy up one thing from last time, about the number of bits required. To round to the nearest even, you need only two bits plus a sticky bit. However, in one case (adding magnitudes) you have:



and in the other (subtracting) the bits should be like this:



On hexadecimal machine these three would have to be digits, no bits. (on hexadecimal machine microprogrammed to use short registers, e.g., 360/30.).

Now people will tell us about some machines. I hope they will tell us to what extent the rules are satisfied or not, How big an error is made.

e.g., can they tell us if on C = A + B the machine stores

$$C = (a + b)(1 + \epsilon) \qquad |\epsilon| \leq \text{something}$$

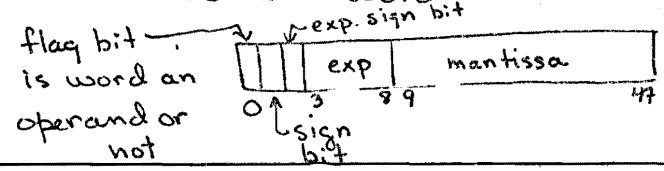
Is this a fair statement or not? Tell us something about the over/underflow characteristics of the machine.

Burroughs B5500 machine.

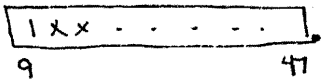
Information from manual did not always agree with that given by Prof. Kahan or by the Burroughs people in Oakland. So some workings had to be guessed at.

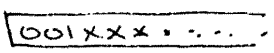
If some of the things told us were right, the manual is visibly wrong.

48-bit word



Representation is in powers of 8 - handled like this by machine. So exponent is 2 octal digits + mantissa is 13 octal digits. Octal point is to the right of the mantissa. A normalized mantissa could be like this:



or in binary 

Exponent is not biased - it is also in sign magnitude, as is the mantissa.

$$-77_8 \leq exp \leq +77_8$$

± 0 are presented, but all operations treat any zero as 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. Two top elements are registers A + B. Operations are performed on these two registers + result is left in B-register. Another register, X, is used to hold shifted out digits + the extension of the result of multiplication + in division. X not mentioned in the manual + its contents are not available to the programmer.

You can't see X but it affects you. You can feel it. There is also an exponent register N.

The rounding rule (not from manual but from users so may not be correct) is that you always round up by 1/2 in magnitude in the last place. You have a 'bias' up.

How addition + subtraction are performed.

If one argument is zero, the other is the answer

If the operands have the same exponent, they are added (subtracted) & answer is rounded up, to 13 digits.

Question: to what extent are single precision arithmetic operations characterized by the rules that Knuth uses - do the operation correctly, take leading 13 digits & round the 14th up.

Answer: It is followed for normalized numbers, in single precision (+ - * /)

Question: is there any case when this isn't true?

Answer: It is not true if the larger operand is not normalized; this can lead to error of 10% according to 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: $0 \cdots 1 \times 8^{0}$
$+ \; 7 \cdots 7 \times 8^{-14}$
} result = 1×8^{0}

whereas the correct result is $1.077 \cdots 7 \times 8^{0}$ which rounds to $1.100 \cdots 0 \times 8^{0}$. More than 10% error.

According to Prof Kahan: if the larger operand is unnormalized, shifted out digits are kept in the X-register (lose only the last 7 in example above). Addition is performed in 26-bit adders; 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 the manual is wrong.

Comments on the six rules:

1. if -x then x : yes

2. except for unnormalized numbers, yes ; no ± 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 Prof. Kahan is right, there is a small discrepancy. If the manual is right, the error is too large to believe.

3. Yes

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 $77\ldots7 \times 8^{63}$?

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

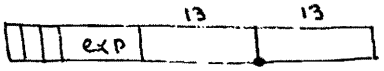
$= 8^{13} \times 8^{11}$

So they should save two characteristic overflow bits. You do not still have the operands - they were on the stack & 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.

Burroughs people were not willing to make the change to use the exponent sign bit for overflow.

5. rounding okay except for unnormalized numbers
6. drift up in that sequence $x_{n+1} = (x_n + y) - y$

double precision:



results in double precision are always normalized but truncated

On multiply, there is a problem only when you get 25 and not 26 digits; then keep only 27 digits as they multiply (don't form the 52 digit product). There can be an error of 1 unit in the 25th digit.

In subtract, keep only 26 digits.

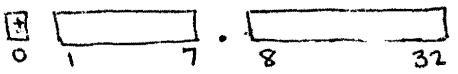
Question: If you have $(A-B) * C$, + $(A-B)$ yields an unnormalized result, is it normalized before multiplication?
Answer: no, multiplication is performed, then the 13 most significant digits are normalized + rounded if possible.

If you multiply integers together, the answer tries to stay an integer, i.e., with zero exponent. There are special rules for rounding them.

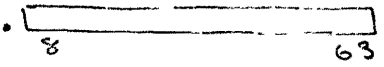
Question: how do ± 0 compare in logical operations?
Answer: they would be different
Question: like if I compare $(x-y)$ with $-(y-x)$?
Answer: You have to distinguish relation operations comparing numbers + Boolean operations on bit stings. In comparisons, ±zero is always zero. There are all the tests, >, ≥, etc. There is no test just for sign except by a Boolean operation.

IBM 360/40

32 bit word



single precision



double precision

It uses true hexadecimal representation

biased exponent by $64_{10} = 40_{16}$

characteristic $0 \leq c \leq 127$, so $-64 \leq exp \leq 63$

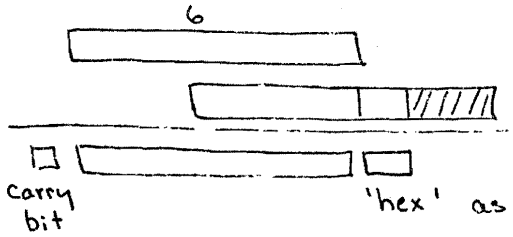
number is fraction $\times 16^{power}$

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

Rounding rule is: do operation to infinite precision, then round to 6 or 14 hexadecimal digits.

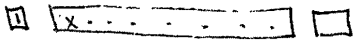
When {adding / subtracting} magnitudes, the rule is

$$z = \text{TRUNC}(x \pm \text{TRUNC}(y))$$

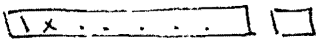


The result is left shifted until answer is normalized. Could be off by almost one hexadecimal digit in the last place

If you get an overflow on adding



result is right shifted by one 'hex' (4 bits)



so you lose 4 bits

Say the bit lost was F ($=15_{10}$). Then the answer is truncated.

maybe have:

1FFFFF|F

Then the answer is just $1FFFFF_{16}$, not 200000_{16} as

might be more reasonable.
If the machine given unnormalized operands, it first normalizes them.

In multiplication, it produces a 14-hexadecimal-digit result (last two always zero) (28 in double precision).
Actual in d.p., it does a curious thing. It gets the 28 digits, truncates to 15, normalizes by left shifting, & truncates to 14 digits.
(A long time ago they used to truncate to 14 hex first, then left shift if there was a high order zero, answer is already truncated.)
You had $1.0 * X \neq X$ because the last hex digit was truncated.
Actually, then never keep more than 15 digits while multiplying. (In the larger model 360's, they do this chopping – then generate the whole product & throw a big chunk away.)

In division, truncation is done properly
Overflow/underflow: in exponent overflow, exponent is small by 128, fraction is correct.
in underflow, exponent large by 128, fraction is correct.
division by zero is suppressed.

360 has a strange thing – over/underflow can cause an interupt; but it can run with the interupt turned off, then a condition code is set, except for multiply & divide. Can get around all this by letting the interupt work but code around 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 exponent is lost on overflow, because can only overflow by 1 bit. Number range (normalized) is not so asymmetric as on B5500.

footnote to page before

(*As an exercise, verify that if 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 normal zero in compare operations
3. exactness preserved
4. can say they don't waste information unnecessarily. but it depends on how you rate how hard that information is to recover.

Question: what about this truncating business?
Answer: in a sense, that is not wasted because you had nowhere to put it.

5. answer won't necessarily be set to a (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:

$$.100000 \mid FFF \cdots F$$

if you throw away that (those) F's, the relative error is

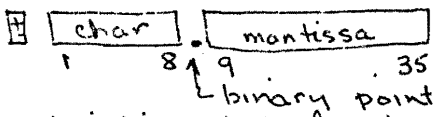
$$\frac{16^{-6}}{16^{-1}} = 16^{-5} \text{ in the last place (almost)}$$

6. sign symmetry 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 words

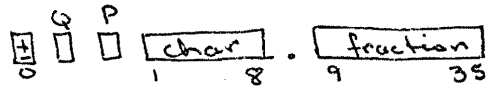


binary machine

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

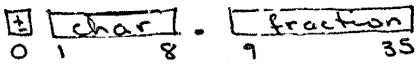
27 bits in fraction.

arithmetic done in an accumulator



Q, P overflow bits

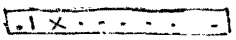
another register which sometimes acts like a right hand extension of the accumulator - called the MQ



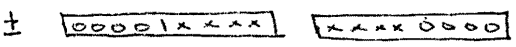
Integers use all 35 bits (are not set up like floating point numbers with zero exponent). Have their own operations.

Add or Subtract operation; different exponents.

puts number with smaller exponent into accumulator. Then right shifts it into the MQ



memory



AC & MQ

perfoms + or -, puts answer in accumulator and normalizes, then doesn't round the answer, altho the information is sitting in the MQ. In normalizing, brings in bits from the MQ. So you have 54 bits when you add or subtract. There is a round instruction; examines highest bit of the MQ & rounds up if it is a 1. Could use Prof. Kahan's rounding scheme here by examining other bits of MQ, but this isn't done. (Couldn't exactly do it that way)

For multiply, you have the 54 bit result in AC & MQ, but you only get the truncated (or rounded) result, as in adding.

In division, quotient is in the MQ & 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 & truncate to so many ~~sig~~ significant figures?

Answer: ~~It's~~ 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 could lead to some sequence that should be monotonic increasing (or decreasing) is not in this case.

Double Precision

AC has high order bits, MQ has low order bits — two words in memory have the other d.p. word.

Add & Sub.

if have to right shift one operand, shifted bits are simply lost. So there are no guard digits.

Multiply

Say words are A & B (in AC & MQ), and C & D (in memory). Leading digits of answer come from $[A \times C]_{\text{high order}}$. Lower order digits come from $[A \times C]_{\text{low order}} + [A \times D]_{\text{high}} + [B \times C]_{\text{high}}$. Can lead to an error of 6 units the last place. Machine truncates on taking $[A \times D]_{\text{high}}$ & $[B \times C]_{\text{high}}$. So could lose -2 in the last place. Lose another 1 (down) in the last place because you ignored $[B \times D]_{\text{high}}$. Then,

if entire answer needs to be left shifted to normalize the result, these -3 would become -6 units in the last place. This has been found to happen by JPL. (Really off by 5.94 in the last place).

Double precision divide can lead to -4 units error in the last place.

Question: If you subtract two different, d.p. words, can you guarantee the result is non-zero? Except for underflow.
Answer: Yes, though not easy to find out.

Question: what if one number is zero & other gets shifted way to the right?
Answer: can't happen as machine would detect the zero & give the other number as the result.

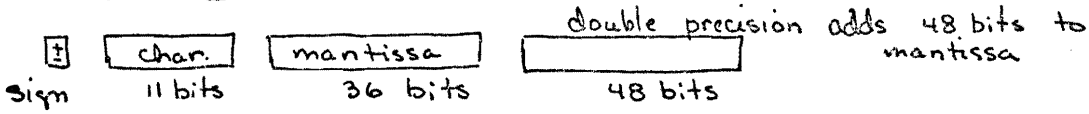
Question: Is any information lost? on over/underflow?
Answer: No, as registers are not cleared on over/underflow.
Also, you go into an interrupt & are given the following information: whether overflow or underflow occurred, in which register this happened, AC or MQ; what operation was being performed, +, -, *, /; the address + 1 is stored.

Can do arithmetic with P and/or Q nonzero, where they act as part of the characteristic. But manual warns that this may give wrong results.

Can get drift because of truncation.

Fifth Lecture – Tues, Oct 20, 1970

BCC Model 1



exponent biased by 2000_8; mantissa is in 2's complement. exponent is not changed for negative numbers. True for both normalized & unnormalized numbers.

4 different kinds of floating point numbers

normal zero: 0 0

normalized numbers: + 0 x···x 1.xx····

− 1 x···x { 1.0····0 or 0.x····x } both normalized negative numbers

unnormalized numbers: + 0 0···0 0.x····x

(smallest exponent) " 1.x····x (sort of normalized)

− 1 0···0 x.x····x

$-\infty$ 1 1···1 0····0

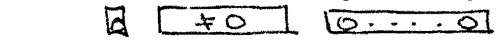
Hardware for machine exists, but not all of the microcoding for floating point & none of it for double precision, or for handling $-\infty$, exists.

Question: Why is the exponent biased?
Answer: Maybe so that zero could be all zeroes, representing the bottom of the normalized & unnormalized numbers.

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. Actually this 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 fl. point number, you get trapped.

Arithmetic

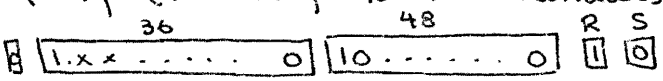
All arithmetic is done in double precision. Accumulator is 84 bits. Result is rounded to 36 bits only when you do a store operation in single precision mode.

Add & Subtract are done with a round bit & a sticky bit.

Question: Aren't there 2 rounding bits?

Answer: No, there's just 1 rounding bit. You can get around this (for subtract where two bits at right may be needed) by shifting left 1 at the beginning & using the overflow bit.

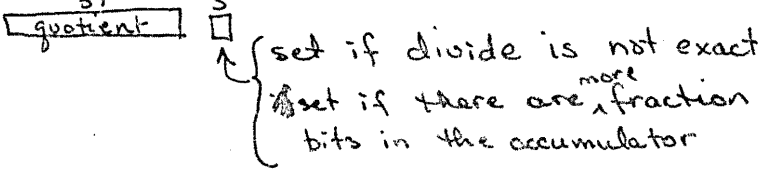
Example where double precision arithmetic & then rounding doesn't work properly (according to the manual):



Round the d.p. word to the nearest even, so R & S are just thrown away. When you try to store this word in s.p., the machine looks only at the 48 bits & rounds to the nearest even in this case (so 48 bits just thrown away) & your answer is off in the last place. This is wrong in the manual. It uses the 48 bits, & the R & S bits to round.

Multiply acts like you expect it to, using the R & S bits.

Divide in single precision (claimed):



Anomoly in double precision divide

Divide 84 bits of accumulator by 84 bits in storage to give an 85 bit quotient, ~~with~~ Should have 1 more bit to be set if division not exact.

The Manual does not treat unnormalized numbers, it just says that they exist.

You can generate them on underflow, & chose to trap on underflow, or get an unnormalized result.

If you trap on over/under flow the mantissa is correct and the exponent is off by 2" (overflow) or -2" (underflow).

There are 5 different rounding modes. & double precision You can select a different mode for one operation & then program reverts to the standard mode (discussed earlier).

Some history of the BCC

The machine is incomplete (in microcoding)

Discussion led to the following:

1) there should be no discernible difference in the way single & double precision rounding are done. Thus, division is a mistake in the design. 1 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 s.p. program to double precision & looks to see if his errors have moved to the right or not.

2) give users access to interval arithmetic - allow user to specify round to next larger or round to next smaller, in value or magnitude as wanted. Thus extra rounding modes.

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 & not when you generate it - then you can overlap operations in the machine & not waste time.

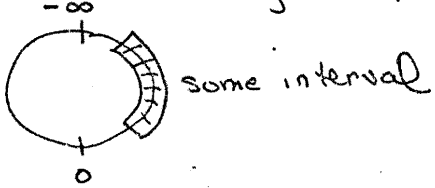
On 7094, for example, Round instruction takes 2 cycles, ordinary Add takes 3, so Add & Round take 5 cycles.

Question: on 7094, is it possible to get the same result in 2 different ways (multiplying), because the binary point is to the right of the first digit? On overflow, that is.
Answer: No, can only get 1 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 & unnormalized numbers: normalized numbers all have reciprocals. But the number [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, & every interval that is representable is an interval on that circle. Its complement is also representable, so that includes only 1 point as ∞.



This machine could be ideal, but that depends on who does the details.

5-5

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, & 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 & multiplied by C. They get around this by always having A+B stored temporarily (by compiler), with store & 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 in time 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 & overflow. The characteristic of the second word is down from the first, so there is a nasty tendency to underflow, & system may be cleared to zero [redacted]. But you might say the problems get worse with higher-precision.

That's true, but most people are content with double precision. Those that want more, are willing to sacrifice efficiency.

CDC 6400

floating point number range:

$3 \times 10^{-293} \leq fl \leq 2 \times 10^{321}$ rather large compared to other machines

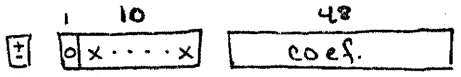
numbers represented as: sign $\times 2^{exp} \times$ coef

coefficient is considered as an integer

$-1022 \leq exp \leq 1022$

$0 \leq coef \leq 2^{48} - 1$ unnormalized

$2^{47} \leq coef \leq 2^{48} - 1$ normalized



exp is 11-bit, 1's complement no., then complement the first bit to bias the exp.

if number is to be negative, the whole word is 1's complemented.

Word is packed so complicatedly so that you could almost take the floating point representions & compare them using fixed point arithmetic, as long as the numbers are normalized & none are indefinite. But you can't do this by using fixed point subtract (it's not really faster anyway)

Floating point operations

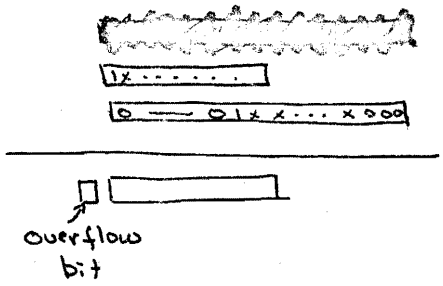
- Add, Subtract, Multiply } truncate, round, double precision
- Divide; truncate, round
- normalize; normalize & round

Talk mainly about 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, & 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 zeroes.

RUN. version of FORTRAN makes sure everything is normalized. Must normalize after add & subtract.

Addition



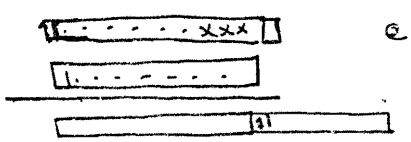
smaller no. is put into a 96 bit register & right shifted - bits fall off the right end.

Truncated add, right bits are dropped. If the sum overflows, one right shift is done.

Actually, 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. Answer may be very small.



exponents differ by 1

↑ this may be the only significant bit & it would be thrown away

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

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

Question: Why is that register 96 bits?

Answer: It is used in multiplication + 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 (+ 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.

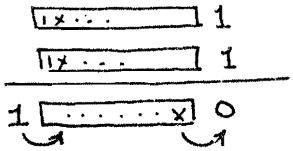
On 650 (to two digits), $100 - 99 = 10$. The ϵ_i's could be made equal, but they'd be huge + you wouldn't want to use them.

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

Rounding Add

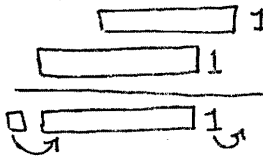
add a 1 bit at the end of each normalized operand (try to normalize the operands), unless you have a 0. (Zero is not normalized as far as this operation is concerned).

case of equal exponents



would add 1/2 in the last place

unequal exponents



could still overflow

{ then adding only 1/4 in the last place makes an error of 3/4 in the last place

$A \oplus B = A(1+\epsilon_1) + B(1+\epsilon_2)$

no overflow $\Rightarrow |\epsilon_i| \leq 2^{-48}$

overflow $\Rightarrow |\epsilon_i| \leq 3/4 \times 2^{-47}$

doesn't lead to so pronounced a drift as in truncation

Multiplication

form exact 96-bit product (may have only 95 significant bits)

if have a leading zero, result is normalized while still in the 96-bit register, then truncate to the 48 high-order bits.

$A \otimes B = fl(A \times B)(1+\epsilon) \quad |\epsilon| \leq 2^{-47}$

Rounded multiply

used to add a 1 to the end of one operand before multiplying, then you could have $A \times B \neq B \times A$.

~~Now, chop off the operation but before~~

~~Chop off~~ Now they add a 1 in the 50th bit instead of 49th, then left shift 1 if necessary.

So they round by 1/4 or 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

0 1 0 . . . . . 0 .

$A \otimes B = A \times B(1+\epsilon) \quad |\epsilon| \leq 3/4 \times 2^{-47}$

not much better than truncated multiply.

Division

truncate exact answer to 48 bits

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

Rounded Division

Appends to the numerator the series 010101....

So they compute $\frac{N + 1/3\,(1 - 2^{-48})}{D}$ & take most significant 48 bits.

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

This assumes that the operands are uniformly distributed between 2^{+47} and $2^{+48} - 1$ & that the part thrown away is also uniformly distributed. Then you get that the mean error, after rounding, is zero. (by hocus-pocus)

Question: Are the operands uniformly distributed?
Answer: I have no idea, but there may be a tendency for smaller numbers to appear.

WHO WANTS TO KNOW IF IT IS ZERO?

WHO WANTS TO KNOW ↑	E ≠0, C ≠0	E 0, C ≠0	E ≠0, C 0	E 0, C 0
ZR	N	N	N	Y
±	N	N	Y	Y
*/	N	Y	Y	Y

→ IS IT ZERO?

E = 0 means smallest possible exponent

Column 3 disappears when unnormalized numbers are demanded. If coef is zero, the whole thing is set to zero by the normalize instruction

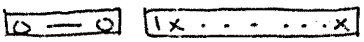
in column 2, it will be set to zero unless it is already normalized. That is, if the normalize box has to shift left, it can't because the exponent is already as small as it could possibly be, so all zeros are entered.

Add box would be happy to add in a number of the 2nd column type.

Sixth Lecture - Thurs, Oct 22, 1970

Still on the CDC

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



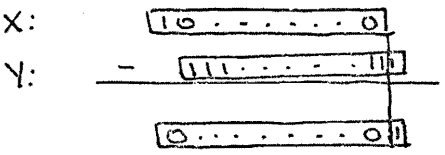
it is normalized since the leading bit is a 1

As far as the add unit is concerned, this number is legal. But the multiplier looks at the zero exponent, & says the number is zero. Thus you can get

$A * 1. = 0$ when $A \neq 0$

If you divide by this number, you get an indefinite answer.

Example of 2 different numbers whose difference is 0.



exponents differ by 1

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 $X = 1.0$
$Y = 1.0 - 2^{-50}$
} FORTRAN VARIABLES

You test for $X = Y$ & get TRUE, implying $X = Y$.

If you had tested for $(X - .5) = (Y - .5)$, you'd get FALSE, implying $X \neq Y$.

Overflow + Underflow peculiarities

If you overflow, there is no trap, but a certain bit pattern is produced, [3777 | x···x], called ∞. When you try to use this number, you are trapped. There is a test for this number, but you'd have to do it after every multiply + 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 + there is no message. Thus, things like the following can happen:

$$\frac{Ax+B}{Cx+D} = 1.0 \qquad \frac{A+B/x}{C+D/x} = 2/3$$

A, B, C, D, x > 0 + normalized

In one case, the numerator + denominator underflow, in the other nothing happens. The point is that on the CDC you have no way of knowing if there was underflow.

Summary of Anomalies

If the machine doesn't do some things they way you'd like, is that bad?

Let's take some of these anomalies + put them into a context to see why they might be bad.

1) You have two numbers, A + B, which are clearly different, but if you ask the machine, it says they are the same, that is:

FORTRAN EXPRESSIONS

A = 1.0···0 × 2⁰ = 1.0

B = 1.1···1 × 2⁻¹ = 1.0 - 2**(-50)

Then if you ask:

IF (A .EQ. B) GØ TØ ... this tests true

IF (F(A) .EQ. F(B)) GØ TØ ... this tests false

F(X) is some simple function, like X - 0.5

If you had both statements in your program, the program would not follow the path that you expected. What significance should the above behavior have?

2) The funny expressions

$$\frac{Ax+B}{Cx+D} = 1 \qquad \frac{A+B/x}{C+D/x} = 2/3$$

These characterize the dilemma caused by underflow messages. On, say 7094, you could get underflow messages for the most innocent things, like in the lower half of a double precision number. People got tired of so many messages & decided to set all underflows to zero. (Actually, this was decided before the floating point units had been build for the 704). That eliminates messages, but the above anomalies can occur.

Question: It has been said that you should never test floating point numbers for equality. You should use an ϵ, typically 4 units in the last place. Even with well-rounded arithmetic, it is possible to get F(A) not equal to F(B), even when they should be.

Answer: I'd like to come back to that. I'll talk about our model of rounding that makes what you said sound reasonable. The Chapter of Knuth, mentioned before, summarizes the idea of axiomitizing just such an idea. But I think that's wrong & will try to prove it later.

Even though you know you shouldn't test for two

numbers being equal, maybe you had a purpose in mind. Maybe A & B are integers, but must be represented in floating point (on CDC) to multiply them.

3) Altho CDC rounded arithmetic is not quite what you'd have expected (since they pre-round the operands), still the error bound is less than on truncated arithmetic. That is, rounded arithmetic error is 3/4 in the last place, while truncated error is 1 in the last place.

But this leads to an interesting & subtle anomaly. After careful analysis, in a program you discover that there are two numbers in the machine such that

$$AB = CD \quad \text{exactly.}$$

(say two factorizations of some integer).

Then you discover that truncated FORTRAN arithmetic does give you that

$$A * B = C * D.$$

(integers or anything else). A, B, C, D are bit strings & the products are equal in all 95 or 96 bits.

But, this may not be true for rounded FORTRAN arithmetic. In truncated arithmetic, the most significant 48 bits are taken as the answer, & since the numbers have all bits the same, the answer is the same.

In rounded arithmetic (BASIC & MNF on the CDC), however, there is that extra bit in the 50th place. Then, if 1 product needs to be left shifted 1 place to normalize, you round by 1/2 in the last place. But if the result is already normalized, no left shift occurs & you round by only 1/4 in the last place. Hence you could have:

$$A * B \neq C * D \quad \text{(rounded)}$$

Are these things important enough to want to change them?

The CDC makes another type of alteration in your expectations with respect to overflow & underflow. With overflow, you have no way of knowing if you overflowed by a lot or just a little.

CDC overflow rules. There is a sign preserved, so you know if you got plus or minus infinity ($\pm\infty$). Once ∞ is in there, usually you will be kicked off if you try to use it. But you can suppress the trap, & then the following rules apply:

$\infty + \infty = \infty$

$\infty + x = \infty$ x is any real no., not ∞

$\infty - \infty = ?$ (indefinite, sort of $\frac{0}{0}$?; you don't

$\pm\infty * \infty = \pm\infty$ really deserve an answer, but

$\infty/\infty = ?$ you can use it & keep

$\pm\infty/\pm\infty = ?\infty$ generating indefinites)

$\infty * 0 = ?$

$0.0/0.0 = ?$

$f_{\infty}(?) = ?$

$\boxed{x/\infty = 0}$ interesting rule

Could you countenance operating without the traps, just letting the machine run, producing infinities & indefinites? But you might end up printing out lots of indefinites, & infinities, & maybe you'd have lost nothing but time.

But the rule in the box above is another matter. It allows you to generate incorrect answers.

I presume that $X/Y = 2^K X/2^K Y$, or I should be told about it (say $X < Y$). with the trap

off, as you increase k, there is a value for which

$$\frac{2^k X}{2^k Y} = 0$$

the next time you increase k, the answer is indefinite. This may look artificial, but look at this example. You are positive $0 < X < Y$. You have worked to insure this. Therefore, you expect

$$\frac{1}{2} \leq Y/(X+Y) \leq 1$$ (the = part allows for rounding errors)

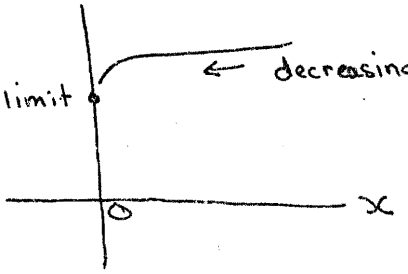
But with overflow, you would get the result of zero, which is not even in the range of answers expected.

There are more errors in some of the subroutines, & then hardware & subroutine can conspire together against the user.

EXAMPLE OF MACHINE CONSPIRACY

Now we consider an example of such a conspiracy on the 7090 & this leads directly to why such things are bad.

A graduate student had developed a marvelous idea for boundary layer control on wings of short takeoff planes. He thought lift should be enhanced by his idea, & had set up the appropriate differential equations to check his ideas. Altho he couldn't solve them analytically, he had some information about how it should behave & approach a limit as the independent variable went to zero.



A limit was not calculable and approach may not be smooth.

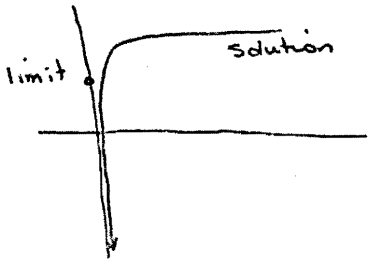
(like $1 + \sqrt{x} \to 1$ as $x \to 0$

or $1 + \frac{1}{\ln \frac{1}{x}} \to 1$ as $x \to 0$)

He could n't show analytically that the limit existed so he turned to numerical methods.* (*You solve a differential equation by breaking the space up into discrete little chunks, & talk about functions with a slope in those intervals. The graph is replaced by a sequence of dots, & this is justified if you can show that as the mesh gets smaller, the dots approach a continuous curve that is the solution.)

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

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



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

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

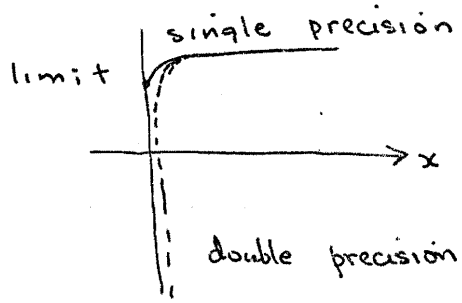
By now he had used up lots of machine time, & seemed at the end of what otherwise might have been a promising Ph. D. thesis.

At this time I was trying to debug a logarithm subroutine,

used to calculate A**B by taking exp(B*Alog(A)). For some values of A + B the results were shockingly less accurate than others.

Looking over his shoulder, I saw he was using a logarithm routine, + suggested he use mine, which was more accurate than the old. How much more? My error was about .52 units in the last place + the old one's error about 3 units + other interesting difficulties.

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

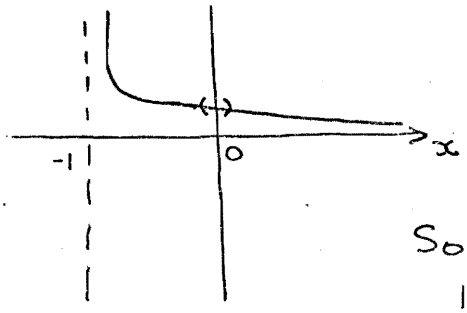


At that point I became interested. He was interested, of course; the s.p. answer said he'd get his Ph.D., the d.p. that he wouldn't.

He was trying to compute something like

$$f(x) = \frac{\ln(1+x)}{x} \qquad -1 < x < \sim 10^{-4}$$

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



Strictly speaking, the point at $x = 0$ is missing.

So use a power series = $1 - \frac{1}{2}x + \frac{1}{3}x^2 + \ldots$

for $-1 < x \sim 1$ in this range.

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

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

he's thinking:

```
X = .0000XX|XXX
Y = 1.0000XX|
Z = .0000XX|
```

so y = 1+z, not 1 + x

if x is small, I've made a rounding error, + I won't take the log of 1+x but the log of 1+(something else) then in dividing by x, I get the wrong thing. So I'll take the log of 1 + something + divide by that something. And since my function is continuous + well behaved, I'll be near my point x + be on the graph of the function.

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

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

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

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

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

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

But $\sqrt{2}$ is not representable by a machine no. So when $\sqrt{2}$ is off, you are in effect changing f from what is was to something else. And f is changed differently in numerator + denominator.

You are calculating ALOG(F(1+ε)) not ALOG(F)
But he wanted to compute this for F close to 1. (1+ε) is also close to 1, & so their logs are comparable. So in single precision things went wrong in a systematic way & drove his graph down. When he used my program, which didn't do this, the graph straightened out.

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

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

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

ECONOMIC COST OF ANOMALIES

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

I don't object because they violate mathematical aesthetics.

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

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

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

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

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

footnote

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

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

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

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

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

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

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

Question:

I had worked on a numerical subroutine for solving D.E. People using it would typically chose a step size & then one 10 times smaller to see if that changed the results. But the program kept blowing up mysteriously.

It appeared that when funny things happened, they were really funny. The error propagated in rather impressive ways.

Answer:

You're saying that errors will be accompanied by symptoms so obvious that one could hardly fail to notice them if he was at all conscientious. ? found a 2×2 matrix which, when put into this iterative solver gave what looked like a correct solution (all tests were satisfied), but not even the leading

In response to the claim, made by the author of a matrix equation solver, that the amygone's matrix that wasn't solved by his routine was so unlikely that field already been run over by a truck,

digits in that solution were correct. I don't think people know when an error is committed.

There are times when in solving continuous functions the intermediate results may be discontinuous. And these discontinuities may be important & you'd like to be told about them. So you depend on laws of arithmetic that may not be honored by your machine. There are FORTRAN programs that run on a 7094, 7090, B5500, & G.E. 645, but they will not run on a 6600, & no one has found out why.

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

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

LIVING WITH THE CDC

How do you live with the CDC, since it is the way it is? How to do calculations & estimate the cost of rounding errors?

Take the FORTRAN statement: $C = A \circledast B$

$c = (a(1+\alpha)) \odot (b(1+\beta))$ (what happens in the machine)

We would like to be able to say that what is stored in c comes from doing the operations with operands that were perturbed just a little bit. If you can say that, you have a beginning at least of an error analysis.

Consider the foregoing operation, & the more nearly ideal one:

$$c = (a \circ b)(1+\gamma)$$

where the operation is done exactly & then rounded.

The difference in many cases may be small. In $*/$, the difference can't be important. But in $+-$ there can be an important difference.

ERROR & UNCERTAINTY

Error is the difference between what you got & what you want. And uncertainty is how big you know the error isn't. Uncertainty is a bound on the error. And when I say a number is uncertain, I mean it is a sample from a set of numbers that are practically indistinguishable as far as you are concerned. The size of that set is a measure of the uncertainty in the number you've written down.

Let's say you want to add a sequence of numbers obtained from other machine calculations.

The numbers you have in the machine are a_j, & the numbers you wanted are $a_j(1+\delta_j)$, $|\delta_j| \le \delta$. You have a bound on the uncertainty in the a_j.

When you add, you get the following:

$$\left[\{a_1(1+\alpha_1) + a_2(1+\alpha_2)\}(1+\beta_2) + a_3(1+\alpha_3)\right](1+\beta_3) + a_4(1+\alpha_4) \ldots$$

You get errors from picking a number up from storage (α_i) & then in doing the addition (β_i).

Rearrange into:

$$a_1(1+\alpha_1)(1+\beta_2)(1+\beta_3) + a_2(1+\alpha_2)(1+\beta_2)(1+\beta_3) + a_3(1+\alpha_3)(1+\beta_3) + a_4(1+\alpha_4)$$

if α's & β's $< \delta$'s, most of the errors is not from roundoff but from uncertainty in the data.

Thus you get the philosophy that if your data is only good to 20 or 30 bits, [redacted] rounding errors from 48 bit word lengths will be negligible.

You'd have gotten a scarcely better result if you had the other model of rounding.

But what if you are solving a D.E. by adding smaller + smaller increments?

$$\text{new } y(t+\Delta t) = \text{old } y(t) + [\Delta t \cdot F(t,y)]$$

$y(0)$

$y(1) = y(0) + I(0)$

$y(2) = y(0) + I(0) + I(1)$

What's wrong with the COC adder for this problem?

$(f(n)+\epsilon)^2(1+\epsilon)$

$(f^2(n)+2f(n)\epsilon)(1+\epsilon)$

$[(f^2(n)+2f(n)\epsilon+\epsilon^2)+1](1+\epsilon)$

$(f^2(n)+1$ [illegible]

Seventh lecture - Tues Oct 27, 1970

Here's an example, in a decimal arithmetic, utilizing COC's method of prerounding, in which

A * B ≠ C * D

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

A = 45 B = 19

```
  45
 x19
0855
  05
0860
```

→ 860 as the answer.

C = 95 D = 9.0

```
  95
 9.0
8550
  05
8555
```

→ 850 as the answer

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

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

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

ADDING MANY SMALL NUMBERS

Let's Returning to the discussion on adding long strings of numbers. The COC philosophy seems to be that since your data is probably uncertain, any errors committed

in adding (subtracting) are insignificant. The sum is no more wrong that if you had altered some of the operands by 1 in the last place at worse.

This sounds okay. But consider the example of solving the diff. equation

$$\frac{dy}{dt} = f(t,y)$$

by performing

$$y(t+\Delta t) = y(t) + \Delta t \cdot F(t,y)$$

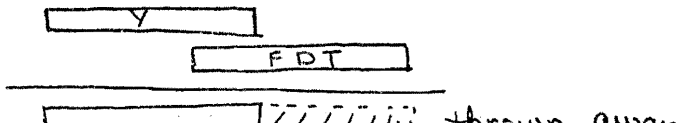
F is closely related to f

You can calculate the last product to a few units in the last place.

What will happen if the only rounding errors you commit are those to make $\Delta t \cdot F$ uncertain in its last place + in doing the addition?

Take the FORTRAN program

```
      Y = Y0
      DO 1 I = 1,N
      FDT = ....
    1 Y = Y + FDT
```



large rounding error when FDT << Y

Throwing away the right hand digits of FDT is equivalent to only a small change in Y, but it is making a rather drastic change in FDT. If Δt is small, you'll have to increment many, many times, say 10^6. Then the error in Y, in those 10^6 steps, is roughly $\frac{1}{2} \times 10^6$ units in the last place. The situation gets worse when most FDT's are subtracted, + the error could be a good deal larger.

Even if you know Y0 quite accurately, + the diff. equ. is quite stable + well-behaved, you can find the bulk of your error in that addition step. You only see it because you accumulated so many of these little errors.

The situation is not any better using correctly rounded arithmetic.

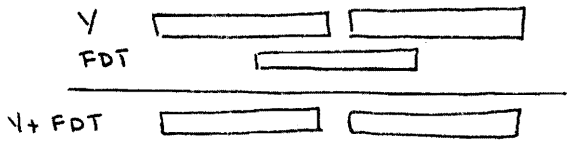
On the CDC, with 14 decimal digits, losing 6 of them may not be too drastic. On the 360, in short word arithmetic, you could lose all 6 of your hexadecimal digits.

But consider the man who wanted all those digits. He may want to know how sensitive his design is to parameter variation. If the variation is in the fifth place, he thinks has 8 digits to play with. But if he's really only got 2 that are perturbed (8 - 6 uncertain ones), he may not trust his results.

It is hard to know when this behavior will be disconcerting. So the man who writes a D.E. solver will try to save his users from this difficulty.

The writer may merely decide to keep the Y values to double precision.

Now we have



Rounding errors at the end of D.P. Y are negligible compared to uncertainty in FDT.

For stable diff. equ. the answer can be accurate to almost a full word; any uncertainty is due to uncertainty in calculating F(t,y), not to rounding errors.

HOW TO WORK WITHOUT DOUBLE PRECISION

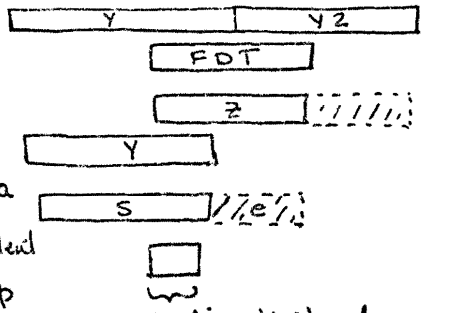
What about a machine that doesn't have double precision? E.g., working to double precision on a 360 (14 hex characters or roughly 56 bits) is not much different from CDC's s.p. 48 bits.

You represent Y by two words, the leading part of which is the single precision value + the second is the trailing part.

start by saying $Y = Y0$, $Y2 = 0$ } Y+Y2 is what will have meaning for us.

Y2 is generally small compared to Y, like a few units in the last place of Y.

```
      Y = Y0
      Y2 = 0
      DO 1  I = 1, N
      FDT = ....
      Z = FDT + Y2      (committed a small error here, equivalent to messing up FDT by 1 in the last place)
      S = Y + Z         S = y + z - e
      Y2 = (Y - S) + Z  Y2 = (y + z - e) + e = y + z
1     Y = S             y = s
```



This process works on machines that normalize sums & differences, before they chop the answer, or round the answer. So it works on all 360's.

So it fails on the CDC, which does not normalize sums & differences. It fails because (Y-S) is not computed exactly, even tho they differ by only a little (roughly by the leading digits of z) & could only have been shifted one or so places right or left. Y2 is not what you want it to be. There you are, blasted.

(also on the IBM 650, 1620, UNIVAC 1107, CDC 3600).

If this trick were needed only to solve differential equations, it would not be worth crying over its loss. You then would write a double precision subroutine, in assembly language if necessary, & call that to add your numbers in double precision.

WHY YOU WANT EXACT DIFFERENCES

But this reaches into many other areas. It affects our ability to code higher precision arithmetic out of single precision by subroutines that are partially machine independent. This may not appear important to you, but when people produce

numerical algorithms they'd like them to work on any reasonable computer. In the middle they'd do calculations to essentially higher precision by some trick. The writer could insist that your compiler provides double precision. But Algol usually doesn't (*except on the B5500).

But there is a nontheorem that tells you there is no theorem to tell you that if you want to solve this problem to single precision you must carry n-tuple precision. There cannot be such a theorem to specify n, since n-tuple precision can be simulated by single precision (*A Report by T.J. Dekker shows how to do this on 'clean' machines, that is, on machines on which the preceeding trick will work. There is a book in manuscript by Patrick Sterbeng in which he also shows how to code double precision arithmetic from single, provided the machine is reasonable, like 360 equipment. To look at these two papers, see Prof. Kahan. Knuth, Sec 4.2 also talks about this, * even * has the trick enshrined as a theorem.)

footnote

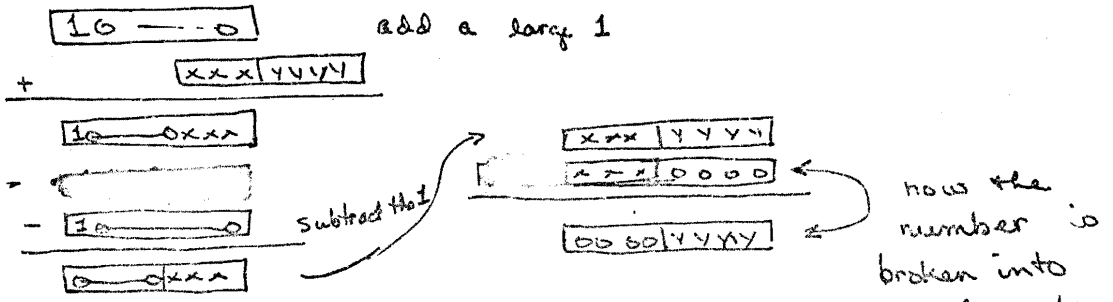
People who talk about coding multiple precision with single generally miss a couple of points. One is that they insist upon being able to compute Y + Z exactly, for all combinations of Y & Z as the sum of two other floating point numbers, say Y + Z = S1 + S2, where S1 + S2 almost constitute a double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

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

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

If you can get a difference exactly (if it is representable exactly), then for all the kinds of arithmetic we've been studying you have the equipment to do double precision arithmetic, coded in FORTRAN or ALGOL, using only the ordinary floating point arithmetic. Then you can pyramid. And the code is transportable to another machine. You need only verify the most rudamentary aspects of the machine, like its number base, number of digits carried in single precision. If you work at it long enough you can get operations to be performed exactly. (702-11)

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



If you can do this, the rest is easy.

Treatment of Under/Overflow

The current treatment is based on an attitude which says that if overflow occurs, you have made a mistake. This attitude leads to having unpredictable results if overflow occurs.

Consider how few ways you could respond reasonably to under/overflow. One reasonable way is just to consider it an error you can't recover from. This looks okay until you make it extremely difficult for anyone to prevent such

errors. Suppose an answer can only be obtained by going through under or overflow. Here's an example: finding the zeroes of a polynomial $f(z) = \det(A(z))_{1000\times1000}$ (or $10{,}000\times10{,}000$) then $f(z) = \prod_{1}^{1000} u_{jj}$ after A has had gaussian elimination applied to it.

If the numbers u_{jj} are not extremely close to 1, like 1/4 and 4, you run the risk of getting under or overflow. You might suggest scaling the matrix, say by dividing each element by 2, thus taking out a factor of 2^{1000}.

$$f(z) = 2^{1000} \prod_{1}^{1000} u_{jj}/2$$

This would work on CDC equipment. But if the dimension is 10,000, this trick fails.

The problem is this: is there a way to find out if under or overflow is going to occur + respond to it, even if it is only to scale by 1/2 or 2. You'd do the calculation, and if overflow occured, do the scaling and recalculate.

So you test for an infinite operand in each loop of the calculation

Loop $P = P * U(J,J)$
is $P = \infty$?

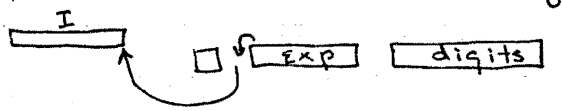
This is typical of situations where you must find the overflow immediately, if at all. If you don't test, on the next loop you get kicked off for picking up an infinite operand. In this case, putting in the test isn't much additional work, since you have so much work in computing the u_{jj}'s.

There is a problem though, in that the final value of the matrix may be very small (you are looking for a zero), because the last u's are tiny, but you still might overflow in the middle of the multiplying. So you scale everything down + then at the end you underflow to zero. And since zero is what you are looking for, you say z is a root. So the

ONE WAY TO HANDLE OVER/UNDERFLOW

problem is not merely testing for overflow in a loop.

Here's a way I devised to cope with this on some machines. Designate some register I as a leftward extension of the characteristic of the one accumulator register.



When the exponent overflows or underflows, put that digit into I. Do the calculation. Test on I at the end only. If $I = 0$, the accumulator has the answer in it. If $I \neq 0$, then the true result is: $P * 2^{256 * I}$. How could this result be used? I was only interested in ratios of f's, + would stop when the ratio was sufficiently small. This shows how you could cope with over/underflow, if the machine doesn't obliterate results if o/uflow occur, but provides an interrupt.

This type of coding is only applicable to problems of the type:

$$\Pi\left(\frac{a_i + b_i}{c_i + d_i}\right)$$

If the numbers used here had come from a nested calculation of this type, you could get overflow in I.

Problems to which this is applicable are: hypergeometric series, + probabilities.

This example was not to say that all overflow/underflow should be treated in this way, but to show that there are reasonable responses which are not simply to abort. And provided a test to tell if o/underflow occurred, but not what happened, may not be enough.

Consider: how to compute geometric mean in FORTRAN?

Mean $(a,b) = \sqrt{ab}$ $\quad a,b \neq 0$

You can get around o/uflow by taking two square roots.

But can you get around this by taking only 1?

In the problem of the quadratic - what happens if you get over/underflow in the middle of the calculation in spite of the fact that the roots are representable? Then consider some of the facilities mentioned + what would you do with them if you had them? (You do not need the leftward extension).

7Q1 Question: In trying to simulate the double precision, wouldn't it be better to unpack the numbers and work on them as integers?

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

7Q2 Question: what's the good of a FORTRAN program, if you have to rethink + reprove that the program will work when you go to a different machine?

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

7Q3 Question: With that length, isn't it still better to work with your own number representation?

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

7Q4 Question: But why not use integer arithmetic in FORTRAN, if you're going up to 100-length words?

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

7Q5 Question: Your code is transportable only with some assumptions about the machine. And you haven't stated what those assumptions are.
Answer: the assumption would be that whenever they do an arithmetic operation the result is no worse than what you would have gotten had you changed both the operands by a unit in the last place.

7Q6 Question: You've drawn pictures of how the numbers appear in the machine. What if they don't appear that way, but involve lots of funny shifts?
Answer: the algorithms would work, but the proof gets harder.

7Q7 Question: It seems you're assuming more than that the result you get is no worse than making a slight modification in the operands because you keep making statements that 'this calculation can be done exactly', but that wouldn't follow from your original assumption.
Answer: No, the original assumption is that if two operands have sufficiently few digits, the operation is done exactly (the digits line up). There are some numbers for which you get what you should, in the absense of rounding errors. Another assumption is the one about modification of the operands. The tricks are designed to work toward the numbers with few digits, where you can do something exactly, & then work your way back to the original problem.

7Q8 Question: In the time you spent working out this trick in FORTRAN, it could have been done in machine language for 5 different machines.
Answer: You could, but the trick, in FORTRAN, can be imbedded in code & carry the code wherever you want to. any machine as long as it isn't too wierd.

7Q9 Question: Are the properties you require (for the tricks) going to be easily determinable for any machine?

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

7Q10 Question: If what say is true, about being able to devise a trick for even dirty machines, why should Knuth, Dekker, + Sterbenz continue to lay out red herrings?

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

7Q11 Question: By the time you find someone who knows enough about the machine to answer your questions, you could have coded it in machine language.

Answer: That I dispute. Consider the B6500. We know what the characteristics are: 13 octal digits characters with such + such arithmetic. We know that now, but not the order code. I would be easier now to code the FORTRAN rather than learn the order code. Or say, in a hurry, I want you to produce roughly quadruple precision add. You'd find it faster probably to take the double precision add + trick it, even on a machine whose assembly language you are utterly familiar with.

Eighth Lecture - Thurs. Oct 29, 1970

It was claimed that on the CDC, you could have two factorizations of the same number +, because of the way CDC does its rounded multiply, you could get two different answers. Here are such numbers, in octal:

$A = 17747\,\overbrace{0\cdots0}^{12}\,500$

$B = 85_{10} = 1726524\,\overbrace{0\cdots0}^{13}$

$C = A * 5 = 200043\,\overbrace{0\cdots0}^{11}\,310$

$D = 17_{10} = 172442\,\overbrace{0\cdots0}^{14}$

$A * B = C * D$ in truncated arithmetic

$A * B \neq C * D$ in rounded arithmetic (differ by 1 in the last place)

$A*B = 200445140\,\overbrace{0\cdots0}^{9}\,324$ } rounded

$C*D = 200445140\,\overbrace{0\cdots0}^{9}\,325$ }

~~I don't have to go back to the second question at all~~
~~Maybe I can ... the question if it is possible~~
~~to compute a difference exactly on machines such as CDC.~~

COMPUTING DIFFERENCES EXACTLY ON MACHINES LIKE CDC

PROBLEM to compute $y = s - x$ to within a few ulps (units in the last place) and exactly when possible, assuming $s/x \geq 1$ and using only single precision, machine independent FORTRAN. We shall ignore over/underflow. I don't claim this program is a solution. In 1960, when it was written, it seemed to be a solution for all the machines we knew about (EDSAC II, IBM 650, 7090, CDC 3600). Then we'll talk about for what machines is it valid + therefore to what extent is the code machine independent.

Let's see why this problem exists.

on CDC type machines. $1.0\cdots0$

$$\underline{-\ .1\cdots11}$$

$0.0\cdots0|1$

answer = 0 ↑ thrown away

on IBM in long word arithmetic

```
  1.0 .... 0 |
- .F .... F | F   <- dropped (no guard digit)
-------------|
  .00 .... 1 |
```

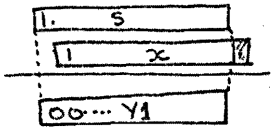
answer should be .00....0 1, only 1/16 as big

FUNCTION DIF1 (S, X) $S/x \geq 1$

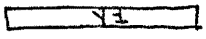
Y1 = S - X not correct to 1 ulp on some machines

$y_1 = s - x - e$

X1 = S - Y1 exact! $x_1 = x + e$



$s + y_1$ will line up so the subtraction is exact

(on CDC, the extra digits of x are thrown away after subtracting; on IBM 650, they are thrown away first: you get the same result; on IBM 360, with 1 guard digit, you get  . But when you subtract, it is still exact).

The surprise is that you should have an operation exact on any of a large number of machines.

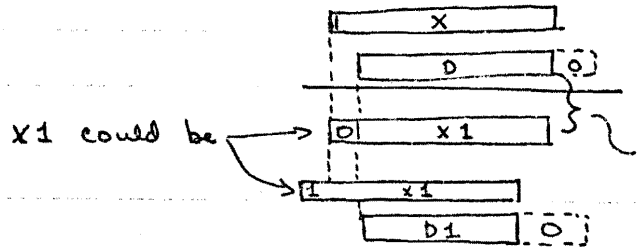
Now we do a fudge. In the cases that are interesting to us, X1 + X are almost equal. If Y is representable exactly, we want to compute it; if it is, it means S + X are not many orders of magnitudes different.

D = X - (x - 0.1 * X) The constant 0.1 is not important — on a binary machine a power of 1/2 might be better.

$d \simeq x/10$ in the cases that interest us: (D - X) will be exact(!) for the same reason that (S - Y1) is exact.

8-3

D1 = X1 - (X1 - D) $d_1 \approx d$; $(x_1 - d_1)$ is exact



bottom digits of D have been cleared to zero

will mix correctly

D1 is D with (possibly) a few more digits removed

$D_1 \ominus X$ will subtract exactly.

E = (X1 - D1) - (X - D) this can be computed exactly when it matters (when X + S are very close)

DIF1 = Y1 + E

RETURN

END

$y_1 + e$ will fit into frames with the same characteristic if y is representable exactly. If y is not representable exactly, there'll be a rounding error here, but that doesn't bother me. (8Q1)

The modest object for this code, remember, is that if the number is representable exactly, let's make sure we get it exactly. If it is not representable, let's just get it to within a couple units in the last place. I'd like to be able to declare everything here to be double precision + still be able to say the same things for double precision numbers. That's harder, because the variety of d.p. codings is much greater than the number of machines. This code is very inefficient, but that, as you will see, is irrelevent. (8Q2)

PROBLEM to find single precision numbers S1 + S2 such that $S_1 + S_2 \doteq s + x$, given s + x to single precision, and such that

$$S_1 \doteq s + x \quad \text{to single precision}$$

(this was the essence of the problem when incrementing the dependent

variable in solving a differential equation).

$(S/X \geq 1)$

S1 = S + X $S_1 = S + x + e$

S2 = X - DIF1 (S1, S) $S_2 = x - (S_1 - S)$

$S_2 = x - (x + e) \approx -e$

To see which machines this will work for, consider the 360 — the second line becomes S2 = X - (S1 - S) : the subroutine DIF1 simulates the 360 operation, for certain operands. S2 is the digits that were thrown away from X+S.

→ Could you obtain the rounded value of a sum from a trick like this?

WHY ARE EXACT DIFFERENCES IMPORTANT

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

footnote

I don't think being able to code double precision is a very desirable thing in itself, tho people have worried about it. * footnote from here

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

details of each machine that he runs his code on.

That's only one problem. That problem collides with another problem - really a different approach to the same ultimate dissiteratum: Is numerical analysis a science? Or is it just an art? It was taught to me as an art. My professors did not think of it as mathematics. To think of it that way is really a step backwards, since all the old classical mathematicians, Euler, the Bernoulli bros., Lagrange, LeGuerre, thought of mathematics & numerical analysis as practically synonymous. They didn't distinguish the two. But subsequent mathematicians did. Math. was regarded as a simplification of real life & numerical analysis was a compromise.

NUMERICAL ANALYSIS vs. MATHEMATICS

If we cannot prove anything about numerical analysis, then we run the risk of never being able to prove anything worth knowing about computers. You must distinguish between num. analy. & math. in which there are infinite processes & in which things are alleged to converge until you start to discuss rounding errors as they really are, & under/overflow as they really are, you don't have num. analy. Van Wijngaarden thinks this way too; in '66 he published a paper "Numerical Analysis as an Independent Science" & Knuth has condensed Van Wijngaarden's many pages into about a page (Van Wijngaarden has numerous rules for entities which would be represented in the machine & would be intended to represent real numbers. These are to supplant the rules we learned about real numbers. But since most people don't understand this much smaller set of rules, what are the chances of re-educating people with Van von Harden's? He tries to skirt around another problem, that of using one symbol to mean different things. He would like his rules to hold if you replace each number by a set of numbers that differ only by a few units in the last place. You should make only those statements that will remain valid if the operands are perturbed before the calculation is done.

footnote

Questions like, are two numbers equal, he thought you ought not to ask. You ask if they are equal to within a tolerance, which is tantamount to saying that the equal sign represents an operation; you perform this operation upon two operands & the result must be independent of what you would get had you perturbed the operands by at most a few units in the last place. Two numbers may be equal, to within a tolerance, or definitely different to within that tolerance, with some borderline areas in between. Knuth discusses these notions. He has $a = b$, $a \approx b$ (a almost equal to b), & $a \sim b$ (a not quite as equal to b as that). Philosophers & other people would sum this up by saying — if you do that (end footnote) you will be unable to say what you mean or to mean what you say.)

Knuth's approach is to say let's discuss what is a reasonable model of what is done in a computer (or could be) as merely a small distortion of what would happen in the world of real numbers.

$$A + B = a + b \quad \text{which gets rounded}$$

That is a very simple rule. Unfortunately computers don't obey it But Knuth does have the beginnings of a science. Knuth has compromised a bit, as we could go further & describe all operations in terms of integers. Knuth has done this, by writing programs for MIX, an integer machine. He says these will be the definitions of the floating point operations.*

(* footnote Anything that is not implied fully by the rule above will have to be ferreted out by looking at the integer manipulation. You have to look at exactly what is that rounding rule & exactly what is the base of your machine. Knuth says he doesn't care what the base is — a byte can hold 64 or 100 possibilities. In my experience that is a disaster. All sorts of ugly things happen to non-binary machines.) (end footnote)

So there you see the two problems converging. One is —

can you write machine independent code. The other is - can you think of numerical analysis as a science? If you claim to have machine independent code it means you have proved something about it which is tantamount to proving a theorem about an algorithm, & that's the type of thing we want to do in numerical analysis.

A THIRD CONSIDERATION

There is really a third leg on this stool. If you lose any one of the three, the stool will fall over. The third leg is this: can you prove theorems about num. analy. comperable to theorems in computational complexity, but bearing instead on how much precision you have to carry to do a certain job. There are theorems about how long it takes to do operations - say multiply two numbers together. (* To add two numbers of length n in machine whose components only have a certain complexity takes on the order of $\log n$. Machines currently do run close to the optimum, which is nice. A lower bound for multiplying is similar, but there are no algorithms that really come close; usually $n \log n$ is more realistic. So there is room to improve multipliers. Or the lower bound and.)

Belaga-Pan There's a theorem by due to (also in Knuth) which tells you that if you have an n^{th} degree polynomial, you can, by rearranging it in various subtle ways, reduce the number of multiplications by a factor near two. The theory doesn't tell you how many digits you'll need to carry, however. How long does it take to multiply two matrices together? The conventional way requires n^3 (for $n \times n$ matrices) multiplications. Winograd showed that roughly half that many will do. Strassen has shown that actually it is $n^{2.xxx}$ where the exponent is $\log_2 7$ instead of $\log_2 8$. Other theorems say how much storage you need to do something. But there is a shocking lack of theorems that tell you how accurately

you have to do something. These theorems don't exist because in principle you could code multiple precision using only single precision arithmetic. And that's the explanation for wanting to do some of this, just to demonstrate that if you had to do it by brute force, you could do this coding, & it would be to some extent machine independent. In the absence of definitive rules on how machines work, it is hard to say just what that code should be like.

I'm going to show you a calculation that doesn't have an iota of this kind of code. In this calculation, it'll look like we ought to be using triple precision, but seem to get away with roughly double. It seems to violate a very tempting theorem! This problem will arise when somebody delivers an algorithm that is very much more accurate than you would have expected it to be on the basis of unproved folk-theorems. If you think this is unrealistic, just remember that there is no reason to give error bounds if they are so drastically unrealistic that everyone ignores them.

footnote ?

THE CUBIC EQUATION

The special problem is: how shall we solve a cubic equation.

$$Q(x) = a_0 x^3 + 3a_1 x^2 + 3a_2 x + a_3$$

The a's are given precisely as floating point numbers in single precision, & they are moderately large.
In worst cases, it would appear that you need triple precision to get single precision answers. Why is that so?

Imagine $Q(x) = a_0(x-\xi)^3$ triple root at ξ

In the course of the calculation, you are going to commit some rounding errors. On the CDC, you could conclude that you wouldn't compute the roots of this cubic, but of another very close to it.

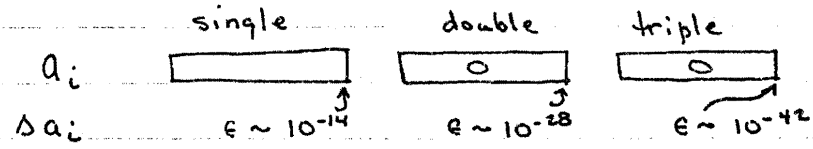
You may actually compute:

$$(Q + \Delta Q)(z) = 0$$

where $(Q+\Delta Q)(x) = \sum (a_i + \Delta a_i) x^{3-i}$

(each coefficient has been perturbed by a little bit.)

The perturbation should be comparable to the precision used. That is, if a_i is s.p., Δa_i is an error in a_i's last place. *



(*footnote The reasonableness of this you should read about in Wilkinson, Rounding Errors in Algebraic Processes

He shows that, using the floating point arithmetic you are already accustomed to, almost every calculation you would want to do with this polynomial will give you instead, among other things, a result that corresponds to having fudged the coefficients by a unit or two in the last place in any precision you are carrying. Say you wanted to compute the value of $Q(x)$. Then you'd write

$$((a_0 x + a_1) * x + a_2) * x + a_3$$

If you look at the last operation alone – it takes the previous rounded number, adds a_3 & rounds; that's like perturbing a_3. You attribute the errors in rounding to having perturbed the coefficients, and say you have exactly the right value of slightly the wrong polynomial.)

HOW MUCH ACCURACY IS NEEDED

At best, you have just computed the wrong polynomial. Actually, things are worse than that.

$$Q + \Delta Q = a_0 (x - \xi)^3 + \Delta Q$$

Let's assume the perturbation is due merely to having fudged a_3.

$$\Delta Q = a_3 * \epsilon$$

ϵ will tell us how many figures to carry.

$Q + \Delta Q$ has different zeroes from Q.

8-10

$Q + \Delta Q = 0$ when $x = \xi + (-a_3\epsilon/a_0)^{1/3}$

because $-\frac{a_3}{a_0} = \xi$

we get $x = \xi(1 + \epsilon^{1/3})$ *

footnote (* When you write $\epsilon^{1/3}$, you really mean 3 numbers, equally spaced on a circle)

The relative error is $\epsilon^{1/3}$; so if we want the perturbation to be small in single precision, we need to carry triple precision. If $\epsilon^{1/3} \sim 10^{-14}$, $\epsilon \sim 10^{-42}$. We almost have a theorem that says to solve a cubic whose coefficients are known precisely in single precision + compute the roots to roughly single precision, it is necessary to carry roughly triple precision. So it seems.

IF THERE ARE TRIPLE ROOTS

But there is, of course, a catch in this. It is the observation that things go wrong when you have a triple root, or a nearly triple root. If there were only a double root, + the other far away, double precision would suffice (by a similar argument). And if the roots are all well-separated (whatever that means), then a little more than single precision will suffice.

You can easily discover when there is a (nearly) triple root; the coefficients are simply related.

$$a_3/a_2 \approx a_2/a_1 \approx a_1/a_0 \approx -\xi$$

in this case. The natural thing to do is to move the origin into the midst of that triple; then the roots won't look so close together. If the root is exactly triple, you'll get the cubic $x^3 = 0$. Or you'd get x^3 + very tiny coefficients $= 0$. Rounding errors in those coeff. will not be less important.

But can you do the transformation? You want to form:

$P(z) = Q(z+\mu)$ where μ is the estimate of the triple root ξ.

this would make the numbers well separated in floating point terms, where powers of the base can be ignored.

Compute

$$P(z) = a_0 z^3 + \cdots + Q(\mu)$$ constant term

But how do you get $Q(\mu)$? By computing

$$Q(\mu) = ((a_0\mu + 3a_1)\mu + \cdots) + a_3$$

MOVING THE ORIGIN

You're doomed! When you compute $Q(\mu)$ you put in those rounding errors you had hoped to avoid. You thought you were moving the origin, but doing so brought in those rounding errors. It is that fact that tempted one of my friends to say "We've got a theorem."

There is another way to do this; it looks like it should round off in the same way but when executed it doesn't.* (*Some of you will say that if what Kahan does is execute the recurrence $Q(\mu)$ in double precision which he implements by trickery doing some single precision dithering, then you'll feel cheated. I'm not going to cheat you.) end

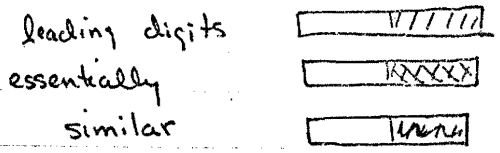
footnote

Here's how the algorithm goes.

$$\mu \approx -a_3/a_2 \approx -a_2/a_1 \approx -a_1/a_0$$

These three quotients will all be similar. I want μ to be all the digits that agree in these three numbers.

You do that in the following way:



Then add & subtract a number large enough to clear out all those trailing digits. The biggest difference tells me how many digits must be removed, which tells me how far the numbers need to be right-shifted to clear out the garbage, which tells me how big the number has to be.

Choose one of these doctored numbers to be μ & it'll have the following property:

$\mu = |1xxx \cdots 0 \cdots 0|$ * (803)

(*footnote. This has all been written in machine independent code, of course.)

~~Here's the algorithm~~: THE ALGORITHM

$$P(z) = b_0 z^3 + 3b_1 z^2 + 3b_2 z + b_3$$

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

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

$b_3 = b_2\mu + b_3''$

~~This is the entire inside loop.~~

My claim is (1) that those are the correct coefficients; (2) if the a's are representable to single precision, you need only a little more than double precision (at most) to get the b's, precisely. You won't commit a single rounding error.

(3) The code is machine independent.

If this cubic should still be difficult to handle because of repeated roots simply repeat the process.**

footnote (** for a proof, see Prof. Kahan)

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~~ The tricks I've been telling you about are important ~~because~~ we would like ~~to~~ to know how to design machines ~~which~~ which are economical, easy to understand, have a rich set of nice properties that would allow you to write reasonably efficient programs. Then your programs would run on

8-13

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. Yet.* (* footnote 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. end)

footnote

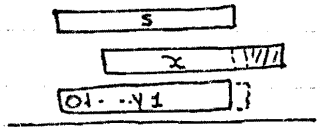
8Q1 Question: It is not immediately obvious that this works for IBM 360. Would you go through the argument?

Answer: If s & x have the same characteristic there is no problem - they line up & Y1 is exact. So consider x having to be right shifted. Assume a shift of 1 only. Then x slips into the guard digit & no error is made; subtract; then (a) the leading digit will clear or (b) it won't.

If it didn't clear (b), then I have the old picture of Y1 fitting into a frame under s.

If it does clear (a), then Y1 is exact, because of the guard digit. When Y1 is subtracted from s, the guard digit is still there. Nothing is lost; the difference is a machine representable number, namely x.

Say now x had to be shifted far to the right.



Things are interesting only if the leading digit clears (if it doesn't, Y1 & s have same characteristic), Y1 has the extra guard digit as part of it. But the difference between s & y, is what is called x after the digits past the guard digit have been thrown away. When you later subtract y, from s, you get back x minus the chopped digits, & it is machine representable.

(8Q2) Question: But you have to verify this individually for all machines?

Answer: Yes, & now you understand why I cannot categorically & comprehensively tell you what are the set of hypotheses that must be satisfied that this code will work.

Q8-2

8Q3 Question: for solving the cubic, in getting μ you have to know the word length don't you?

Answer: Yes. Then the issue is: do we need single, double, or triple precision, so I assume I'm entitled to know how accurate single precision is. But I really don't need the word size; I don't care how many digits are shifted off, just so they are. Dividing the biggest difference into one of the quotients + getting another quotient tells you how big the adding number should be. Nowhere does the word length appear.

$f(0) = 1 - e(0) = 0$

$e(1) = 1/2 \; mlp$

$f(n+1) =$

$f: \sqrt{x} \equiv \sqrt{x}\left(1 + \frac{1}{2}e^{-\epsilon}\right)$

$e(0) = 0$

$\therefore e(F(1)) = F(1)\left(1 + \frac{1}{2}e^{-\epsilon}\right)$

$e(F(n+1)) = \left[\left(F(n) + \epsilon_n\right)^2 + 1\right.$

Ninth Lecture - Nov. 3, 1970

Certain problems will be assigned to the students, as major projects

a) code for precision - doubling + pyramiding it. The purpose here is not to provide efficient multiple precision coding, but rather to see if it is possible for a reasonable range of machines to show that no theorem of the sort that such + such precision is needed to solve this problem is provable, because you can write almost machine-independent code that will pyramid the precision to any extent that you like. It would be enough to start with addition, since then multiplication + division are simply technicalities.

b) write a subroutine to solve a cubic, to approximately single precision without worrying too much about over/underflow. You are to deal with rounding errors. The essence is to solve the cubic to single precision using double precision (provided it is decent). This is to be done in FORTRAN.

c) the third project is to look at certain CDC subroutines - SQRT, CSQRT, CABS - + do to them what was done to the SQRT on the 7094. The object will be to find out how accurate the routines are, to make them more accurate if you can without making them much slower, + check them for other desirable properties.

CAN USING DIF1 MAKE THE ANSWER WORSE

Is it possible, by using the function DIF1 from last time, to get an answer that is worse than not having used it at all? This needs to be checked for machines like 7094 + B5500, which have guard words. Must check that if the difference is representable exactly, that's what you get. This code will obviously

not work on machines which represent numbers by their logs.

The essence of working out DIF1 for machines on which it is not needed is that in using this code to make it machine independent the results may be worse than before. * (9Q13,-5)

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

9Q6-8,10

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

SOLVING A CUBIC

Now we return to the problem of solving the cubic.

$$Q(x) = a_0 x^3 + a_1 x^2 + a_2 x + a_3$$

I went through an argument that showed that to solve this problem to single precision, you somehow needed to carry triple precision. If the roots are nearly coincident, a small change in the equation will perturb the roots by roughly the cube

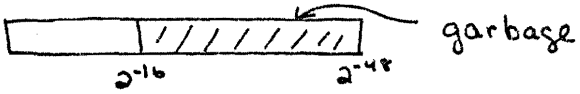
root of the perturbation.

$$x = \xi(1 + \epsilon^{1/3})$$

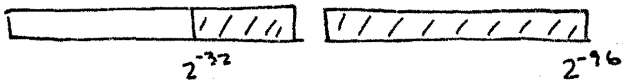
~~So to get $\epsilon^{1/3}$ to ϵ^{11}, you must have ϵ to ϵ^{33}, or~~

~~roughly, triple precision.~~

If the error in the coefficients is in the last place in single precision, $\epsilon \sim 2^{-48}$, then the error in the roots is $\epsilon^{1/3} \sim 2^{-16}$.



If you were to use double precision, then $\epsilon \sim 2^{-96}$ & $\epsilon^{1/3} \sim 2^{-32}$



Yet there is an algorithm which looks innocent enough, but when used in double precision gives results as good as another using triple precision. It is important that the algorithm look innocent, because I can do this same sort of thing by another sort of trickery.

USUAL APPROACH TO SOLVING A CUBIC

To illustrate this, consider what people would normally do, when faced with this problem.

Write the cubic slightly differently:

$$Q(x) = a_0 x^3 + a_1 x^2 + a_2 x + a_3$$

If it appears that the roots are close, translate the origin to increase their relative separation. Then the roots are much less sensitive to perturbations.*

footnote (*This hides a host of other argumentation, that says things go wrong only when roots are repeated. That's true for cubics but not for higher degree polynomials. If you take the polynomial whose roots are the integers from 1 to 20, you find that the coeff. don't fit into the machine. But you solve the resulting equation exactly anyway, and the roots are nowhere near the integers from 1 to 20.)

For polynomials of degree 2 or 3, things only go wrong when the roots are repeated. If there is a double root well-separated from a simple root, double precision does the trick. Triple roots require triple precision. So you want to separate the roots.

To translate the origin, the usual way is to construct:

$$Q(z+\mu) = \beta_0 z^3 + \beta_1 z^2 + \beta_2 z + \beta_3$$

CANCELLATION IS NEEDED

You compute the β's easily, using Horner's (≈ Ruffini's) recurrance:

$$\beta_0 = \alpha_0 \qquad \beta_1' = \beta_0\mu + \alpha_1 \qquad \beta_2' = \beta_1'\mu + \alpha_2 \qquad \beta_3 = \beta_2'\mu + \alpha_3$$
$$\beta_1'' = \beta_0\mu + \beta_1' \qquad \beta_2 = \beta_1'' + \beta_2'$$
$$\beta_1 = \beta_0\mu + \beta_1''$$

Calculate these from left to right on each line by nesting do loops.

There is a serious problem here. If the roots are nearly repeated, nowhere in this scheme is there the cancellation that we shall see is essential to make the scheme work properly. (I know many of you have been taught that cancellation is bad. This can happen if you've earlier thrown away digits that you should have saved before doing a subsequent subtraction. If you were using an a 360 with the significance exception, you'd get trapped out if cancellation occured early. So you'd turn it off. I stress that it is not the cancellation itself that is bad. It is the lack of cancellation in Horner's recurrance that makes it bad.)

I can roughly compute what the α's are going to be.

$$Q(x) \approx \alpha_0 (x-\xi)^3$$
$$\alpha_1 \approx -3\alpha_0\xi$$
$$\alpha_2 \approx 3\alpha_0\xi^2$$
$$\alpha_3 \approx -\alpha_0\xi^3 \qquad \mu \approx \xi$$

Then $\beta_1' \approx \alpha_0\mu - 3\alpha_0\mu = -2\alpha_0\mu$

If you continue, none of the numbers actually cancell off.

HOW MANY DIGITS NEEDED

Because cancellation does not occur, the number of digits required to represent them exactly, in the absence of rounding error, tends to grow. It grows until you get to the bottom numbers in each column, where cancellations begins.

e.g., $\beta_3 \approx Q(\mu) \approx Q(\xi) \approx 0$

Two large numbers will have had to cancell. How many digits did they require? β_1' requires d.p. (product of 2 s.p. numbers); β_2' then would take triple p.; which would lead to quadruple precision for β_3 to do the job exactly*. But that can't be right; α_3 is s.p. + the two numbers are suppose to cancell; but quad. precision is needed to be exact.

You can get around this by the following trickery: have μ approximate ξ in only one digit. Then β_1' is s.p. + 1 digit; β_2' is s.p. + 2 digits; β_3 is s.p. + 3 digits but then cancellation shortens it.*

footnote

(*That is exactly what Horner + Ruffini had in mind. Their reason for wanting μ to be one digit was that then you could do the multiplication in your head. So they said here was a way to solve any polynomial without having to do lots of long arithmetic. Nevertheless, as the process continues, the numbers get longer + longer, so other rules for tossing away digits were made. These are somewhat described in Uspensky's Theory of Equations or Turnbull's theory of Equations. To see the rules in detail, you'd have to go to an old edition, like 1812, of Barlow's Tables.)

But on a computer nowadays, we just wouldn't do things that way. It wouldn't be worth the time to write a loop that picks off one digit at a time from your approximation. There are too many decisions like how do you know you have 1 or 2 digits of the approximation (in machine independent code), what to throw away at the right, etc.

ONLY DOUBLE PRECISION IS NEEDED

But a little bit of thinking that way would help. Altho this scheme doesn't get around appearing to require either triple precision arithmetic or this sort of horrible mess, there is another arrangement which does. I started it last time.

$$Q(x) = b_0 x^3 + 3b_1 x^2 + 3b_2 x + b_3$$

The recurrence I use seems to be doing the same thing, but in a slightly different order.

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

This scheme requires the same amount of work as before, just in a different order. It is even hard, unless you notice the a's in the first line, to see that this isn't Horner's recurrence. Well, it is not. Cancellation now will take place at a gracious rate.

e.g., $b_1 \approx a_0\mu + a_1$ but $a_1 \approx -a_0\xi$

$\approx a_0\mu - a_0\xi \approx 0$ if $\mu \approx \xi$.

for all the first row, lots of cancellation occurs. All you need to insure is that each multiply + add is done precisely to get numbers for the next stage of the algorithm that are quite precise, even tho cancellation has occurred. So at the very worst, you need double precision and a guard digit; or if you insure that μ is not a full single precision number, d.p. will be enough.

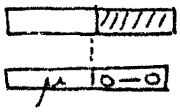
FINDING μ

In the worse cases, cancellation will occur in the second line as well, and right down to the end.

The problem is to find a value of μ for which this cancellation will take place. There was a question last time if finding this μ was machine independent, + I gave a misleading answer. It is machine independent in a sense; you do have to state either the base of

the machine or the precision of the machine in single precision. Given either of those pieces of information you can compute a μ that is approximately equal to the multiple root, but has some trailing zeroes. You need those zeroes as the b's will grow in length, slowly.

$$a_i/a_{i-1} \approx \mu \xi$$



FINDING μ IS HARD

μ matches ξ in all the digits you write down for μ. This has to be true for all three of the roots. So suppose it is true. Then I can prove that the algorithm will work.

But there is a hooker, + that is that it is very hard to do this. Even in assemply language. And the reason is this. If you have three roots close together,

$$\xi_i = \xi(1+\lambda_i)$$

then for the quotients you have:

$$-a_i/a_{i-1} \approx \xi(1 + \tfrac{1}{3}\epsilon\lambda_i + O(\lambda^2))$$

The quotients agree with each other more closely than the roots agree, even to triple precision when the roots agree to only single precision. When you see the three quotients matching beautifully, you mustn't think the roots will also match beautifully. They don't. You choose some number that matches those three quotients closely. Then μ has superfluous digits in it & my proof doesn't work, but the algorithm continues to work for reasons that are not entirely clear to me. The cancellation is so ferocious in the first row that it makes up for the cancellation that doesn't occur in the second or third.

Therefore, it is conceivable that we could solve a cubic using only double precision or a bit less, even tho it looks like triple precision or a bit more is needed.

(9Q11)

WHEN WILL THIS ALGORITHM WORK

So this is an example of an algorithm for which we would drawn the following conclusion: we could write an essentially machine independent algorithm that provides results that are more accurate than you could hope to prove if you used the kind of error analysis in Wilkinson's book, where he says whenever you add two numbers:

$$C = A+B$$

the sum $c = a(1+\alpha) + b(1+\beta)$ is no worse than you'd have gotten if you bumped each of the numbers A & B by no more than a unit in the last place.

That will turn out to be wrong, because, as you see, this trick will not work on machines that use a logarithmic representation for numbers, even tho on such a machine the above relationship is true. For the cubic code to work, it is essential that there be cancellation, and that it can occur without error. On a log machine that is just not true.

If we use this analysis we'll get an error bound that is much worse than the observed error in almost any case & maybe in all cases. If the error bound to very much too big, it won't be used; so why have it? Error bounds are only worthwhile if they are a modest multiple, say 100 or 1000, of the worse possible error, not when they are of the order of 10^{14}, as on the COE for this example.

A SUCCESSFUL ERROR ANALYSIS

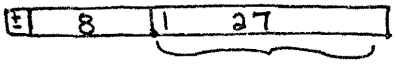
Now I'll give you a successful error analysis. It is based on a very intimate appreciation of the hardware of the machine. You have to choose a simple algorithm - I have chosen square root. this analysis must be done for elementary functions.* (* all the elementary functions for IBM 360/50, in single and double precision, have been analyzed in this way to the extent that number theory

footnote

wasn't needed. The man who wrote them has done this & is able to say ~~something~~ 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.)

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

this is for a 7090 or 7094.



sign magnitude machine

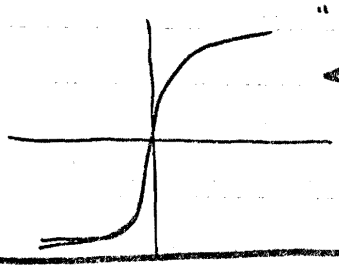
important details are here, & the last bit of the characteristic.

Specification for the SQRT routine:

i) $SQRT(x) \doteq \sqrt{x}$ $\quad x \geq 0$

ii) $SQRT(x) \doteq -\sqrt{-x}$ $\quad x < -0$ *

and an error trace & message "SQRT(-x) = -SQRT(x)"



← graph of SQRT(x), a nice continuous graph

(* the response to taking a square root of a negative number is not at all obvious. It's not obvious that he should be kicked off the machine.)

Specifications on the ERROR

i) error cannot exceed .5000163 ulp's (recall 27 bits is roughly 8 decimal digits, so the error 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' 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 & then rounding it correctly to 27 bits.

The error bound is obtained by exhibiting those 29; for those correctly rounded, the error is ½ 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 & in those regions you enumerate all the arguments.
(9Q12-14)

SPECIFICATIONS TO BE MATCHED

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

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

ii) SQRT(X**2) = SQRT(RND(X*X)) = ABS(X)
(exactly 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.)
(9Q15)

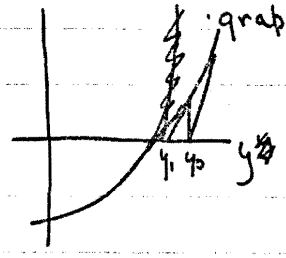
There are questions of how long the program should

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 squareroot of a number that has temporarily overflowed into the P & Q reg 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' rule, now know as Newton's method for solving a quadratic.

$y^2 = X$



$$y_1 = \tfrac{1}{2}(y_0 + X/y_0)$$

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 is to pretend:

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

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

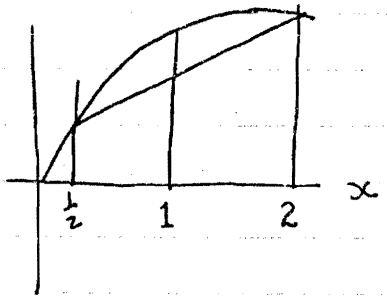
$$\text{so } \delta_1 = \delta_0^2 \quad \text{(quadratic convergence)}.$$

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 squareroot.

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.

(9Q16)

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:

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

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

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

$$y_0 = 2^{I} \left(F/2 - J/4 + C \right)$$

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

9Q17

Questions lecture 9

9Q1 Q: How did you happen to write this code?
A: It was written when we were switching from a 650 to to a 7090 using code that had been previously written for a Frandtti-Manchester Mark I, when we were wondering how we'd ever live through all these transitions. So an attempt was made to write code that was machine independent

9Q2. Q: I can't see in my mind the sequence that led to your having written such code.
A: In my mind was a picture of digits, of course, digit strings as might come from a desk calculator, octal calculator, or a binary machine. The Frandtti-Manchester had to be coded in teletype code -- there was no assembly language so everything had to be visualized quite clearly - digits had to be input as characters like /, @, 4.
It was the practice of visualizing what was going on in the F-M that made it possible for me to see what was going to happen for the various implementations. The tricky part is the 0.1 * X - who would think to do that?

9Q3 Q: That was the part I was looking at. I more or less convinced myself that it would work on the 6400 + probably on the 360, for entirely different reasons. I don't call that machine independent.
A: The trouble is that it is machine independent in the sense that it is independent of the machines currently on the market.

9Q4 Q: By a proof that is different for each one.
A: Right + that is very unsatisfactory.

905 Q: I don't see any underlying principals.
A: The underlying principles is are that the digits get pushed off the right hand side of the register but it is not exactly certain what they do as they go. ~~[illegible]~~

906 Q: Isn't it true on that precise argument, that code like this, that looks like FORTRAN, is actually going to be more expensive to transfer to a new machine than code that is explicitly machine code where somebody knows he'll have to go in + rewrite the code
A: ~~[illegible]~~ What you are saying is, shall we write the code in assembly language with careful documentation to explain exactly what we are doing hoping then, that when we switch to another machine, anyone who reads and understands the documentation will be able to translate ~~[illegible]~~ into the new assembly language, or should we try to write in machine independent code with some sort of theorem, even an ugly one, that tells us it'll work on almost any machine you can think of. It seems to you that the first rationale is more sensible. I used to think so to. But I have some code I wrote in 1965 that I can no longer understand, even tho it is richly commented It was written in assembly language + uses every bit of the machine to squeeze every ounce of performance out of its code. Now, even tho it was perfectly reasonable + transparent at the time, it would take me several days to once again understand it. That's very expensive.

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

p8
9Q3

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

9Q8 Q: Why are manufactures so unresponsive to your pleas?
A: Manufacturess are individuals who are from time to time on the ascendancy politically, or not. The sales organization is generally on the ascendency, when the company is on the make. The salesmen have their own particular way of finding out what their customers want. But customers only know part of what they want. So salesmen make rather shallow estimates of what is wanted + present misbegotten specifications to the engineers, who are happy to implement anything. CDC salesmen collected the specs for the 6000 + 7600. One salesman tried to tell me his customers didn't want the machine to round, because he hadn't heard the appendage "the way they were doing it." And you know why. So the salesmen said, they don't have to use the round instruction.

9Q9 Q: In going through the code for DIF1, I couldn't find examples when D & D1 were different.
A: It happens when X & X1 have different exponents, X1 with the larger. The last digit of D gets tossed off (that is the last digit of D when it fits into the same frame as X). This can't happen on a truncating machine (because X1 ≥ X in this case). On a rounding machine, can have X ≥ X1.

trunc.	round.
$y_1 = s - x - e$	$y_1 = s - x + e$
$x_1 = x + e$	$x_1 = x - e$

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

9Q10 Q: Not only does it vary from compiler to compiler, but one of my friends on the system staff tells me it is incorrect. The guy who wrote the algorithm for the compiler changed certain things in the way it did its double precision multiplies, for instance, so that it would go faster.

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

9Q11 Q: Are you saying that the group that works on this project would have succeeded if they turned in a counterexample. to your arguments?

A: If they actually generated a cubic for which a failure occured in this part of the algorithm, that would be of enormous value to me. It would be a disappointment to the users of the subroutine. But I have reasons to believe that that won't happen. I don't really believe the proof for this algorithm, but I believe it enough to write a program to see what happens. The proof comes from the fact that you can force enough cancellation in this algorithm so that after the cancellation, if you didn't make any rounding errors before the cancellation, the rounding errors tend to be rather irrelevant. You don't have to do the rest of the algorithm exactly to get all its benefits.

9Q12 Q: When you are checking these routines for the largest errors, suppose you are checking your double precision version?

A: 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.

9Q13 Q: Would you go thru the argument of what the 2^{34} numbers there are, + in particular, do you except unnormalized numbers?

A: 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' essentially different operands, or 2^{27}. Only 29 of those give incorrect rounding.

9Q14 Q: Is it part of the specifications that the routine only produces correct results for normalized operands?

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

9Q15 Q: 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 that is left is in the double precision part of the register.

A: 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, + on it if two numbers are different their difference is nonzero, unless it underflows, + then you get a message.

9Q16 Q: Wouldn't it be the range 0 to 2, not ½ to 2?

A: No. Remember, zero is not a normal, floating point number. SQRT(0) = 0; it is too easy. And the squareroot 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 ½ to 2, without any rounding error.

9Q17 Q: Did you try table lookup?

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

Tenth Lecture – Nov. 5, 1970

Question on the material from last time – See 10Q1-3

The digression (in the questions) may have frightened you into thinking that to write a squareroot routine you have to have spent years studying obstruse theories. I guess if you want to write the best possible squareroot 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 squareroot. 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: $\varphi \le x \le \varphi^{-1}$ (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 sence. The most natural is

relative error for floating point numbers.

$$\min_{a,b,c,d} \max_{\varphi \le x \le \varphi'} \left| \ln\left(\frac{ax+b}{cx+d}\right) - \ln \sqrt{x} \right|$$

The relative error is perhaps most easily written in the log form.*

footnote (* I know that certain kinds of people think of relative error in a different way. If you consider the ratio of the values by which one function approximates another, f and F, the relative error is presumably like: Alternate ways to measure relative error are:

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

What I have is

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

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.)

HOW MANY COEFFICIENTS ARE THERE?

In discussing a specific problem, I get rather different answers sometimes, depending on which measure of error I've used. If instead of the above, I used for the absolute error,

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

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

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 (can divide all coefficients by a constant, or make one of them 1).

Actually, there are only two coefficients that matter.

9Q4 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 squareroot may not be obvious. So consider the sine function. It is an odd function, & 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 squareroot. Well, I've somewhat begged the issue by providing the interval $[\varphi, \varphi^{-1}]$.

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 squareroot is such a case. (* footnote) (A book on this subject is by J. Rice, in two volumes; Cheney in one volume (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 & is unique.)

THE BEST APPROXIMATION

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

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

The function is replaced by one of the same kind;

The function exhibits the same symmetry properties as the square root.

Then you can use the same criterion for relative error:

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

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:

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

There are only two independent parameters. *

---

footnote (* 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 squareroot of the reciprocal is the reciprocal of the squareroot, 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.)

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, & then people are reduced to what amounts to a certain amount of intelligent trial & 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.* 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.

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

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

$$y_0 = 2^I \left(C + F/2 - J/4 \right)$$ starting approximation for the squareroot.

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

```
CLA     op'd > 0      (leave out test for sign + 0 in this discussion)
STO     X  (op'd)     store X
ORA     776 77...7    ← fraction part : this picks J off
ARS     1             right shift 1 the exponent
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, * adjusted the constant. *

(*The ORA was used because it took 2 cycles & allowed overlap, the ANA took 3 cycles & suppressed overlap). (10Q5)

WHAT HAPPENS IN THE CODE

	J = 0	J = 1
CLA (clear & add)		
STO (store)		
ANA (and 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	$(I) + (\frac{1}{2}F + C)$ recall J=0	$(I) + (\frac{1}{2}F - \frac{1}{4} + C)$ recall J=1

(last two instructions have been swapped, doesn't matter except maybe for overflow) (10Q6)

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 $y_0 = 2^I(C + F/2 + J/4)$ ***

*** footnote After this was done, a man by the name of Herando 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.) (10Q7-8)

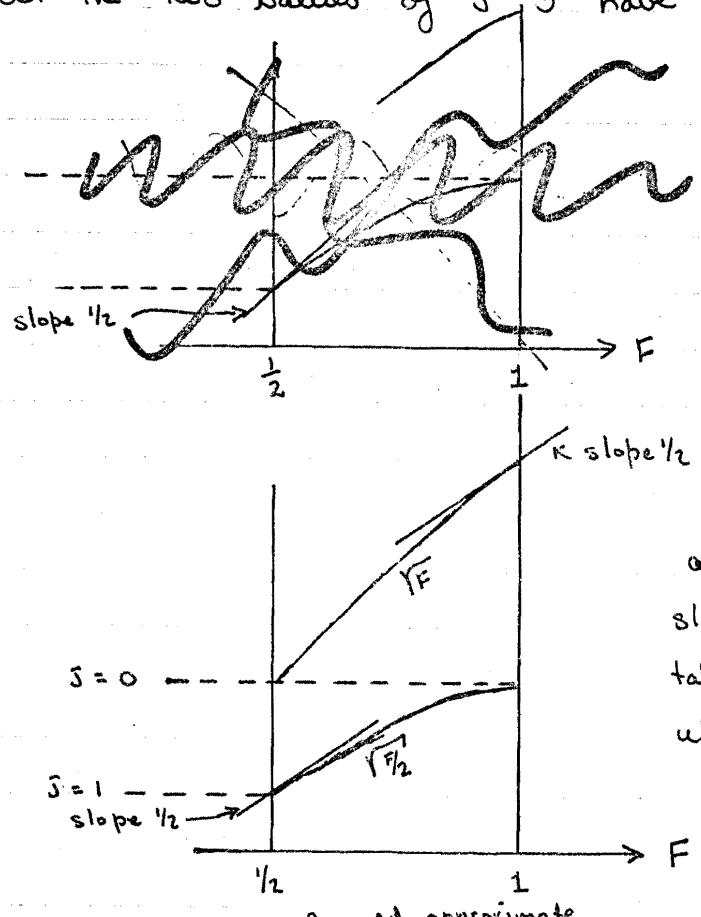
CHOOSING C

In order to explain what to do with the approximation, I will again have to introduce an artifact. 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 1/2. C_J has the task of shifting the line up and down in parallel.

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

All I have to do is to 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:

$J = 0$

$y_0 = C_0 + F/2 \approx \sqrt{F}$

$J = 1$

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

Do a transformation for $J = 1$

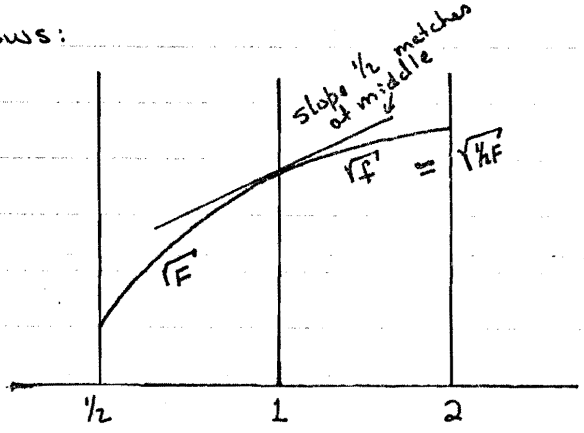
$F \equiv \frac{1}{2} f$

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

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

$1 \le f < 2$

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:

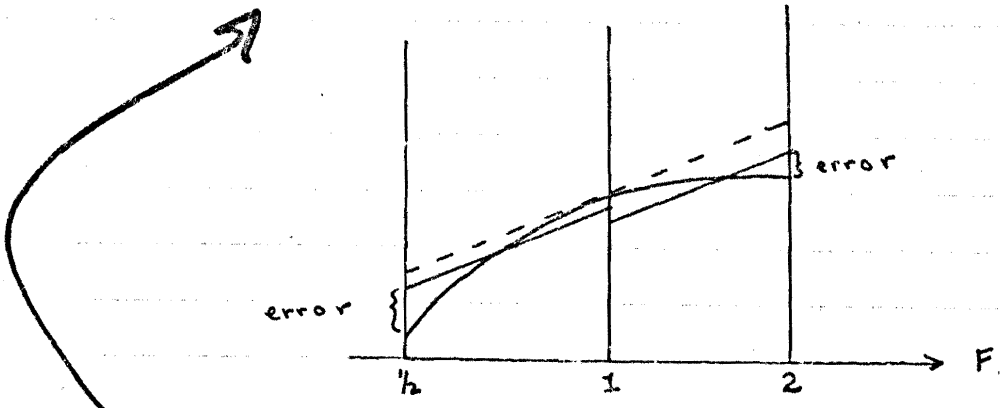


It is just a matter of scaling, so the relative error is still the same.

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

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

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

If 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.* 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 & C_1 are the same, you'll see that C_1 is doubled in the constant $(2C_1 - 1/2)$. So decreasing C_0 reduces the constant in brackets twice as much & there is a break in the line. 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.

*footnote

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

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

$$y_1 = \tfrac{1}{2}\left(y_0 + \tfrac{x}{y_0}\right) = \sqrt{x}\left(\frac{1+\delta_0^2}{1-\delta_0^2}\right) = \sqrt{x}\left(\frac{1+\delta_1}{1-\delta_1}\right)$$

$$\delta_1 = \delta_0^2$$

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 BEST VALUE TO USE FOR C

The right value for C worked out to be:

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

$$= .367056630_8$$

So C is a little less than a half. Needless to say, altho 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 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 squareroot, (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.

$\delta_0 = .0348\cdots$

$\delta_1 = .00063\cdots$ after one application of Heron's rule

$\delta_2 = .0000002$

$\delta_3 = .0^{13}21$

?

After 3 applications of Heron's rule, we have a very accurate result, in the absense of rounding error. We only need 27 bits, which is 10^{-8} or 10^{-9}. The error is down to 10^{-13}. (10Q9)

WHAT IF YOU USE C LESS ACCURATE

This tells us that the program is now feasible.

If you use the less accurate value for C (say 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 zeros in δ_3. If I had only 10 zeros there, the time needed to find out how good the program was would have been multiplied by 10^3. (10Q12$ -17)

HOW ACCURATE ARE THE RESULTS

To find out how accurate the final result is, we have to examine the coding for Heron's 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 characteristics lined up, which they usually did)
ADD   -1 B8 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

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

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

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

[redacted], 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 much could we do & how much would it cost.

10Q18

Questions from the 10th lecture

10Q1 Q. I have a question about the function you chose to start the SQRT. You said there were other choices. What were they? I didn't understand the tree.

A. The idea is to consider all possible programs, no longer than one program that already worked, that could compute a squareroot. 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 obviously things not to do (like increment an index you don't know).

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

A: Don't it?

Q: Most times you have some sort of objective in mind.

A: 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 & 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 & such an approximation.

I worked out this tree & 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, & 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 it is 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, [illegible] is very nice. Of all higher 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:

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

This converges to $x_\infty = \varphi(x_\infty)$; we then talk about the speed with which this iteration converges.
Convergence can be arbitrarily slow. But if φ is differentiable and if $|\varphi'(x_\infty)| < 1$, 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 $\varphi'(x_\infty) = 0$ and $\varphi''(x_\infty) \neq 0$, then the convergence is quadratic; i.e., the number of correct digits nearly doubles with each iteration.

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

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.

$$x = \varphi(x)$$ (there are infinitely many of these)

Say you want to solve

$$f(x) = 0.$$

You could just as easily say you want to solve:

$$\psi(x) \cdot f(x) = 0$$ (for any ψ)

Then consider:

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

Saying $x = \varphi(x)$ 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:

$$\varphi(x) = x - F(x)/F'(x)$$ (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 computeable - 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

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

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

Some of this rather elaborate theory is discussed in a book by J. F. Traub (~1960, Prentice Hall), discussing 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.

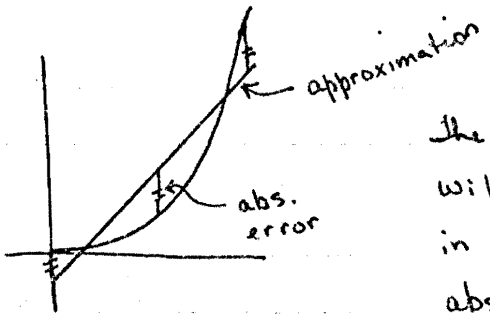
1002 Q: Why did you limit yourself to the number of instructions in another program?

A: Once I have a program that computes the squareroot, 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 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.

1003 Q. I don't see that you can get an upper bound. How many iterations of Heron's rule to you intend on normally using?
A: 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.

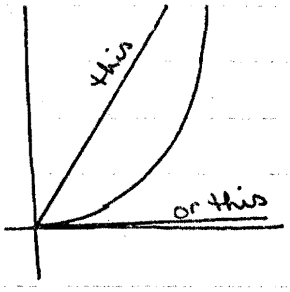
You get indications of how complicated a function is from the entropy theory of approximation; the theory is an attempt to decide how complicated a function is to compute (it is at best rudimentary). There are discussions which say analytic functions are infinitely less complicated to compute than, say,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

1004 Q: 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?
A: There is a difference between the way the ln form and the other one 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.



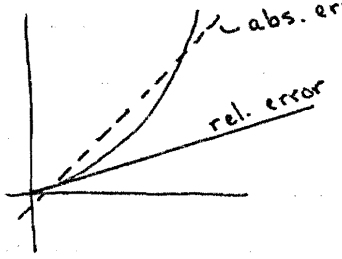
the best linear approximation will have the maximum error in each section equal, using absolute error.

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 & 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 squareroot function.

1095 Q: 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?

A: 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 recurrs. 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.

10Q6 Q: There's got to be more to it than your just having figured it out on a big sheet of paper with a tree. You had to work out the functions, & that meant a lot of interpretation of what all those bits meant and I think it is funny.

A: Okay.

10Q7: Q: How long did it take you to run the tree?

A: 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 left over 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.

10Q8 Q: This rather reminds 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.

A: It didn't actually get that bad. Altho 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. Altho I am peculiar I am not insane.

10Q9 Q. 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.

A: That is true. We could still get the trunction error small even with $C = 1/2$ that's what [illegible] did, & 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.

10Q10 Q. But you don't have that many digits around.

A. Actually, when you try to minimize the maximum error, it doesn't look like: 

but rather it is like  .

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.

10Q11 Q. You just said that if $C = 1/2$, you do better than a few ulps. You still have more accuracy than the machine can hold.

A. That's true, but on the other hand for the best C, $\delta_3 \sim 10^{-13}$. If I change C to $1/2$, $\delta_3 \sim 10^{-10}$, 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.

10Q12 Q. That's a different argument than you've been using for why you wanted the best value of C

A. I only want that much precision in C because I want to know what is 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 squareroot routine, & that's what Kuki did.

10Q13 Q. And it would be just as accurate as yours?

A. 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.

10Q14 Q. Is that a large difference in error?

A. 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. Every time improbe someone thought of a way to improbe a program epsilonically, he said "comeon 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.

10Q15 Q. Hadn't that been reached with Kuki's .5001?

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

10Q16 Q: What if you had used the exact same program, just with a constant closer to 1/2?
A: It wouldn't have mattered a great deal.

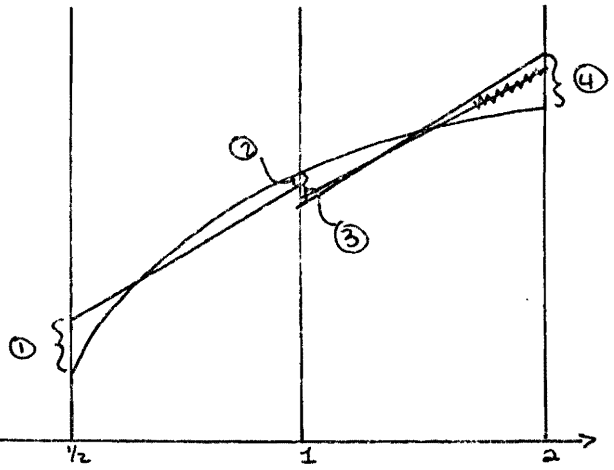
10Q17 Q: Wouldn't you have ended up with his error & your speed?
A: Yes. But I was determined to get the best possible program. You're trying to ask me, was it worth the money spent. Of course it wasn't worth the money spent if you want to figure it in terms of the number of happier users. I probably tested more numbers than'll be run thru the SQRT in a year on the 7094.

But we are trying to see how well we can do. & For the practical question, most people would have stopped where Kuki did. I hope so. We can't afford too many guys like me. But we can't afford to do without them either.

10Q18 Q: Why did you write out Heron's rule three times?
A: The index register instructions take 3 cycles for testing, 3 for setting, & 2 for restoration. Why bother when the loop is so short? Kuki used a loop; that's why his takes longer.

ELEVENTH LECTURE, NOV. 10, 1970

In getting the first approximation to the square root, we make linear approximations on each of two intervals, ~~from~~ $[1/2, 1]$ 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 reduced error ③ ④ ... the same with different signs, they ... equal to the relative error ①. The relative error ② is better.~~

The values of δ_0 that you would compute at points ①, ③, and ④ would all be the same, altho 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 δ 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.

$\delta_3 < 9.685 \times 10^{-15} < 10^{-14}$ (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 & see why rounding errors, up to the third iteration, do not make things appreciably worse. Here is the code again.

	STO	S	$\approx \sqrt{A}$
	CLA	A	} A/S in accumulator
	FDH	S	
exchange	XCA		
	FAD	S	
	SUB	=0.001000000000	divide by 2 by subtr. from exp.
	STO	S	$= 1/2\,(S - A/S)$
	DO this again		

→ At the end of one iteration, the error, δ_1, would be ------, 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.

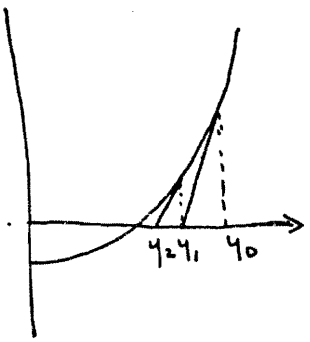
We are only interested to know if the error is appreciable and not just a few units in the last place of the squareroot. So say that the error is close to the computed bound, that is, about twice δ_1. Then it looks like a rounding error or two

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.

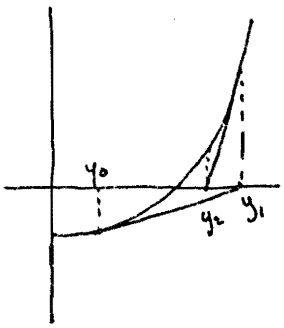
The only affect rounding errors will have on this monotonically decreasing sequence is to maybe speed things up a little bit, if you're far away. Why? You compute A/s + truncate so you throw a little bit away. The division by 2 doesn't affect anything. Then you do a floating add + throw away a bit more. 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.

---

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



or



You can also see that they decrease by showing that

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

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

ROUNDING THE THIRD TIME ONLY IS SUFFICIENT

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

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

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:

$$(S + A/S)/2$$

A/S has been truncated, ~~a truncating error of no consequence~~.

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).

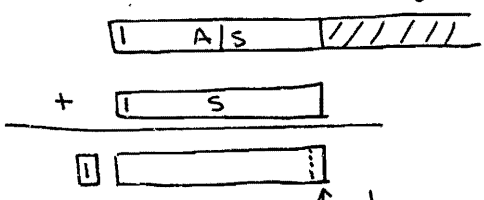
The truncating error in A/S is of no consequence. Why?

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}).

---

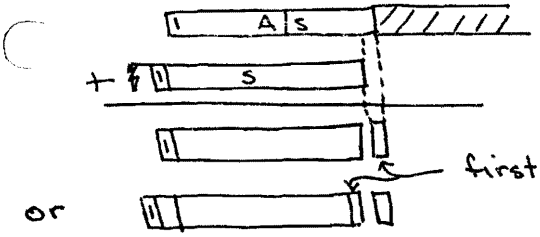
* If the machine had a rounded add instruction, you would use that instead of FAD, FRN.

Case 1: exponents equal



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 + I do have something there. The truncation didn't matter because I merely threw away digits I would not have looked at ~~anyway~~ 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 ~~not~~ been used.

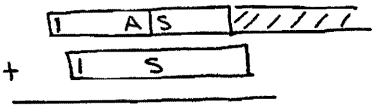
Question: What if your machine obeyed the rule ~~rounded~~ to nearest even'?

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 ~~was~~ is the first situation I've seen in which rounding to nearest even might be less than advantageous.

CANNOT BE MUCH TOO SMALL

You should remember that by this time our approximation S is extremely good. It's relative error is down to around 10^{-7}. So my assumptions are valid, unless our approximation was so good

at a bunch of things happened that couldn't happen, [redacted], so that A/s is too big ($s < \sqrt{A}$). Then the situation is:



[redacted]

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 squareroot 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. [redacted]

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 + 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 + then some bigger garbage. Then you 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 + 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:

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

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

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

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

If n is odd, the MQ looks like *(1)

$$\boxed{011\cdots\cdots1xxx}$$

garbage from $2^{-55} n^2$ term

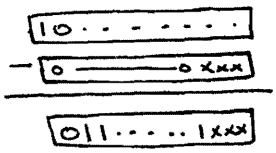
If n is even, the MQ looks like

$$\boxed{100\cdots\cdots0xxx}$$

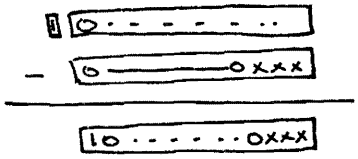
Therefore, to round $\sqrt{A}$ correctly, we should use:

$$\sqrt{A} = 1 + 2^{-26} \left[\frac{n}{2}\right] \qquad \left[\text{largest integer in } n/2\right] \quad *(2)$$

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



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



(2) footnote: if n is even, $\left[\frac{n}{2}\right] = \frac{n}{2}$, 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:

$$S_2 = 1 + 2^{-26}\left\{\left[\frac{n}{2}\right] + k\right\} \qquad \text{where } k \text{ is modest integer} \geq -1$$

(S_2 could be small by a unit in the last place)

Now what happens in Heron's rule?

$$A/S_2 = (1 + 2^{-26}n)\left(1 - 2^{-26}\left(\left[\frac{n}{2}\right]+k\right) + 2^{-52}\left(\left[\frac{n}{2}\right]+k\right)^2 - \ldots\right)$$

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

$$A/S_2 = 1 + 2^{-26}\left(n - \left[\frac{n}{2}\right] - k\right) + 2^{-52}\left(\left[\frac{n}{2}\right]+k\right)\left(\left[\frac{n}{2}\right]+k-n\right) + \ldots$$

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.*(3)

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

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

Answer: No, I mean machine chopped.

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

Answer: The quotient is exact if you write out the whole series, but the machine only writes out the first 27 bits of it. Those 27 bits will have contributions from later terms. If the 2^{-52} term is positive, the bits simply get thrown away. But if that term is negative, 1 will be subtracted from the 27th bit + 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.*(4)

$$(A/s_2 + s_2)/2 = 1 + 2^{-26}[\tfrac{n}{2}] + 2^{-53}(\quad)(\quad)$$

---

*(3) If you don't believe that, run thru the cases when k is a small integer, + see what happens.

*(4)
$$A/s_2 + s_2 = 1 + 2^{-26}(n - [\tfrac{n}{2}] - k) + 2^{-52}(\quad)(\quad) + \cdots + 1 + 2^{-26}([\tfrac{n}{2}] + k)$$
$$= 2 + 2^{-26}(n - [\tfrac{n}{2}] + [\tfrac{n}{2}] - k + k) + 2^{-52}(\quad)(\quad) + \cdots$$
$$= 2 + 2^{-26}n + 2^{-52}(\quad)(\quad) + \cdots$$
$$(A/s_2 + s_2)/2 = 1 + 2^{-26}[\tfrac{n}{2}] + 2^{-53}(\quad)(\quad) + \cdots$$

There are two possibilities for the 2^{-53} terms. (1) It could be positive or zero. Then, if n is odd, adding half in the last place will bump up the sum.

SPECIFIC EXAMPLE OF INCORRECT ROUNDING

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

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

$A/S_2 = 1$

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

1 0 . . . . . 0 | 1 0 0

This then gets rounded up to $1 + 2^{-26}$ or 1 0 . . . . . 0 1, 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. *(6)

So you see that there are cases when the error in the squareroot 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.

---

*(6) If $A = 1 - 2^{-27} n$ (if a number is a little less than 1, the leading bit represents a half 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 & paper. You should verify for small odd n, that for small $k > 0$, you round up when you should round down.

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.

$\sqrt{23790000} = 4877.4994\ldots$

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 on.

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 systemically, mainly because it can be done 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 & stated that the error was no more than 1 ulp? 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 sqrt was monotonic & 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 sqrt. will not decrease. Or that the sqrt of a perfect square recovers the number.

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

Answer: My estimate will be sufficiently close so that getting those

lets
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 squareroot almost to double precision; it's a little too big by some numbers near the end of the second word. If I take the squareroot of a perfect square, the squareroot fits into the top word; the extra digits from Heron's rule go away when I round.

Questions: That ignores truncation errors from subtracting and dividing.

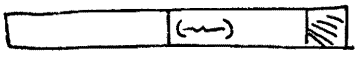
Answer: No, I showed that truncation during division is irrelevant.

So I have computed the correct squareroot 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 squareroot 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.

$\sqrt{x'} \approx$ 

$x' > x$

$\sqrt{x'} \approx$  may have smaller garbage but is bigger than $\sqrt{x}$ by $\sim$ ½ 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 squareroot 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.*

* footnote

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 COC 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.

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:

$$\left.\begin{array}{ll} X = 2^{26} M & \text{case 1} \\ X = 2^{27} M & \text{case 2} \end{array}\right\} \quad 2^{26} \le M < 2^{27}$$

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 squareroot, 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, if you have done your job properly.

$$N - 1/2 \le \sqrt{X} < N + 1/2$$

$$2^{26} \le N < 2^{26}\sqrt{2} \qquad \text{case 1}$$

$$2^{26}\sqrt{2} \le N < 2^{27} \qquad \text{case 2}$$

N will just barely fit into a single precision word. N is the correct value you would get for the squareroot 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; ~~but~~ 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: [redacted]

$$\text{If } \sqrt{X} \approx N \pm 1/2, \text{ then}$$

$$4X \approx (2N \pm 1)^2$$

But I have to know what I mean by approximately equal ($\approx$).

1-16

I will have trouble when $4X(1+e)$, which is, afterall, what I will have computed, is approximately equal to, or indistinguishable from, $(2N \pm 1)^2$. I will have problems when the set of numbers, $\{4X(1+e)\}$, $|e| < 2 \times 10^{-14}$, is included in $(2N \pm 1)^2$.

$$\{4X(1+e)\} \geq (2N \pm 1)^2 \qquad |e| < 2 \times 10^{-14}$$

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 thru the values -2×10^{-14} to 2×10^{-14}, thru 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, + at the bound on e, we see:

$$|C| \lesssim 4 \times 10^{-14} \cdot (2N \pm 1)^2$$ it can't be any bigger than this.

We have the following relations:

$$(2N \pm 1)^2 \equiv C \mod 2^{28} \quad \text{case 1} \ *$$
$$\equiv C \mod 2^{29} \quad \text{case 2}$$

---

* This means $(2N \pm 1)^2 = C$ times some integer multiple of 2^{28}

$|C| \leq 4 \times 10^{-14} \cdot 4X = \ldots\ldots < 2000$ in case 1 *

< 4000 in case 2

There really aren't very many values of C. There are about 4000 in case 1 + 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, + compute the squareroot + see if I get N. That's all. But it is not at all clear how you'd solve that equation. It's was clear to me for quite a while.

~~[illegible]~~ SOLVE BY RECURRENCE

A man by the name of Heilbrand, a renowned number theorist, said isn't there recurrence for things like that. ~~&~~ the one he gave me didn't work, but there was one.

We will pursue the recurrence by which you can solve equations of that kind. We want to solve:

$$(2N \pm 1)^2 \equiv C$$

Case 1 $2^{26} \leq N < 2^{26}\sqrt{2}$ & $(2N \pm 1)^2 \equiv C \bmod 2^{28}$

Case 2 $\sqrt{2}\,2^{26} \leq N < 2^{27}$ & $(2N \pm 1)^2 \equiv C \bmod 2^{29}$

Given C, find N. That's the problem.

Then we observe that

$C \equiv 1 \bmod 8$ † (next page footnote)

That cuts down on the interesting numbers; there are only 1/8 as many.

* $|C| \leq 4 \times 10^{-14} \cdot 4X$ because $4X$ very nearly equals $(2N \pm 1)^2$

[illegible]

We are down to about 1000 numbers in case 2 & 500 in case 1; that is so small as to be almost ~~negligible~~ 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

$$z^2 \equiv C \mod 2^m \qquad \text{when} \quad C \equiv 1 \mod 8$$

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.

$$(2^M - z)^2 \equiv z^2 \mod 2^m \qquad \text{(is really } (-z)^2\text{)}$$

[redacted]

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 $1/2 \cdot 2^M$, then compute $(2^M - z)$ & get an answer that is smaller than half $\cdot 2^M$.

So I might as well assume:

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

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

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

Either N is even, or $N \pm 1$ is even: $\therefore 4N(N \pm 1)$ is a multiple of 8

therefore $C \equiv 1 \mod 8$

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 & observe:

$$(2^{M-1} - z)^2 \equiv z^2 \mod 2^m$$

This happens because of the factor of 2 that appears in the square.*

Thus, z can be further reduced to:

$$0 < z < 2^{M-2}$$ z can be tranformed 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~~ four. Are there any more? No & let's see why.

Suppose $z^2 \equiv Y^2 \equiv C \mod 2^m$

$$0 < Y \leq z < 2^{m-2} \qquad C \equiv 1 \mod 8$$

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 & Y must be odd (their squares are odd).

$$0 \equiv z^2 - Y^2 \equiv (z-Y)(z+Y) \mod 2^m$$

I can factor 2^m times some integer into those two factors.

---

* $(2^{M-1} - z)^2 = 2^{2M-2} - 2 \cdot 2^{M-1} z + z^2 = 2^M (2^{M-2} - z) + z^2 \equiv z^2 \mod 2^M$

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

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 & 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 & q. Then when I solve these equations, ~~for~~ z & y, ~~they~~ 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.

---

* $z - y = 4 \cdot 2^{i'} p$

$z + y = 4 \cdot 2^{j'} q$

$2z = 4(2^{i'} p + 2^{j'} q) \Rightarrow z = 2(2^{i'} p + 2^{j'} q) \Rightarrow z$ even $\Rightarrow y$ even

† j must be ≥ 1; both $z + y$ are odd so their sum is even.

‡ $y < 2^{m-2}$, $z < 2^{m-2}$; $y + z < 2 \cdot 2^{m-2} = 2^{m-1}$

The last case to try is $j=1$

$$\Rightarrow i \geq m-1$$

$$2^{m-1} \leq 2^i p = z - Y < 2^{m-2} \qquad \text{OR} \quad P = 0$$

Therefore, we must have $P=0$, or $z=Y$ ∴ there is only one solution.

THE RECURRENCE

Now we need the recurrence. I solve it for $m = 3, 4, \ldots, 28, 29$.

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

We are using the lower m bits to represent C.

If C is negative, I go thru 2's complement + take the bottom m bits.

$$C_3 = 1 \qquad (\text{since } C \equiv 1 \bmod 8)$$

Let $z_3 = 1$.

Suppose for any m we have

$$z_m^2 \equiv C \bmod 2^m \qquad (\text{true when } n = 3)$$

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 + 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 + 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 + it rounded up.

This was done, and on 29 occasions theses 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.

TWELFTH LECTURE, THURS NOV 12, 1970

In the last few days 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.

~~Now let us look at some other aspects of the SQRT program.~~

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 cuberoot, exp, log, & trig functions. The last three require a very different point of view & they are much harder to cope with.

Questions: 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, altho none would actually reach the bound.

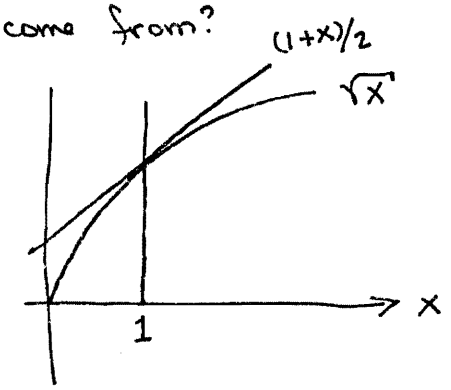
OTHER ASPECTS OF SQRT

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 (what to do is a separate argument)
      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
      GO TO 1
      END
```

Notice that this uses a crummy first approximation to $\sqrt{x}$. Where did it come from?



$(1+x)/2$ is tangent to the graph of $\sqrt{x}$ at $x=1$, & 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 + 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. It's 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; ~~similarly if the first approximation is a much smaller~~. since no machines have an infinite exponent chain range, convergence will eventually get faster. It could conceivably require a thousand iterations to get 1 correct digit, tho; then 7 more are enough. So it is ~~not~~ 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 squareroots 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. (See 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 a 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 squareroot. 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, & the right shift necessary

reduces the error to 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 machine by the following dodge:

$$SQRT = Y - (Y - X/Y)/2.$$

This code takes advantage of the fact that on most machine, 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, 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, tho, 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, & 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.

---

* $Y - X/Y$ is tiny, so it has lots of zeroes. Dividing by 2 on any even base machine can at worst add one digit to the number, & that can still be represented exactly.

By using ~~this~~ 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.*

(*footnote 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 Household on Numerical Analysis, 1953, and by John Todd in some numerical analysis notes that he's been using for the past 40 years.)

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.

OTHER FUNCTIONS

It gets more interesting when you consider what to do with functions other than the SQRT. You cannot always say that you will compute such a function to within a unit in the last place. Half a ~~the~~ unit in the last place is not achievable, because that is the table maker's dilemma. To be able to compute a function to within 1/2 ulp, it may first be necessary to compute it precisely, that may require infinitely many digits, if the value is exactly half way between two machine representable numbers. To make the correct decision you have to discover that. For the SQRT, any number whose squareroot was half way

between 2 machine numbers would not be representable to single precision; so the problem doesn't arise in SQRT. We saw that we had trouble only when 4X is very near an odd square; but it couldn't be equal to that odd square because 4X is even.

It isn't clear why the dilemma cannot occur for, say, the exponential routine. It is possible, altho we have every reason to doubt it. Could you construct an argument X, such that e^X was exactly half way between 2 machine numbers? *

(*footnote Actually, e^x is a bad example. It is known, I think, that for all rational x, e^x is trancendental (not rational), except for $e^0 = 1$. A similar result then follows for the logarithm. And similarly for the sine + cosine. But there could be other values where the issue is in doubt.)

Question: What is the significance of a number being trancendental?

Answer: A trancendental number or an irrational number number cannot lie exactly half way between 2 machine representable numbers + the table maker's dilemma will not arise. But the dilemma can be arbitrarily closely approximated.

Question: Using the infinite series you can get to within a half ulp.

Answer: Yes, you can compute them as accurately as you like. And you know if you compute them accurately enough you can decide. But if you didn't know that the result couldn't be halfway between 2 machine numbers, you might have to compute to infinite precision, because no error, however small, would enable you to render a decision.

REASONABLE BOUNDS FOR OTHER FUNCTIONS

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

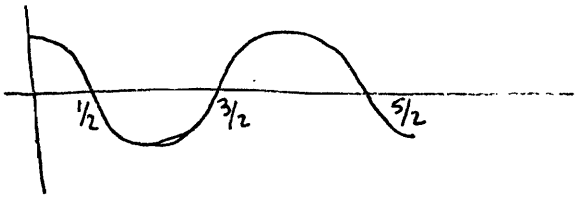
SQRT	< .50000163
LOG, QBRT	< .52 *
EXP	< .77
CABS	< .854
COSPI, DSQRT SINPI, DQBRT	< 1.0

(* footnote The .52 for QBRT is an acknowledgement that it is not a sufficiently important function to bother getting a better bound on the error. For this error, however, I was able to compute all the arguments for which the error was approximately that big. We'll see how that was done later.)

COSPI & SINPI

COSPI(X) is what you ask for when you want to compute $\cos(\pi x)$.

$\cos(\pi x)$ vanishes when x is half an odd integer; the routine does vanish exactly at those points.



graph of cospi(x)

The claim that the error is at most 1 ulp is reasonable, even near a zero, because we know exactly what the function looks like near its zeros.

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

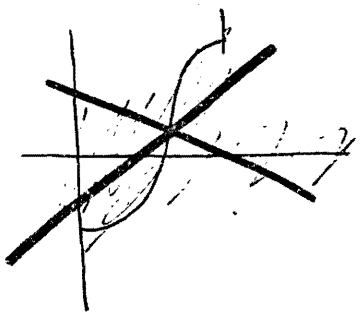
You find out what ξ is by subtracting half an odd integer (representable precisely for integers of decent size), and compute as accurately as you want using the power series.

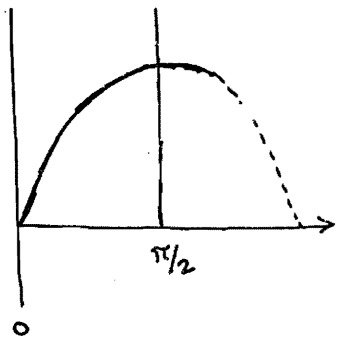
The reason I'm pointing all this out is because for functions like cos & sin, even tho we know where the zeros are & how the function behave near them, the roots are half integer or integer multiples of π, & we don't know π, not exactly. We know it to a large number of decimal digits, but we can't even represent it to in the machine to as many digits as we know. Thus we are unable to say exactly where the functions vanish.

COMPUTING TRIG FUNCTIONS WHEN π UNCERTAIN

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

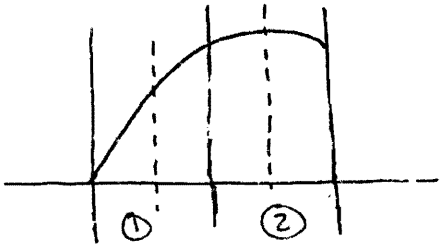
Suppose I wish to compute sin x. Since sine is periodic, the approximating function need not be repeated.





Need only consider this arc in computing sin x

What is conventional to do is to have 4 intervals



In region 1, approximate sin x by an odd function; in region 2

approximate what amounts to cos x by an even function. Each arc is really only half as long by symmetry arguments; you build up the whole function by piecing together these arcs. In order to use these approximations, you have to reduce the given argument to 1 of those 4 intervals. That means dividing by some integer or half-integer multiple of π.

You have to compute + represent:

$$\frac{x}{\pi} \text{ or } \frac{x}{\pi/2} \text{ or } \frac{x}{\pi/4} = \text{integer} + \text{fraction}$$

The integer tells you which interval + which sign to use (that's called quadraproduction); the fraction says how far to move in that interval.

Then someone may say, why not use a representation that does not involve this argument reduction. There are infinite series after all. But look what happens in sin x for a moderately large argument.

$$\sin x = x - x^3/6 + x^5/120 - \ldots.$$

Say x = 100. How many terms will you have to carry? What if x = 10,000? The series very quickly becomes useless, not matter how much work you were willing to perform. It is not because you have to compute a large number of terms; the problem become acute when you realize that most of the digits you compute are going to cancell off.

COMPUTING SIN 100

What happens for x=100? You know sin x cannot exceed 1, but the first term is 100. The leading two digits of 100 have to cancell off. $x^3/6 = 10^6/6$; the gives 5 digits to be cancelled off. $x^5/120 = 10^{10}/120$; 8 digits that must be cancelled.

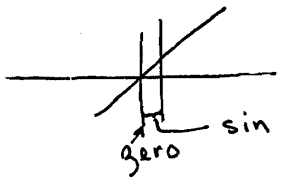
After this the terms get smaller; but you see you'll have to carry 8 decimal digits over and above the 14 you wanted in your answer. That is a more serious fact than ~~the~~ having to compute many terms; they just use a do loop & take a few microseconds. But the extra digits require double precision, & that is not done with a do loop. That takes a D.P. declaration & means the whole D.P. package sits in core

Thus, while it is not impossible to do things this way, it is impractical.

ACCURACY FOR TRIG FUNCTIONS

To make things more interesting, suppose I want to say my result is accurate to within a few units in the last place.

How close to a zero of $\sin x$ can we come? When you are close to a zero, $\sin x \doteq x$ (the slope of the graph is nearly ± 1).



$\sin x$ is nearly equal to this distance

How small can that distance be for numbers representable in the machine? If you represent numbers to 48 bits, you can approximate a root to within 96 bits, by a dodge.

You want to represent $m\pi$ by a number that for all practical purposes is an integer. (it has to be rational in the machine).

$$m\pi \approx P/q \qquad q \text{ is a power of } 2$$

you are representing π by:

$$\pi \approx P/m \times 2^{-t} \qquad p, m \text{ are each 48 bits}$$

There is a theory that says if you allow yourself integers of a certain number of digits, you can approximate irrational numbers to at least

twice as many digits as in either numerator or denominator. That's reasonable since you have twice as many digits to play with.

For certain, abnormal numbers, that is the best you can do. For most numbers, you do better.*

Will just assume we can match the zero to 96 significant bits. Then you'll ~~need~~ want another 48 bits. ~~that brings you up to about 150 bits.~~ It looks like you'll have to carry 150 bits after the binary point, to get 47 or 48 that are correct. Not to mention the ~~xx~~ digits before the binary point that are going to cancell. Now you see the utter impracticality of it all.

Question: It appears there are several reasons for wanting to reduce x. One is the fact that you'll get overflows. That seems even more important than questions of precision.

Answer: Of course, if x is enormous, x^3 would overflow before you got anywhere. But that situation could be coped with, by whatever means you used to cope with multiple precision. If you have to assign extra words to the right, you wouldn't mind assigning a word ~~to~~ for the exponent.

More to the point is if x is small; then x^3 may underflow & you may get all kinds of messages that have no significance at all. In cases like this, x is already a very good approximation to $\sin x$. If $x = 10^{-100}$, $\sin x = x$ is correct to something like 100 decimal digits.

* See Hardy & Wright's book on the theory of numbers.

Question: Can't you tell people something who want to compute things like $\sin 10^{-100}$? Like maybe to rephrase it?

Answer: I really haven't explained how I'm going to do $\sin x$. I was only showing why the obvious ways won't work. You need 150 digits to the right of the point if x is close to a zero, and some interesting number to the left if x is large. The conspiracy is getting worse; that why we don't do it that way.

QUADRAPRODUCTION

So we use quadraproduction. That gets rid of needing digits to the left of the point. But that has not gotten rid of the problem of carrying digits to the right. In some respects we have made that problem worse.

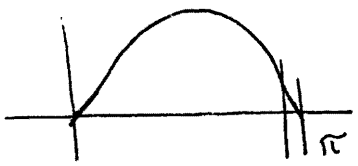
Remember, we don't know the value for π. And no matter how many digits we put in for π, we can't do the division, $\frac{x}{(\pi/2)}$.

You compute: $\frac{x}{(\pi/2)}(1+\xi)$ ξ is at most 1 ulp of the precision you are using for π + the division.

You are going to commit at least 1 rounding error in the division. And you have already made an error in π.

You have effected quadraproduction not on x but on $x(1+\xi)$. Already, you are computing the sin of the wrong angle. You can imagine what that will do if x is close to a zero. You just moved the argument. You are computing for the wrong angle, so you can't possibly get sin x correct to a unit in the last place for any sort of moderately large argument.

Say you do division to double precision, and $x \approx \pi$.



you have moved x by a unit in the last place of double precision; the closest x can come to a zero is about 1 ulp of single precision; you still have roughly a single precision word to play with.

The situation isn't too bad at π, 2π, or 3π. But how about $10^5\pi$? You'll have lost 5 decimal digits.

ERROR IN SIN(x) & COS(x)

How you actually compute the sin is not pertinent to this class. What is important is that you have to think about what you can compute in a rather different way than you might be accustomed to. Namely, that whenever you ask the machine to compute SIN(x), you can be fairly confident that that is not what it is going to do. The best you can hope for is:

$$SIN(X) = (1+\epsilon)\ \sin(X(1+\delta))$$

$|\delta| \sim$ 1 ulp of d.p.

$|\epsilon| \sim$ 1 ulp of s.p. (answer has to packed into a single precision word)

That is the most you can ask for, unless you want the division done to more than single precision. *

---

* If you are very interested in the values of the functions near their zeroes, you'd normally use SINPI or COSPI. People [illegible] in this situation are usually computing sin(πx) anyway.

For the trig functions, you simply cannot [illegible] always compute to within 1 unit in the last place.

You could reasonably expect COS(X) to satisfy:

$$COS(X) = (1+\gamma)\cos(X(1+\delta))$$

where the δ is the same as in SIN(X)
$|\gamma| \sim 1$ ulp of s.p.

You get the same error δ because you do exactly the same division + then interpret the integer part differently.

SIN(X) + COS(X) computed for very large operands X may be wrong, but nevertheless they are the sin + cos of some reasonable argument. Consequently:

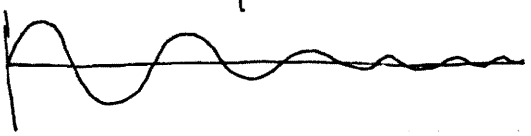
$\frac{\sin}{\cos} \approx \tan$ to within a couple ulps

Also $\sin^2 + \cos^2 \approx 1$ will also hold, if you use the same argument reduction for both.

Question: If you pass the sin routine a very large argument, the argument itself may be represented rather poorly.
Answer: That's right. If X is really large + it is at all uncertain (consider, where did X come from, another computation maybe?), the uncertainty in X will cover a large interval, and the sin + cos could oscillate several times in that interval. That does happen. And when it does, I think the most we could hope for is some kind of internal consistency.
Question: Shouldn't there be a diagnostic?
Answer: There is sometimes. But it is hard to say whether this is an error or not. In some asymptotic formulas, altho the sins + cos's oscillate in an uncertain way, they are multiplied by

12-16

things that are very small. Such as in the bessel functions:



it approaches $\frac{\pm 1}{\sqrt{x}} \sin x$

People sometimes do have formulas in which they want trig functions of large arguments. But the uncertainty gets less important as the argument gets larger, as there later terms are a small contribution to some series.

Question: Wouldn't you suggest to people that they write their own sin' routine, so that the argument is in some interval in which you can actually compute the sin, using the system subroutine. If they are just going to get garbage, they should get a result that is essentially zero.

Answer. But it really isn't garbage, you see. The function is only important where it is big (in the above picture, say), & there you get reasonable accuracy, sometimes. Remember, you only get troubles like this on our machine for $x \sim 2^{40}$ or so. That's gargantuan. ~~For~~ for numbers like 10,000, or 100,000, ~~will be treated~~ ~~[illegible]~~ the sin & cos will have deteriorated a bit, but this is generally not serious ~~for the applications I have in mind~~ for the applications involved. It is rather ~~difficult~~ difficult for somebody to go thru the analysis that tells him he should do something different rather than accept the values as computed.

For people who use abnormally large arguments & may not realize what they are doing, you're right — they should be given a diagnostic. But that involves a decision — where to draw the line. Should you tell him when he's lost all his digits, or only half of them, or what? On IBM equipment, it is customary to issue a diagnostic when changing the argument by 1 ulp can run you thru an interval ~~can~~ comparable to π. On the 360, this means $x \sim 10^6$. On the 7094,

$x \sim 10^8$. My programs don't give a diagnostic. They just say here is what you get, & if you are worried about it, use the SINPI & COSPI routines, for which no diagnostic is needed. Of course, for roughly half the machine number arguments, SINPI & COSPI give you +1, -1, or 0. ~~the arguments are~~ The numbers are mostly integers times powers of two; thus SINPI & COSPI usually return 0 or +1. But that's alright. We now have a reasonable way of interpreting what you get & why you get it.

WHAT YOU CAN EXPECT FROM ERROR ANALYSES GENERALLY

I guess I'm introducing you to the rather interesting notion that instead of being able to say you have gotten something that is wrong by a unit in its last place, it may be that you'll be obliged to say that the answer you have differs by a unit in its last place from the exact answer of a problem that differs by some small amount from the problem you originally posed. And that is normally what is considered to be a successful error analysis. But even a statement like this usually cannot be made. For nontrivial problems like solving a set of linear equations, even using a decent numerical method, the best published analyses say that the answer is no worse than if you had solved precisely a problem with slightly perturbed data. That is not the same thing as saying that the answer is to within a unit in the last place of the exact solution of a problem with slightly perturbed data. They're quite different statements; the first is much weaker than the second.

LINEAR-EQUATION-SOLVING EXAMPLE

As an example, consider solving a set of linear equations,

$$A x = b.$$

This is best thought of as a linear mapping between two spaces; the vector (b) sits in one space, vector (x) in another, & they are related by some linear mapping.

Of course, A & b are a little bit uncertain. We would like to think of A as belonging to some space also; if we don't think of it that way, it is a little hard to talk about errors in A. So what we do is say (A,b) belongs to a space. The solutions maps the combination of the data (A,b) onto some vector (x). It is linear in b; it is almost but not quite linear in A.

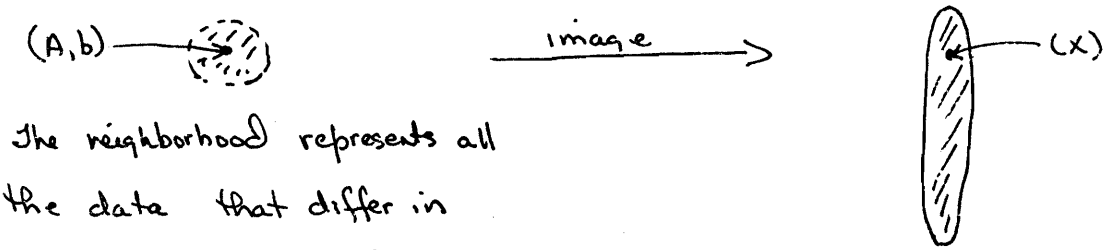
$$x = A^{-1} b \qquad \text{if } A \text{ is nonsingular}$$

$$dx = d(A^{-1})\, b + A^{-1}\, db$$

$$= -A^{-1}\, dA\, A^{-1} b + A^{-1} db$$

With respect to small perturbations dA & db, the variation dx is linear. Altho the whole transformation is not linear, in the small it is approximately linear. The above equation for dx is exact for differentials; for variations it is true to within terms of order $(dx)^2$, etc.

Now I can discuss not the image of the point (A,b), but rather the image of a neighborhood of that point.



The neighborhood represents all the data that differ in some small way from the data that I see.

The image is usually a needle shaped region.

When you solve the linear equations, you'd like to be able to say that the solution lies in that needle shaped region, to within 1 ulp.

← would like to say solution is in this neighborhood

But nobody has ever proved such a statement, nor will they. All that they can prove is that the solution lies in a region roughly spherical that is not appreciably bigger than is needed to contain that needle-shaped region.

← can prove the solution is in this region

When you try to solve a set of linear equations what you can hope to get, if you used published analyses, is an answer that is no further away from what you wanted than are some of the answers belonging precisely to problems that differ from your problem by amounts that you consider negligible.*

That is a typical error analysis. It is not as good as I would like.

HOW APPROXIMATE ARE YOUR RESULTS

(pervasive in numerical analysis)

The sin & cos have thus introduced you to the notions that you cannot compute approximately the right answer to your problem, you can only hope to compute approximately the right answer to very nearly your problem. If you can do that, people will say you have used a stable numerical method.

* If you solve the equations to double precision, the answer will lie in the needle-shaped region. But no theorem has been published to this effect, & none probably never will be.

There are exceptions, which correspond to rather peculiar measures of what we mean by approximately. For example, if you examine the quadratic equation, $Ax^2 - 2Bx + C = 0$, it is clear that A, B, & C are pieces of data. But what about the 2 on x^2? Is that datum or part of the structure of your mapping? If you think of x as a datum susceptible to variation, so that you might have written $x^{2.00000003}$, then there could be an infinite number of solutions, whereas the equation in x^2 has only two.

The way that the solutions vary with changes in A, B, C, & 2 are different from the way the solutions vary with changes in A, B, & C only. So when you talk about a problem very near yours, you may be fixing things that might otherwise have been thought of as data, subject to variation.

WHAT SHOULD BE DATA

In some cases, the issue, as to what should be data & what shouldn't, is not altogether clear. What should be allowed to vary?

As an example, I'll talk to you as a CDC engineer or programmer would. He would say that nobody can ever know exactly what the operands should be (I dispute that, by the way), [illegible]

"If you want to compute LOG(x), you should be willing to accept

$$LOG(x) \approx (1+\epsilon)\log(x(1+\delta)).$$

(That is, he is willing to perturb the argument). You don't know what x is anyway, so why should you care if I change it a little bit?"

Remember the grad student working on wing design? He cared.

I would care a great deal if I were computing A^B. You write is as: $A^B \approx EXP(B * LOG(A))$.

[redacted]

If B is a large number you find you didn't compute A^B but rather something else. If B is large, A had better be close to 1, or you'll overflow. But if A is very close to 1, and you have to take

[redacted] $\log(A(1+\delta))$

where $1+\delta$ is also close to 1, the log can be changed drastically, say by a factor of two. Then when you do the rest of the computation, you're dead.

The engineer would say that's perfectly reasonable because you don't know A + you don't know B. But you might want to dispute that. 1+δ SHOULD NOT BE IN YOUR ARGUMENT

It is my judgement that the $(1+\delta)$ in the log function does not belong there because there are perfectly economical ways to compute the log without it being there. You just have to be a little bit careful. It amounts on our machine to changing statements like $X - 1.0$ to $(X - 0.5) - 0.5$. In the first case, if X is close to 1 the answer may be zero, whereas in the second case, the difference is taken correctly.

By using a better approximation and a bit more care, you should be able to get a log function in which the $(1+\delta)$ perturbation term does not appear. You'll have to agree that it's [redacted] preferable to think of the logarithm without that term.

My argument for wanting to get rid of those terms when you can is that they make [redacted] the structure very different from what you're used to, + from what you expect.

that is my argument for putting the perturbation in SIN(X(1+δ)) down to double precision. For arguments in the range of π or smaller, you could eliminate δ by slightly increasing ε. We saw that perturbing x by an ulp of d.p. might change the single precision answer by a few units in its last place. So you say ε is 5 units, instead of 1, * not have to talk about δ at all.

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

QUADRATIC EQUATION SOLVER

Now we will tackle a real live problem, an error analysis of a quadratic equation solver. We might actually accomplish something - maybe even get the program to work.

The equation to be solved is $Ax^2 - 2Bx + C = 0$.

---

* Hernando Kuki: "Getting rid of them (factors like the 1+δ) gives you the strictest accuracy requirement for a subroutine that you could conceive of. Therefore it gives the simplest goal for the programmer to aim at, so far as accuracy is concerned. And in some computation, for example with integer arguments or assuming all prior computations went meticulously well, where there is no error in the argument, the benefit is real. And, it is simpler to explain to users." However, it may cost diamond where mere glass may have served.

The code might go as follows.

```
      D = B**2 - A*C                    compute discriminant
      IF( D .LE. 0)  GO TO  1
      .
      .
      .
      Complex or coincident roots (RR ± i RI)
    1 RR = B/A
      RI = SQRT (-D) / A
```

What have I actually computed?

$d = b^2(1+\epsilon_1) - ac(1+\epsilon_2)$ $\quad |\epsilon_1| < 2^{-46}$

that's what gets stored. $\quad |\epsilon_2| < 2^{-46}$

$rr = b/a\,(1+\epsilon_6)$ $\quad |\epsilon_6| < 2^{-47}$

$ri = (\sqrt{-d}/a)(1+\epsilon_7)$ $\quad |\epsilon_7| < 3\times 2^{-48}$

two errors

In what way are rr & ri related to the roots of the equation we set out to solve? They might not resemble the roots too closely.

Where the roots are complex, you can demonstrate that rr & ri each differ by at most a couple of units in the last place from the exact components of the roots of a quadratic whose coefficients differ at most by a unit or so in the last place from the coefficients given in the problem. You do this by juggling the ϵ's in such a way that you can say that the equation satisfied by these numbers has coefficients that are perturbed relative to the original equation.

The thing that seems to bother students most is that there

is a certain arbitrariness in the way you do perturbations. There is no unique natural mathematically preordained choice of what numbers close to rr + ri should be thought of as the roots to some equation. There are lots of choices, & even then, there is no preordained choice for the coefficients; I can multiply thru by any factor close to 1 + & have another equation.

So there is a great deal of arbitrariness.

THIRTEENTH LECTURE, NOV 17, 1970

QUADRATIC EQUATION SOLVING

I will first consider the errors due to roundoff, and later discuss the problem of under/overflow. For purposes of analysis, it is simpler if you separate the two problems.

Let's look at a FORTRAN program, written with the assumption that no over/underflow occurs. We'll see what effects rounding errors have.

We are solving: $A x^2 - 2Bx + C = 0$

Assume $AC \neq 0$.

Notation: Capital letters are FORTRAN variable names; small roman letters are the values as they appear in the machine. Small greek letters are the relative errors, which are bounded to a unit or so in the last place of the precision carried.

THE PROGRAM

```
D = B**2 - A*C
IF (D .LE. 0) GO TO 1
```

relations for things actually in storage

At the beginning in storage we have a, b, c as the values of A, B, & C, the variables

$$d = b^2(1+\epsilon_1) - ac(1+\epsilon_2)$$

the error due to the subtraction is incorporated into the error in the operands (the operands are perturbed)

Real *, distinct roots R1 * R2 ; D > 0

S = B + SIGN (SQRT (D), B)

the sign of b is overwritten on the sign of $\sqrt{d}$; we are really adding magnitudes

take sign(0) = 1 in case B = 0 →

$S = (b + \sqrt{d} * \text{sign}(b))(1 + \epsilon_3)$

ϵ_3 combines the errors due to taking $\sqrt{d}$ and doing the addition *

$|\epsilon_3| \sim 2^{-46}$

$S = b(1+\epsilon_1) \pm \sqrt{d}(1+\epsilon_2)$

R1 = S/A

R2 = C/S

$r_1 = s/a(1+\epsilon_4)$

$r_2 = c/as(1+\epsilon_5)$

R1 can be computed because A does not vanish. R2 is legitimate only if S doesn't vanish ; but S cannot vanish because D > 0 (so SQRT (D) > 0)†

(footnote on next page)

---

* to illustrate why we can do this, consider adding two positive numbers.

$x > 0,\ y > 0$

$S = x(1+\xi) + y(1+\eta) = (x+y)(1+\sigma)$

$\sigma = \frac{x\xi + y\eta}{x+y}$ This is just a weighted average of ξ & η, so σ lies between them. Thus the bounds on $\xi + \eta$ are a bound on σ. This is true on all machines for adding magnitudes.

C Complex or coincident roots $RR \pm i\,RI$ when $D \le 0$

```
1   RR = B/A
    RI = SQRT(-D)/A
```

$r_R = b/a\ (1+\epsilon_6)$

$r_i = \sqrt{-d}/a\ (1+\epsilon_7)$

This represents a FORTRAN program as might be written by a student who is not completely naive, because of using the trick to get R2. He has realized that if $B^2 \gg AC$, then D is very close to B^2 & he doesn't want to compute $B + \sqrt{D}$ and $B - \sqrt{D}$. One would have lots of cancellation & bring up lost digits that had been made into zeroes. This program will solve quadratic equations that stump the usual approach. You can find reasonable looking quadratic equations for which failure to use the C/S trick will give incorrect results. yet there are examples for which this method will cost you half your digits in accuracy.

THE TRICK FAILS

The problem occurs when B^2 is close to AC. Take N as a huge number, but not so large that $N \pm 1$ and N^2 are not representable. Consider the equation

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

$$x = \frac{N \pm \sqrt{N^2 - (N^2-1)}}{N+1} = \frac{N \pm 1}{N+1}$$

so the two roots are: 1 and $\frac{N-1}{N+1}$, precisely.

Now consider what happens if rounding occurs as in our model, namely that computing B^2 gives you $b^2(1+\epsilon)$ where ϵ is a few ulps. Similarly for $A * C$.

Compute the discriminant

$$N^2(1+\Theta(2^{-47})) - (N^2-1)(1+\Theta(2^{-47})) \approx N^2 * 2^{-47} = D$$

Take the squareroot of that & compute the roots

$$r_{\pm} = \frac{N \pm 2^{-24} * N}{N+1} \simeq 1 \pm 2^{-24}$$

Only half your digits are correct.

Using these trick formulas does avoid the problem of cancellation in subtracting magnitudes in $B - \sqrt{\ }$, when $B^2 >> AC$.

To eliminate the problem of losing digits, it will be necessary to carry double precision for one operation, namely the subtraction $B^2 - AC$. Then you only need a single precision squareroot. It's only that subtraction: you compute B^2 precisely & AC precisely, using double precision; subtract, and then take the single precision result.*

(Notice I haven't said how small the ϵ's are, only that they are a few ulps of the precision being carried.)

ERROR ANALYSIS

At this point, I haven't demonstrated that double precision is needed, except by that example. So we might want to know, how good is the

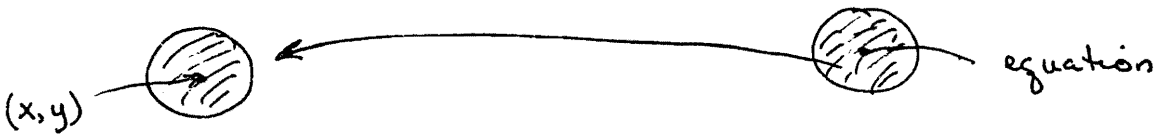
---

* On a log machine, B^2 & AC would be formed precisely in single precision, but double precision would be needed to get their difference. The answer may be an infinite string of digits but you only take the leading single-precision-word's worth of them. Even then the difference might be so small that it underflowed.

solution using this program. If you can perform some sort of error analysis then you can deduce how much precision is sufficient even if you don't know that it is necessary. I'll show how to use the ε's to analyze what you got + how to relate that to what you started with, + what you want.

What you have is almost the roots of almost the given quadratic. (This is normally as good an error analysis as you would expect.) To prove this is a technical detail. Thinking of it that way is what is important. You must be willing to speculate on whether such a statement is true.

The proof is time consuming because there are many arbitrary decisions. I may have almost the roots of one quadratic which is almost mine, but there are infinitely many such equations. If I vary the roots by a very little bit, that changes the coefficients by only a little bit.



The roots lie in a neighborhood of a point that is exactly the roots of an equation in the neighborhood of mine.

If I alter the roots slightly, I get another quadratic near mine. There are an infinite number of such choices.

The analysis necessarily cannot be unique. That causes the the technical trouble. The problem is not a hard one to solve, but

you are bound to make certain arbitrary decisions. If you make a wrong arbitrary decision (+ some are really wrong), you won't get an answer.

~~It takes a set of formulas. If you assume those & get th~~

Question: In talking about linear equations before, you said that the image space was sensitive in one parameter. Is that the case here?

Answer: Yes, the image space will be needle-shaped. When two roots are close together, something goes wrong. The roots are not simply differential functions of the coefficients other than when they coincide.

Question: If you take a particular point in the domain, what is the shape of the range, in the general case?

Answer: In the general case, the closer you get to a singularity, the more oblong the image becomes. It is characteristic of what we call singularities, that near them, sphere-shaped neighborhood are mapped into needle-shaped regions. A singularity is a point where the derivative (or Jacobi matrix of partial derivatives) is nonexistent or singular. If it is nonexistent, clearly something terrible happens; if it is singular, the inverse transformation doesn't exist; or both Jacobians may fail to exist. In the neighborhood of such a point you expect this tearing phenomenon.

Question: If you do an analysis that shows how much our roots differ from the exact roots, you can get a big error. You say you can remove most of that big error by putting it into

the original equation as a smaller error. Can you put all the error into the original equation?

Answer: We will see that you can't. [redacted].

A PERTURBED EQUATION

$$\tilde{r}_j = r_j(1+\delta_j) \qquad j = 1, 2, R, \text{ or } I$$

I want to show that there are roots $\tilde{r}_j$ that differ from our approximations by only a few ulps. Those roots exactly satisfy a slightly perturbed quadratic:

$$\tilde{a}x^2 - 2\tilde{b}x + \tilde{c} = 0$$

where $\tilde{a} = a(1+\delta_a)$ $\quad \tilde{b} = b(1+\delta_b)$ $\quad \tilde{c} = c(1+\delta_c)$

I'll give you an answer without saying how I got it. If you want to find out how I did get my answer, you could try to find a different answer. You'll either give up & accept mine, or find a different one.

I'll make the arbitrary decisions that:

$\tilde{a} \equiv a$ $\quad \tilde{b} \equiv b\sqrt{1+\epsilon_1}$ $\quad \tilde{c} \equiv c(1+\epsilon_2)$

these come from $d = b^2(1+\epsilon_1) + c(1+\epsilon_2)$ which has important rounding errors.

In effect I have reasoned as if the only important errors were made in computing d; we know from the example that those are pretty important. The question then is, if I hadn't made those errors, could the rest have been essentially correct? If I can later show that making ϵ_1 & ϵ_2 errors of a few units in the last place of double precision accomplishes that, then we'll be finished. I'll be able to show that by a simple argument that has nothing to do

with rounding errors.

I did not have to choose $\tilde{a}, \tilde{b}, \tilde{c}$ as I did, but this choice will rescue the analysis later.

SOLVING FOR THE $1+\delta_i$'s

Now all that is left is to solve for the δ_j's. If $d \le 0$,

$$\tilde{r}_R = r_R \sqrt{1+\epsilon_1} / (1+\epsilon_6)$$

$$\tilde{r}_I = r_I / (1+\epsilon_7) \qquad *$$

If $d > 0$, things get very complicated so I'll define a few things.

$$\epsilon_8 \equiv (\tilde{b} + \sqrt{d} \cdot \text{sign}(\tilde{b})) / (b + \sqrt{d} \cdot \text{sign}(b)) - 1$$

ϵ_8 tells me how well numerator & denominator approximate each other; the numerator is computed using perturbed coefficients, the denominator using unperturbed coefficients, except that both use d; there is no $\tilde{d}$. (This is why $\tilde{a}, \tilde{b}, \tilde{c}$ were so chosen; so that $\tilde{d} = d$)

$$\epsilon_8 = \frac{\epsilon_1}{(1+\sqrt{1+\epsilon_1})(1+\sqrt{d}/|b|)} \qquad \pm \text{ so } \epsilon_8 \text{ is very small}$$

the denominator is bigger than 1, ϵ_1 is small

$$\tilde{r}_1 = r_1 \left(\frac{1+\epsilon_8}{(1+\epsilon_4)(1+\epsilon_5)} \right)$$

$$\tilde{r}_2 = r_2 \left(\frac{(1+\epsilon_2)(1+\epsilon_3)}{(1+\epsilon_7)(1+\epsilon_8)} \right)$$

$*$ $r_R = b/a\,(1+\epsilon_6)$ $\qquad \tilde{r}_R = \tilde{b}/\tilde{a} = b\sqrt{1+\epsilon_1}/a = \frac{b}{a}(1+\epsilon_6)\frac{\sqrt{1+\epsilon_1}}{(1+\epsilon_6)}$

$= r_R \sqrt{1+\epsilon_1}/(1+\epsilon_6)$

$r_I = \sqrt{d}/a\,(1+\epsilon_7)$ $\qquad \tilde{r}_I = \frac{\sqrt{-\tilde{d}}}{\tilde{a}} = \frac{\sqrt{-d}}{a}(1+\epsilon_7)/(1+\epsilon_7) = r_I/(1+\epsilon_7)$

$$\pm \epsilon_8 = \frac{\tilde{b} + \sqrt{d}\ \text{sign}(\tilde{b})}{b + \sqrt{d}\ \text{sign}(b)} - 1 = \frac{b\sqrt{1+\epsilon_1} + \sqrt{d}\ \text{sign}(b\sqrt{1+\epsilon_1})}{b + \sqrt{d}\ \text{sign}(b)} - 1 = \frac{b\left(\sqrt{1+\epsilon_1} + \sqrt{d}/|b|\right)}{b\,(1+\sqrt{d}/|b|)} - 1$$

$$= \frac{\sqrt{1+\epsilon_1} + \sqrt{d}/|b| - 1 - \sqrt{d}/|b|}{1+\sqrt{d}/|b|} = \frac{(\sqrt{1+\epsilon_1} - 1)(\sqrt{1+\epsilon_1} + 1)}{(1+\sqrt{d}/|b|)(\sqrt{1+\epsilon_1} + 1)} = \frac{1+\epsilon_1 - 1}{(\ )(\ )} = \frac{\epsilon_1}{(1+\sqrt{d}/|b|)(\sqrt{1+\epsilon_1}+1)}$$

That is what we get solving for the $1+\delta_i$'s. I have demonstrated that we have almost the solution of almost our quadratic. What is the point, since we cannot say that we have almost the roots to exactly our quadratic nor can we say that we have exactly the roots to almost our quadratic.*

* This is illustrated when B is tiny. When D is computed, B^2 may very well get lost, so $D \approx AC$. Then in computing S, B could also get lost; the computation would go through as if $B = 0$. ($AC > 0$ is necessary for this to happen). The roots are exactly the solutions to an equation for which $B = 0$; we cannot say we have approximated our equation, not in the floating point (relative error) sense of approximation.

~~There is another way to think of the problems of approximation that I spoke of.~~

Question: You did the computation & came up with some answers, that agreed with the answers of the perturbed equation within a few ulps. However, during the computation, certain things happened that made it appear that $B = 0$.

Answer: You've lost something. In the calculation, I was showing that the numbers printed out as roots are almost the roots of almost the equation you started with. I considered two ways of simplifying that. We accepted that we could not say we had almost the roots of exactly our equation; we cannot simplify things by throwing away the perturbations on A, B, & C, & saying that the coefficients have been preserved. The roots would be wrong by more than a few ulps.

The other simplification is to say that the results printed out are the roots of almost the equation read in; I've just shown that can't be done. In the critical case, the roots would solve an equation for which $B=0$ when, in fact, B is nonzero.

Question: Your point was that in that case, B almost vanished anyway. The relative error is quite large; the absolute error would be small.

Answer. That brings us to the other way of looking at this problem.

ERROR ANALYSIS WITHOUT REFERRING TO THE PROGRAM

A second way of looking at things divorces us a bit from rounding errors. The point of this analysis is not just to allow us to make a simple statement about rounding errors; it has the important additional advantage that if we want to know what the consequences of rounding errors are, we can find out without any more referring to the program. We now have a statement, imbedded in all those ϵ's, whose values you do not need to know beyond the fact that they, the δ's, are small. That is all that is needed to do the rest of the analysis.

The point of an error analysis of this kind is not, primarily, to get best values for ϵ's & δ's. It's main point is to enable you to divorce your thinking from the program. We say the program is numerically stable if we can forget about it, once it has been aptly characterized.

The roots that we have computed, when substituted into the original equation, will almost satisfy it. The error is necessarily small when viewed in this way.

If you compute $ar^2 - 2br + c$, [redacted] for any root we've written down, you'll get a very tiny number. Why? If I perturb a, b, c by a few ulps, & then perturb the r's, I could arrange things to get exactly zero.

$$a(1+\delta_a)r^2(1+\delta)^2 - 2b(1+\delta_b)r(1+\delta) + c(1+\delta_c) \equiv 0$$

That's what we proved; the perturbed coefficients & perturbed roots belong to each other. The value taken by the above equation, [redacted] is whatever value, other than zero, I would get if I perturb each coefficient by a little bit. It is conceivable that the rounding error committed in evaluating the equation could be bigger than the value it ought to have. You could get an answer that looked like zero, even if only half the digits in the root were correct.

Question: It would appear that each root is the root of a slightly perturbed equation, different for each root.

Answer. That's right. Sometimes it is important that one of the roots will satisfy almost your equation, but will not both roots satisfy almost the same equation.

ILL-CONDITIONED PROBLEM; STABLE METHODS in S.P.

If we can satisfy the equation, with numbers that are nowhere near the correct solutions (like having half the digits wrong), we sometimes say the problem is ill-conditioned. It is characteristic of the problem & has nothing to do with the method used in solving. It could have been a good or crummy method.

However, using the method $B \pm \sqrt{B^2-AC}$ to compute the r's can lead to error bigger than can be attributed to perturbing the coefficients by a little bit. That would be an unstable numerical method, because the answers you compute are very much more wrong

man can be attributed to uncertainties in your data, + other places.

The method used in the FORTRAN program is a stable method because it can be characterized by a relation using δ's. If a stable method gives wrong answers, then the problem must be ill-conditioned. It might be that any stable method would give you a wrong answer. The issue is clouded, unfortunately, because we use different ways to measure error; you can confuse the issue by discussing different measures of what you call error; you may not be able to tell if a method is stable or if a problem is ill-conditioned.

If I had said I wanted to do certain things absolutely precisely + allow no error at all in certain parts of the calculation, I could convert an otherwise unstable method into a stable one. But I'm then measuring error in such a way that an error in that certain place is given enormous weight, so big a weight that it clobbers everything + makes the process invalid.

ILL-CONDITIONED + SENSITIVITY

If I can prove a relation like

$$a(1+\delta_a)r^2(1+\delta)^2 - 2b(1+\delta_b)r(1+\delta) + c(1+\delta_c) = 0$$

I can discover how the roots behave in the face of rounding errors. Let us consider only the error in the coefficients + ignore the rounding in the r's, which amounts to chopping a number that might require infinitely many digits.

What are the consequences of perturbing the coefficients? It suffices to analyse the original equation, + the simplest analysis will do. Let a, b, + c be independent variables + the roots, x, be

Dependent variables.

$a x^2 - 2bx + c = 0$ defines $x = x(a,b,c)$

How does x vary with $a, b, \& c$?

$$da\,x^2 + 2ax\,dx - 2b\,dx - 2x\,db + dc = 0$$

$$dx = \frac{x^2\,da - 2x\,db + dc}{2(b - xa)}$$

That's the change in x as a differential; in some sense it is valid only to second order terms in the differentials. We'll analyze this, even tho it is only an approximation.

As long as b and xa are quite different, dx is approximately a linear function of da, db, & dc. When b & xa are very close together, the roots are very nearly equal.

To see that $b \approx xa$ implies nearly equal roots, let x, y be roots, both positive. Then

$$\frac{b}{a} = \frac{(x+y)}{2} \quad \text{so} \quad b = \frac{a(x+y)}{2}$$

$$\frac{c}{a} = xy \quad \text{so} \quad c = axy$$

$$b - ax = \frac{a}{2}(y - x) \to 0 \quad \text{as} \quad y \to x$$

We only get into difficulty when we have nearly-repeated roots. In that case, we'd do a different analysis, like saying $r_+ = b + a$, $r_- = b - a$ & seeing what happens.

If you can compute $(b^2 - ac)(1+\epsilon)$ (ϵ is s.p. error), then everything is fine. But b^2 & ac may almost agree to double precision, &

you don't know what will happen at the end of the d.p. words.

points numbers come under one set of conventions, integers under another.

If you are doing binary-decimal conversions, it is necessary to compute to more accuracy than is requested, in order to have something to round. To get single precision, your table of constants will need to be to double precision, + the conversion done with d. p. hardware, + the result rounded to single precision. Then the job is done correctly except for those miserable cases that fall halfway between; in binary-decimal those cases can be characterized.

Double precision conversion, using double precision hardware is sloppy. You really need some extra bits around and no machine provides those.

FOURTEENTH LECTURE, Thurs Nov 19, 1970

I showed last time that the numbers stored in the machines as the roots differ by no more than a unit or two in the last place from the exact roots of a quadratic whose coefficients differ by no more than unit or two in the last place from the coefficients read into the machine. The value of such an analysis, irrespective of the precision used, was that it made it possible to analyse the consequences of errors in a manner that no longer depended intimately on the algorithm used to solve the quadratic.

We might define a stable algorithm as one that allows us to make the above statement.

If you now want to know how correct the answers are, you can do a calculation that doesn't depend on the algorithm used, except where bounds are concerned. Those bounds sum up everything that you have to know about the algorithm.

THE ANALYSIS

I'll write down the analysis. Remember that nothing has been said about what precision must be carried to achieve a given result. It will be a consequence of the following error analysis that tells us what precision is needed.

The values a, b, & c are read in. Values called the "roots" are printed out, $r_R \pm i\, r_I$ or r_1 & r_2. Each r_i satisfies an equation like:

$$a\,(r_i(1+\delta_i))^2 - 2b(1+\beta)(r_i(1+\delta_i)) + c(1+\gamma) = 0$$

(Note the change in notation)

If you take the roots + perturb them by a few ulps, they both will exactly satisfy the same slightly perturbed quadratic. The only interesting errors will be those caused by perturbations in the coefficients. There will be errors caused by perturbations in the coefficients that will cause the perturbed r_j's ($r_j(1+\delta_j)$) to differ from the true roots by quite a bit. (upto half the digits wrong) of the original quadratic.

NOTATION

Some notation as follows:

Call the true roots of original quadratic r_+ and r_-.

$$r_\pm = \frac{b \pm \sqrt{b^2-ac}}{a}$$

What I'd like to know is how the true roots differ from the $r_j(1+\delta_j)$'s.

Call the true roots of the perturbed equation s_+ and s_-.

$$s_\pm = \frac{b(1+\beta) \pm \left(b^2(1+\beta)^2 - ac(1+\gamma)\right)^{1/2}}{a}$$

We know we have computed the s's to within a few units in their last places. i.e δ_j's are few ulp We want to find the difference between the s's and the r's. If we can find out how wrong the s's are, we'll know how wrong our answers are, since we know the s's to a few ulps.

To make the notation simpler, introduce $t_\pm$:

$$t_\pm \equiv s_\pm/(1+\beta)$$

The $(1+\beta)$ term perturbs the $s_\pm$ by only a few ulps; the t's are introduced because they satisfy a rather simpler quadratic:

$$at^2 - 2bt + c(1+\gamma) = 0$$

where $(1+\tilde{\gamma}) = (1+\gamma)/(1+\beta)^2$

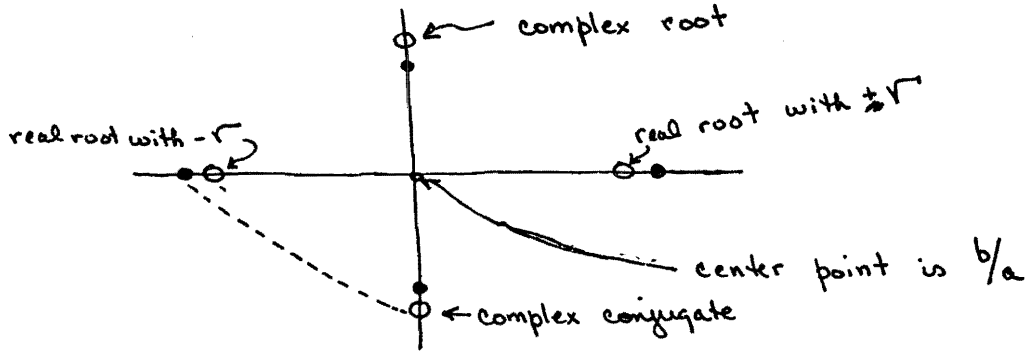
$\tilde{\gamma}$ is thus a greek letter (i.e., error bound) in the same family as γ & β.

$$t_{\pm} = \frac{b \pm (b^2 - ac(1+\tilde{\gamma}))^{1/2}}{a}$$

All I have to do now is analyze the affect upon the roots of a quadratic of a perturbation in its last coefficient. The transformation to the t's could have been made all at once, doing it in steps shows more clearly how it is done, & makes it less mysterious.

MUST LOOK AT CASES

Unfortunately, to analyze the problem requires looking at a large number of cases. That this is necessary is seen as follows: consider the locus of the roots as $\tilde{\gamma}$ varies thru real values. As $\tilde{\gamma}$ varies, the roots fall on one of two perpendicular lines in the complex plane.



For complex roots, the real part is b/a. The imaginary part comes from the $(\ )^{1/2}$ term. The roots have b/a as their average.

The problem is, where are the r's & where are the t's? The r's & t's could both be complex & separated by only a little bit. Or they might both be real & only slightly separated. Or one set might be real

while the other is complex. Then the comparisons may be of little distances along one of those two lines, or longer looking distances connecting the two lines (dotted line in drawing).

We already have the names for our two sets of roots: $r_\pm$ and $t_\pm$. It is helpful to make some assumptions that tell us which r is to be paired with which t.

Question: Do you have an expression relating the computed roots to the t's?

Answer: r_j (computed) $= t_\pm (1+\beta)/(1+\delta_j)$ if r_j real.

I am claiming that the r_j's different from the t's by only a little bit.

ASSUMPTIONS TO IDENTIFY ROOTS

The assumptions are:

(1) $\underset{\text{Im}}{\text{Re}} \{ (\ )^{1/2} \} \geq 0$ we are restricting the radical to lie on the + real or + - imaginary axis.

This assumption makes the corresponding plus & minus signs on the r's & t's match. r_+ goes with t_+; r_- goes with t_-. Even if the roots r & t are not both real or both complex, I'll have a better approximation if I take this matching. So this assumption identifies which roots we are talking about. (Why??)

(2) take $b/a \geq 0$ for definiteness only. If this were not so, I'd simply reverse signs throughout the analysis. (i.e. mean $b/a \geq 0$)

also $a \neq 0$, $c \neq 0$ (if either of these is zero, the problem is very different

The first assumption is crucial in analyzing an algorithm when the answer is more than one thing. If I simply swap the two roots, then for most purposes I would have solved the equation as well as before.* This problem of identity is also critical in eigenvalue problems; you have to

*footnote

decide which computed root goes with which exact root. This decision is not trivial, because it is the essence of polynomials that there should not be any strictly algebraic way to tell one root from another. You'd have to make arbitrary decisions of the foregoing kind. You always have to make an assumption like the first.

Now we express the t's in terms of the r's:

$$t_\pm = r_\pm (1 \pm \delta_\pm)$$

$$\delta_\pm \equiv \frac{\tilde{\gamma}\, r_\mp}{\left(\tfrac{1}{4}(r_+ - r_-)^2 + \tilde{\gamma}\, r_+ r_-\right)^{1/2} + \tfrac{1}{2}(r_+ - r_-)}$$

This was worked out so most terms in the denominator are positive: the $(\quad)^{1/2}$ term has real part ≥ 0 and imaginary part ≥ 0; likewise for $\tfrac{1}{2}(r_+ - r_-)$. There will be no complete cancellation in the denominator. That is, $r_+ - r_- \approx (b^2 - ac)^{1/2}$, is non-negative whether real or imaginary.

Now let's look at some examples of cases.

CASES: COMPLEX ROOTS

Case C; complex $r_\pm = u + iv$ $\qquad u, v \geq 0$

$$|\delta_\pm|^2 = \frac{\tilde{\gamma}^2 (u^2+v^2)}{|v^2 + \tilde{\gamma}^2 (u^2+v^2)| + v^2}$$

The perturbed discrimanant between the magnitude bars could be positive or negative.

Case C1: $v^2 + \tilde{\gamma}(u^2+v^2) \leq 0$

$$|\delta_\pm|^2 = \frac{\tilde{\gamma}^2 (u^2+v^2)}{-v^2 - \tilde{\gamma}(u^2+v^2) + v^2}$$

$$|\delta_\pm|^2 = -\tilde{\gamma}$$

Case C2: $v^2 + \tilde{\gamma}(u^2+v^2) > 0 \qquad \tilde{\gamma} \geq 0$

$$|\delta_\pm|^2 = \frac{\tilde{\gamma}}{1 + \frac{2v^2}{\tilde{\gamma}(u^2+v^2)}} \leq \tilde{\gamma}$$

denominator is > 1 since everything is positive

Case C3: $v^2 + \tilde{\gamma}(u^2+v^2) > 0 \qquad \tilde{\gamma} < 0$

that is, $v^2 > -\tilde{\gamma}(u^2+v^2) > 0$

$$|\delta_\pm|^2 = \frac{\tilde{\gamma}^2(u^2+v^2)}{\underbrace{\tilde{\gamma}(u^2+v^2)}_{<0} + \underbrace{2v^2}_{>0}} < \frac{\tilde{\gamma}^2(u^2+v^2)}{-\tilde{\gamma}(u^2+v^2)} = -\tilde{\gamma}$$

this denominator is bigger than

You continue to run thru all possible cases for the complex roots. But no matter how you look at it, you get the same bound. for all complex roots: $|\delta_\pm|^2 < |\tilde{\gamma}|$

In the next few cases, I'll show more systematically how all this is done. I use the relations I do because it shortens the result.

CASE B: REAL ROOTS

Case R: real roots, $r_\pm$

Because of our assumptions, $r_+ \geq r_- \geq -r_+$

Because $b/a \geq 0$, the average of the roots is non-negative.

To reduce the number of r's define:

$$r_- \equiv u r_+ \qquad -1 \leq u \leq 1$$

This substitution gets rid of a scale dependence on r. The bound on u leads to the following relationship:

$$|\delta_+| = |u\,\delta_-| \leq |\delta_-|$$

So we only have to get a bound for δ_- & we have a bound on δ_+ as well. Doing the substitution for $r_- = u r_+$ & cancelling the r_+'s:

$$|\delta_-| = \frac{|\tilde{\gamma}|}{\left|\left(\frac{1}{4}(1-u)^2 + \tilde{\gamma}\,u\right)^{1/2} + \frac{1}{2}(1-u)\right|}$$

GETTING BOUNDS FOR $|\delta_-|$

How do you go about establishing the bounds for $|\delta_-|$? First you may not play around with $\tilde{\gamma}$; it is an uncertainty for which you have a bound but you do not know what it is. You can think of u, however, as an independent variable; it is the ratio between the roots.

[redacted] If you are to get a bound that is valid, it must hold for all u. What we are really interested in is the maximum value taken by $|\delta_-|$ when $\tilde{\gamma}$ is fixed and u varies [redacted] from -1 to +1. But finding the maximum value is hopeless unless you know something extra about $\tilde{\gamma}$, namely that it is a small number.

So I will assume

$$|\tilde{\gamma}| << \frac{1}{16}$$

This says that on a hexadecimal machine [redacted] I'd like to have at least 1 digit correct. You can get by with $|\tilde{\gamma}| << 1/2$. But you do have to make this assumption. If $|\tilde{\gamma}|$ were allowed to vary freely, you might find that there was no simple & satisfactory bound.

Now you allow u to vary and plot $|\delta_-|$ [redacted] for small values of $|\tilde{\gamma}|$. You assign values to $|\tilde{\gamma}|$, compute the function for various u's & plot it. That gives you an idea of the range of values the

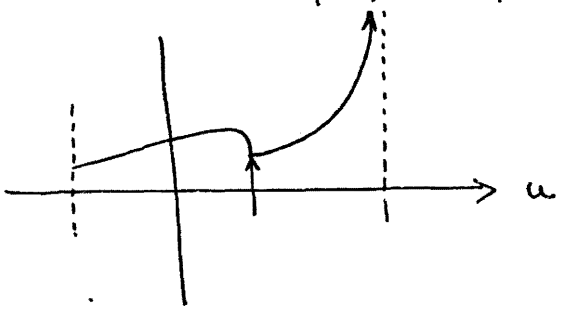
function can take.

Question: If $u = -1$, in $|\delta_-|$ you could be taking the squareroot of a negative number in the denominator.

Answer: that is possible. That corresponds to the discriminant of the perturbed equation being negative; the perturbed equation has complex roots while the original equation has real roots.

When you plot the graphs, you find that they have breaks points wherever the $(\ )^{1/2}$ term vanishes + changes sign. Whenever you get sharp breaks, you know you're going to have to look at cases. There isn't going to be one uniform algebraic scheme that'll do everything for you, because uniform schemes don't allow for notches in your graphs.

An example of what the graph might look like is:



You have to treat the graph in two separate parts. You get your cases by looking at the graphs.

SPECIFIC REAL CASES

Case R1: $\frac{1}{4}(1-u)^2 + \tilde{\gamma} u \le 0$: perturbed quadratic has complex roots even tho the unperturbed has real roots

We then find that:

$$\tfrac{1}{4}(1-u^2) \le |\tilde{\gamma}| \cdot |u| \ll \tfrac{1}{16} \quad \Rightarrow |u| > 1/2$$

u cannot be negative

$$|\delta_-|^2 = \frac{\tilde{\gamma}^2}{-\frac{1}{4}(1-u)^2 - \tilde{\gamma} u + \frac{1}{4}(1-u)^2} = \frac{-\tilde{\gamma}}{u} < 2|\tilde{\gamma}|$$

I get the above for $|\delta_-|^2$ because, in the denominator for δ_- I have the magnitude of a complex number; $(\frac{1}{4}(1-u)^2 + \tilde{\gamma} u)^{1/2}$ is imaginary. To get the magnitude squared, I take the square of the imaginary + the square of the real parts, + I must change the sign on the imaginary part squared.

All the rest of the cases are similar.

Case R2: $u \le 0$ & not R1

Case R3: $u \ge 0$ & not R1 & $\tilde{\gamma} \ge 0$

etc. You can see how to get the cases. In all cases, no matter what happens

$$|\delta_\pm|^2 < 2|\tilde{\gamma}|$$

In some cases you can have

$$|\delta_\pm|^2 = |\tilde{\gamma}|$$

This would happen if either set of roots were coincident.

The bound I have for δ is a good bound, and it is independent of the coefficients a, b, & c. If we took the coefficients into account, the bound would be terribly wrong, because it says if you carry single precision, then your roots will be perturbed to half-precision, in worse cases. And we know those worst cases can't occur often, or else nobody would solve quadratics on a computer.

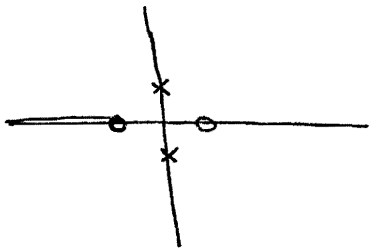
page 16

All of the trouble arises because of the singularity in the analysis when the roots are coincident. That causes the square on $|\delta_\pm|^2$. If you knew that the roots would be reasonably well separated, then the simple analysis with the differentials could be made rigorous, & you'd have gotten $|\delta|$ as some modest multiple of $|\tilde{\gamma}|$

SPECULATIONS ON THE ANALYSIS

Once you have finished this analysis of cases, there are some interesting speculations to make. First, if you use single precision, you'll only have half your digits correct, in critical cases. If you could figure out a way to solve the quadratic such that the perturbations in the coefficients were in double precision, then your results would be very nearly good to single precision. δ would be $\sim 2^{-95}$ so $|\delta_+| \sim 2^{-47}$; since all the other greek letters are no bigger than this, your result is good to roughly single precision.

It is this analysis that motivates the assertion that double precision will do. But there is a hooker. If you compute complex roots when the roots should be real, do you regard that as being correct to single precision? One sign of the perturbation is bound to be small.



o: roots should be real
x: roots computed as complex

The numbers may be correct to a few ulps, but you got the imaginary part quite wrong.

COMPLEX VS REAL ROOTS

Can you satisfy more stringent criteria, namely that not only do you compute the magnitude of the roots correct to a few ulps, but also that you compute each component of complex roots correct to a few ulps.

It is important for some people that this be done. Some people make a greater distinction between real & complex roots than is justified. Complex roots mean oscillations; real roots don't, in some electrical & mechanical problems. The difference, unfortunately, is o.

[redacted] exaggerated in their minds because of how they were taught.

They are solving an equation of the form

$$a\ddot{x} - 2b\dot{x} + cx = 0 \qquad \dot{x} \equiv \frac{dx}{dt} \qquad \ddot{x} = \frac{d^2x}{dt^2}$$

they solve the obvious quadratic & get a solution like

$$x(t) = A_+ e^{r_+ t} + B_- e^{r_- t}$$

If the $r_\pm$ are complex conjugates, you replace the above with a linear combination of sines & cosines, multiplied by suitable [redacted] exponentials. You can see the distinction which seems to be important. If the roots are complex, there is an oscillatory behavior; if the roots are real, there will not be an oscillatory behavior.

The above formula is invalid anyway, when the roots are coincident In that case we get

$$x(t) = (A + Bt)\, e^{rt} \qquad r \text{ the common root.}$$

There is a rather serious discontinuity in the way you would think about the problems as the roots pass thru coincidence. On one side, the roots are real & distinct, on the other the roots are complex conjugates, & somewhere [redacted] in between is the above [redacted] disconcerting solution, which doesn't look like it belongs to the same family at all.

If you look at the solutions to the differential equation, you would not be persuaded that so small a change in the coefficients could cause so drastic a change in the [redacted] behavior of the equation. Nor will it, for finite time. For moderate periods of time, small changes in the coefficients (keeping the initial values essentially unchanged) [redacted] will not make the solutions look much different. It is only as $t \to \infty$ that the change is drastic. [redacted]

Whether people ought to care or not, they do. And they would be upset if roots they were sure were real turn out computed as complex.

ANALYZING THE PROGRAM ITSELF

[redacted]

By a fluke, peculiar to the quadratic + other simple equations, an analysis of the program is easier to pursue than the analysis just worked out that asks what I can say mathematically about the roots if I perturb the coefficients, irrespective of how those perturbations were accomplished.

The formula you analyze is almost the same as the one you use in the computer, so analyzing one is very like analyzing the other. More generally, however, for equations more complicated than the quadratic, being able to divorce yourself from the algorithm would be an enormous simplification. Very often, the algorithm used in the machine bears only the vaguest resemblence to the mathematical problem being solved.

GETTING BACK TO THE PROGRAM

The quantity we were computing was:

$$d = b^2(1+\epsilon_1) - ac(1+\epsilon_2)$$

The question to be answered is:
When can this be written in the form:

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

where $|\delta| < m \cdot 2^{-47}$ m a small integer

If I can say this, the rest of the calculation will go thru easily.

I suggested that, [redacted] you compute b^2 and ac correctly as double precision products of single precision numbers. Then the only way to introduce the ε's is in the subtraction. You'll do a double precision subtract, normalize, & take the single precision part of the result.

If you do that, will you be able to write $d = (1+\delta)(b^2-ac)$? The answer is no. You can't even get this.

Why not? Suppose

$b^2 = 011\ldots\ldots111$ all the way thru double precision

$ac = 100\ldots\ldots00_{\text{u}}$

b^2 is a little less than a power of 2, ac is [redacted] power of 2.

In double precision, something will go wrong, & what goes wrong will depend on how it's coded. b^2-ac might be computed as zero; then I couldn't possibly find a reasonable value for δ.

that is: $d = 0$ but b^2-ac isn't. That means $\delta = -1$, which is quite unsatisfactory.

More likely, the last 1 in b^2 would be lost; then b^2-ac would be twice as big as it should be. That would make $\delta = +1$, also unsatisfactory.

In other double precision coding, the subtraction might be done exactly. You just don't know.

So there is the possibility that I won't be able to make a statement of the kind $d = (1+\delta)(b^2-ac)$. And in general, I cannot make such a statement.

We are reduced to seeing if we can get by with what the hardware provides. And the answer that

we can is essentially machine independent, in a rather funny way, as long as the machine works with digits in the manner we are accustomed to, in some reasonable base. As long as it has the property that the error is a few ulps committed in a reasonable fashion*, even tho δ is not bounded in a satisfactory way, we will still get a decent answer.

WHAT HAPPENS WHEN $d \geq 0$

Consider the case when $d \geq 0$; assume $b \geq 0$ also. Then I'll compute:

$$s = b + \sqrt{d}$$

The rounding errors are in single precision. How accurately can I compute this sum?

d is nonnegative and smaller than b^2. So for a binary machine say:

$$d \approx 2^{-i} b^2 \qquad \text{error in } d \approx 2^{-95} * b^2$$

that corresponds to the fact that a certain number of digits will cancell. (i could be a negative integer, but the only interesting case is when i digits have cancelled).

$$\sqrt{d} \approx 2^{-i/2} b$$

To find out what the error in $\sqrt{d}$ is, you need the relative error in d.

$$\text{error in } d \approx 2^{i-95} d$$

---

* that is, if the characteristics are equal, no error is commited; if they are not equal, the error made is tantamount to having perturbed one of the operands by a few ulps, tho exactly how it was perturbed you don't know.

relative error in $d \approx 2^{i-95}$ roughly

Therefore the error in $\sqrt{d}$ becomes:

error in $\sqrt{d} \approx 2^{i-95-1}\sqrt{d}$

That's the error inherited from the double precision subtract; the error in the squareroot we already know is unimportant. In taking the squareroot, it's relative error is decreased by roughly a factor of a half.*

error in $\sqrt{d} \sim 2^{i-95-1}\ 2^{-i/2}\ b$

$b + \sqrt{b} \approx (1 + 2^{-i/2})\, b$ absolute error $\sim 2^{i/2-95-1}\, b$

The relative error will be $< 2^{i/2-95-1}$

How big can i be; how many digits could you lose? You can't lose more than 96, so $i/2 \le 48$

So the relative error $< 2^{-48}$. There may be a couple factors of 2 that'll come in; maybe it should be 2^{-49}. The point is that the error made in the double precision subtract, however bad it is, will give a relative error in the end of at most that much. The other rounding errors will tend to mask that error. All that I really have to be sure of is getting the sign of d correct; that's important. If I get the sign of d wrong, then I'll get complex instead of real roots or vice versa, & that's what we're trying to avoid. If the sign of d is correct,

* computed $d = (1+\epsilon)D$ D is what is wanted $\epsilon \sim 2^{i-95}$

computed $\sqrt{d} = (1 + \frac{1}{2}\epsilon + O(\epsilon^2))\sqrt{D}$ from binomial exp of $(1+\epsilon)^{\frac{1}{2}}$

That's where the half comes from.

we've seen that we don't care how sloppy the arithmetic is.

To insure that the sign of d is correct, it suffices that the d.p. subtract preserves the monotonicity relations. (I don't know of any that don't, but it is easy to think of some.) All we want to arrange is that if

$$b^2 \geq ac$$

then when we compute we get

$$\text{"}b^2 - ac\text{"} \geq 0$$

WHAT HAPPENS WHEN $d<0$

The other interesting case is what happens when $d<0$. Now we are very concerned because $\sqrt{d}$ is essentially our result. The accuracy in $\sqrt{d}$ will be crucial.

Under what circumstances will there be an error at all? If b^2 and ac have the same characteristic, there won't be any error. The only way for something to go wrong is for

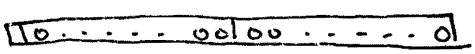
$$ac \geq 2^m$$

$$b^2 < 2^m$$ by a little bit; digits get lost, but how many do?

If b^2 is a little less than a power of 2, then it is less by at least (approximately) a single precision word's worth, in the lower half of a double precision word. (i.e. $b = 2^{48}-1 \Rightarrow b^2 = 2^{96} - 2^{49} + 1$

b^2: 

might be like this

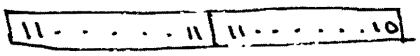
ac: 

then the error is ferocious if $ac = 2^m$

But things don't happen quite as you'd think. Usually, ?? for $b^2 < 2^m$, the bottom digit is a zero, so nothing is lost.

what?

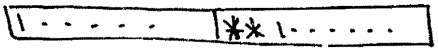
Say $b < 2^{m/2}$ when m is odd. Then b^2 is less than $\sqrt{2}$ times a power of 2; when that is squared, only 95 digits are needed to hold the product; normalization brings in a zero for the bottom bit.

b^2: 

when normalized

When m is even, b is a little less than a power of 2.
Consider $b = 1 - 2^{-48} n$ (ignoring the power of 2)
Then $b^2 = 1 - 2^{-47} n + 2^{-96} n^2$

b^2 is less than 1 by some digits in the top bits of the second half of the double precision word.

b^2: 

there are some zeroes up here.

There is almost a whole word at the bottom which will be different from ac. When you subtract, there may be errors at the right end, but you'll still have almost a whole single precision word correct for $b^2 - ac$. So instead of 1 ulp you may have 5 ulps for error.

SUMMARIZING THIS ANALYSIS

Hmmm!

This analysis goes thru for all machines in all bases. The odd thing is that in the complex case you can actually get a δ for $d = (1+\delta)(b^2 - ac)$, such that $|\delta| < 2^{-47}$

You can't get such a δ for $d > 0$, but that doesn't matter, as we've seen.

It is possible to solve a quadratic correct to a few ulps, except for over/underflow conditions.

Had we been obliged to use the rounding rule that says that the results of arithmetic operations are no worse than if you had perturbed each operand slightly, we would have gotten a crummier result than is in fact the case. Even using d.p. we could not have predicted the above result for complex roots. It was necessary to use an analysis that says we are carrying digits in a certain way.

APPENDIX TO LECTURE FOURTEEN

LOOKING AT THE CASES FOR THE PERTURBED QUADRATIC WITH REAL COEFFICIENTS

$$x^2 - 2bx + c(1+\gamma) = 0 \qquad \text{for } |\gamma| < \tfrac{1}{16}$$

THIS HAS ROOTS: $s_\pm = b \pm (b^2 - c(1+\gamma))^{1/2}$

Assume $b \geq 0$.

Let $r_\pm = b \pm (b^2 - c)^{1/2}$ [corresponds to $\gamma = 0$]

$s_\pm = (1 \pm \delta_\pm) r_\pm$

Then $r_+ + r_- = s_+ + s_- = r_+ + r_- + r_+\delta_+ - r_-\delta_-$ so $\boxed{r_+\delta_+ = r_-\delta_-}$

$r_+ r_- (1+\gamma) = s_+ s_- = r_+ r_- (1+\delta_+)(1-\delta_-)$ so $\boxed{(1+\delta_+)(1-\delta_-) = 1+\gamma}$

Case C: $r_\pm$ are complex

$r_+/r_- = e^{-2i\theta} = \delta_-/\delta_+$, so let $\delta_\pm = 2e^{\pm 2i\theta}\delta$

Now $\delta^2 - 2\delta \sin\theta - \gamma = 0$

Of the two roots δ, the lesser in magnitude corresponds to the more apt association of r_+ and r_- with s_+ and s_-, respectively or vice-versa.

Case C1 $\sin^2\theta + \gamma < 0$

Then both roots δ are complex and conjugate and have magnitude: $\boxed{|\delta| = \sqrt{-\gamma}}$

Case C2 $\sin^2\theta + \gamma \geq 0$

Then both roots δ are real with product $-\gamma$, so at least one has magnitude $\boxed{|\delta| \leq \sqrt{|\gamma|}}$

Case R: $r_\pm$ are real

take: $r_+ \geq r_- = ur_+ \geq -r_+$ (because $b \geq 0$)

hence $-1 \leq u \leq 1$

Then $\delta_+ = u\delta_-$, so $|\delta_+| \le |\delta_-|$, and we can consider only $\delta = \delta_-$; it satisfies

$$f(\delta) = u\delta^2 + (1-u)\delta + \gamma = 0$$

Of the two roots δ of $f(\delta) = 0$, the smaller corresponds to the more apt association of $r_\pm$ with $s_\pm$, respectively.

Case R1

$(1-u)^2 - 4u\gamma < 0$, hence the roots δ are complex conjugates and $u > 1 - 2\sqrt{u\gamma} > 1/2$.

Then $|\delta|^2 = \gamma/u < 2\gamma$ so $\boxed{|\delta| < \sqrt{2}\sqrt{\gamma}}$

(more precisely, $|\delta| < \sqrt{\gamma}(\sqrt{1+\gamma} + \sqrt{\gamma}) < 1.3\sqrt{\gamma}$; see case R4 for the idea behind the proof)

R2:

$-1 \le u \le 0$. Since $\frac{(1-u)^2}{-4u} \ge 1 > -\gamma$,

$f(\delta) = u\delta^2 + (1-u)\delta + \gamma$ has two real zeroes δ.

But $f(-2\gamma)/f(0) = \dfrac{4u\gamma^2 - 2(1-u)\gamma + \gamma}{\gamma} = -1 + 2u(1+2\gamma) < 0$,

so one of the zeroes δ lies between 0 & -2γ. $\therefore$ $\boxed{|\delta| < 2|\gamma|}$

R3:

$1 \ge u \ge 0$ and $\gamma < 0$; so $(1-u)^2 - 4u\gamma > 0$ and the roots δ are real. But now

$$\frac{f(\sqrt{|\gamma|})}{f(0)} = \frac{u|\gamma| + (1-u)\sqrt{|\gamma|} - |\gamma|}{-|\gamma|} = -(1-u)\left(\frac{1}{\sqrt{|\gamma|}} - 1\right) < 0$$

So at least one zero δ lies between 0 & $\sqrt{|\gamma|}$; $\boxed{|\delta| < \sqrt{|\gamma|}}$

(and equality is possible if $u = 1$)

R4:

$1 \geq u \geq 0$ and $\gamma > 0$ and $(1-u)^2 - 4u\gamma \geq 0$

this means both roots δ of $f(\delta) = 0$ are real.

It also means $0 \leq u \leq \hat{u} \equiv 1/(\sqrt{1+\gamma} + \sqrt{\gamma})^2$, since $\hat{u}$ is the lesser root of $(1-\hat{u})^2 = 4\hat{u}\gamma$.

Now $f(-\sqrt{\gamma/\hat{u}})/f(0) = (u\gamma/\hat{u} - (1-u)\sqrt{\gamma/\hat{u}} + \gamma)/\gamma$

$$= 1 - 1/\sqrt{\gamma\hat{u}} + u(1/\hat{u} + 1/\sqrt{\gamma\hat{u}})$$

$$\leq 1 - 1/\sqrt{\gamma\hat{u}} + \hat{u}(1/\hat{u} + 1/\sqrt{\gamma\hat{u}})$$

$$= 2 - (1-\hat{u})/\sqrt{\gamma\hat{u}} = 2 - 2\sqrt{\gamma\hat{u}}/\sqrt{\gamma\hat{u}} = 0$$

therefore one root γ lies between 0 and $-\sqrt{\gamma/\hat{u}}$, so

$$\boxed{|\delta| \leq \sqrt{\gamma}(\sqrt{1+\gamma} + \sqrt{\gamma}) < 1.3\sqrt{\gamma}}$$ since $\gamma < \frac{1}{16}$

SUMMARY

$s_\pm = (1 \pm \delta_\pm) r_\pm$ with $|\delta_\pm| < 1.3\sqrt{|\gamma|}$ for $|\gamma| < \frac{1}{16}$

Alternate Analysis

Patterned after Ostrowski : valid for complex coefficients

$$Q_\gamma(x) \equiv x^2 - 2bx + c(1+\gamma)$$

when $\gamma = 0$, $Q_0(x) = (x - r_+)(x - r_-)$: $\begin{cases} r_+ + r_- = 2b \\ r_+ r_- = c \end{cases}$

when $\gamma \neq 0$, $Q_\gamma(x) = (x - s_+)(x - s_-)$: $\begin{cases} s_+ + s_- = 2b \\ s_+ s_- = c(1+\gamma) \end{cases}$

$Q_\gamma(x) = Q_0(x) + \gamma c$, so

$$Q_\gamma(s_\pm) = 0 = Q_0(s_\pm) + \gamma c = (s_\pm - r_+)(s_\pm - r_-) + \gamma r_+ r_-$$
$$Q_0(r_\pm) = 0 = Q_\gamma(r_\pm) - \gamma c = (r_\pm - s_+)(r_\pm - s_-) - \gamma r_+ r_-$$

Ex. $\gamma \underset{①}{=} -\left(\frac{s_+}{r_+} - 1\right)\left(\frac{s_+}{r_-} - 1\right) \underset{②}{=} -\left(\frac{s_-}{r_+} - 1\right)\left(\frac{s_-}{r_-} - 1\right) \underset{③}{=} \left(1 - \frac{s_+}{r_+}\right)\left(\frac{r_+ - s_-}{r_-}\right) \underset{④}{=} \left(\frac{r_- - s_+}{r_+}\right)\left(1 - \frac{s_-}{r_-}\right)$

Let $\delta \equiv \max\left\{\left|\frac{s_+}{r_+} - 1\right|, \left|\frac{s_-}{r_-} - 1\right|\right\} \leq \max\left\{\left|\frac{s_+}{r_-} - 1\right|, \left|\frac{s_-}{r_+} - 1\right|\right\}$

to correspond to an apt choice of pairs $(s_\pm, r_\pm)$.

Therefore

Either $\left|\frac{s_+}{r_+} - 1\right| = \delta \geq \left|\frac{s_-}{r_-} - 1\right|$ (.0) or $\left|\frac{s_+}{r_+} - 1\right| \leq \delta = \left|\frac{s_-}{r_-} - 1\right|$ (.1) or both,

and either $\left|\frac{s_+}{r_-} - 1\right| \geq \delta$ (0.) or $\left|\frac{s_-}{r_+} - 1\right| \geq \delta$ (1.) or both.

In Case (0.0): $\left|\frac{s_+}{r_-} - 1\right| \geq \delta = \left|\frac{s_+}{r_+} - 1\right| \geq \left|\frac{s_-}{r_-} - 1\right|$ we find from ① that

$|\gamma| = \delta \left|\frac{s_+}{r_-} - 1\right| \geq \delta^2$, so $\delta \leq \sqrt{|\gamma|}$.

In Case (1.1): a similar application of ② yields $\delta \leq \sqrt{|\gamma|}$ again; merely repeat the foregoing argument after exchanging subscripts + and −.

In Case (0.1): $\left|\frac{s_+}{r_-} - 1\right| \geq \delta = \left|\frac{s_-}{r_-} - 1\right| \geq \left|\frac{s_+}{r_+} - 1\right|$

We shall show $\delta \leq K\sqrt{|\gamma|}$ where

$$K \equiv \sqrt{1+|\gamma|} + \sqrt{|\gamma|} = 1/\left(\sqrt{1+|\gamma|} - \sqrt{|\gamma|}\right)$$

If $\left|\frac{s_-}{r_+} - 1\right| \geq \delta/K^2$ then ② implies the desired conclusion. Therefore suppose, if possible, that $\left|\frac{s_-}{r_+} - 1\right| < \delta/K^2$; then ② would imply $\delta > K\sqrt{|\gamma|}$, which we shall show to be impossible.

First we find $\left|\frac{r_-}{r_+}\right| = \left|\frac{r_+ - (r_+ - s_-) + (r_- - s_-)}{r_+}\right| = \left|1 - \left(1 - \frac{s_-}{r_+}\right) + \left(1 - \frac{s_-}{r_-}\right)\left(\frac{r_-}{r_+}\right)\right|$

$$> 1 - \delta/K^2 - \delta\left|\frac{r_-}{r_+}\right|$$

whence $\left|\frac{r_-}{r_+}\right| > (1 - \delta/K^2)/(1+\delta)$

Secondly, ④ implies

$$|\gamma| = \delta\left|\frac{r_- - s_+}{r_+}\right| = \delta\left|1 - \frac{s_+}{s_-}\right|\left|\frac{r_-}{r_+}\right| \geq \delta^2\left|\frac{r_-}{r_+}\right| > \delta^2(1 - \delta/K^2)/(1+\delta)$$

whence

$$0 < \delta^3 - K^2\delta^2 - K^2|\gamma|(\delta+1) = (\delta - K\sqrt{|\gamma|})(\delta^2 - K\sqrt{1+|\gamma|}\,\delta - K\sqrt{|\gamma|})$$

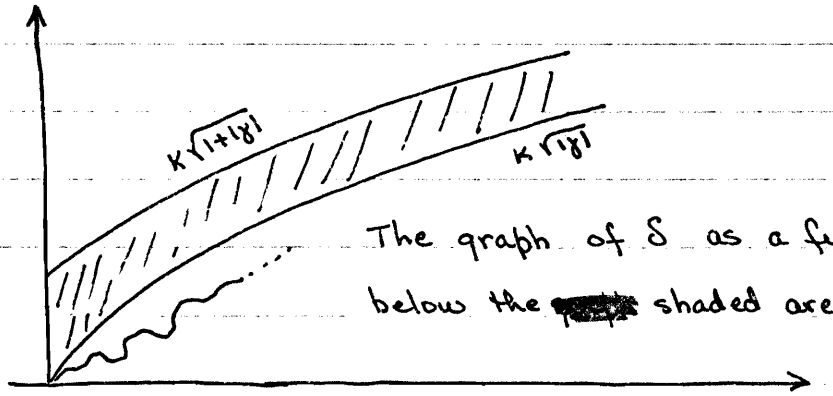
Since ② implied $\delta - K\sqrt{|\gamma|} > 0$ we should conclude that

$$\delta^2 - K\sqrt{1+|\gamma|}\,\delta - K\sqrt{|\gamma|} > 0 \quad \text{too,}$$

and hence $\delta > K\sqrt{1+|\gamma|} > 1$.

(In fact, $\delta > \frac{1}{2}K\left\{\sqrt{1+|\gamma|} + \sqrt{1-3|\gamma|+4\sqrt{|\gamma|}\sqrt{1+|\gamma|}}\right\}$ but we shall not need this).

Now think of s_+ and s_- as functions $s_\pm = b \pm (b^2 - c(1+\gamma))^{1/2}$ of γ; note that they are continuous functions of γ except when the way δ was defined forces s_+ and s_- to be swapped, but even so δ is a continuous function of γ and $\delta = 0$ when $\gamma = 0$. Therefore the inequality $\delta > K\sqrt{1+|\gamma|}$ is impossible for all sufficiently small $|\gamma|$, and consequently $\delta \le K\sqrt{|\gamma|}$ for those $|\gamma|$. But the continuity of δ and the fact that $K\sqrt{|\gamma|} < K\sqrt{1+|\gamma|}$ for all γ implies that $\delta \not> K\sqrt{1+|\gamma|}$, hence that $\delta \le K\sqrt{|\gamma|}$, for all γ:



The graph of δ as a function of γ starts below the shaded area and is not allowed to enter the shaded area.

Similarly for case (1.0); just interchange subscripts + and −.

<u>Conclusion</u> $\delta \le \sqrt{|\gamma|}\left(\sqrt{1+|\gamma|} + \sqrt{|\gamma|}\right)$ for all γ

which is our best upper bound for the relative errors in the perturbed roots $s_\pm$ caused by the perturbation γ. Can this bound be achieved for every $r_\pm$ and $|\gamma|$? No. But for each $|\gamma|$ there is a pair $r_\pm$ and a choice of sign (γ) for which the bound is achieved; see the earlier proof.

PERTURBATION OF A COMPLEX QUADRATIC EQUATION

Let the quadratic $x^2 - 2bx + c = (x - r_-)(x - r_+)$ be perturbed to $x^2 - 2bx + c(1+\gamma) = (x - s_-)(x - s_+)$. Our object is to obtain bounds for the differences between the zeroes $r_\pm$ and $s_\pm$ in terms solely of a bound upon $|\gamma|$. We do NOT assume that b or c are known. Consequently, among appropriate measures of difference between $r_\pm$ and $s_\pm$ are the relative differences $\delta_\pm = 1 - s_\pm/r_\pm$ respectively, for by substituting $s_\pm = (1 - \delta_\pm) r_\pm$ into the perturbed quadratic thusly

$$r_+ + r_- = 2b = s_+ + s_- = r_+ + r_- + r_+\delta_+ + r_-\delta_-$$

$$(1+\gamma) r_+ r_- = c(1+\gamma) = s_+ s_- = r_+ r_- (1 - \delta_+)(1 - \delta_-),$$

we obtain two equations:

$$r_+\delta_+ + r_-\delta_- = 0 \qquad (1 - \delta_+)(1 - \delta_-) = 1 + \gamma$$

for δ_+ and δ_- in which only the (unknown) ratio $z \equiv r_-/r_+$ appears instead of b and c. We seek bounds for $|\delta_\pm|$. We shall find

$$|\delta_\pm| \le \sqrt{|\gamma|}\left(\sqrt{1+|\gamma|} + \sqrt{|\gamma|}\right)$$

Certain ambiguities remain to be resolved. One is in the definition of $r_\pm = b \pm (b^2 - c)^{1/2}$; by insisting that $|r_-| \le |r_+|$, hence $|z| \le 1$, we eliminate that ambiguity except when $|r_+| = |r_-|$. The second ambiguity is in the definition of $s_\pm$. Here it is more convenient to observe first that $|\delta_+| = |-z\delta_-| \le |\delta_-|$, so only δ_- need concern us, and it satisfies the quadratic equation

$$(1 + z\delta_-)(1 - \delta_-) = 1 + \gamma$$

or $$z\delta_-^2 - (z-1)\delta_- + \gamma = 0$$

This equation has two roots δ_- corresponding to the two ways to define $s_\pm$. The apt definition is obviously the one corresponding to the smaller (in magnitude) root δ_-.

Thus we have reduced our problem to the following: given only $|\gamma| > 0$ find $\max |\delta(z,\tau)|$ over $|z| \le 1$ & $|\tau|^2 \le |\gamma|$ where $\delta(z,\tau)$ is the smaller root of

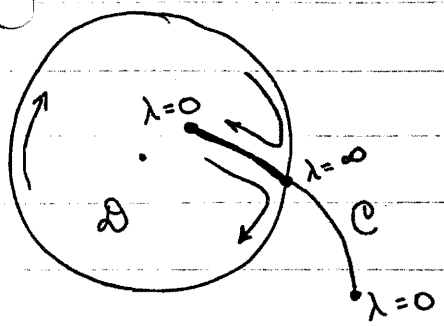
$$z\delta^2 - (z-1)\delta + \tau^2 = 0 \qquad (*)$$

Evidently $\delta(z,\tau)$ is a holomorphic function of z except for critical values of z where both roots of equation $(*)$ have equal magnitudes. The critical values of z turn out to be those for which $\lambda = 4z\tau^2/(z-1)^2 - 1 \ge 0$; these values all lie on a curved arc C traced out by

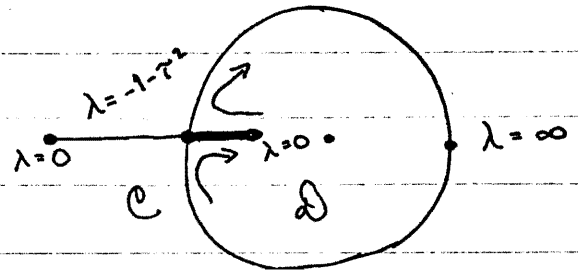
$$z_\pm = \left(\tau \pm (1+\lambda+\tau^2)^{1/2}\right)^2 / (1+\lambda) \qquad \text{respectively}$$

as λ runs through all non-negative values. Note that $z_+ = 1/z_-$, so part of C must lie in the disk $|z| \le 1$. Let D be the domain obtained from that disk by slitting it along C, so that the boundary of D consists of the circumference $|z| = 1$ transversed once plus that part of C inside the disk $|z| \le 1$ transversed twice.

Some examples follow:

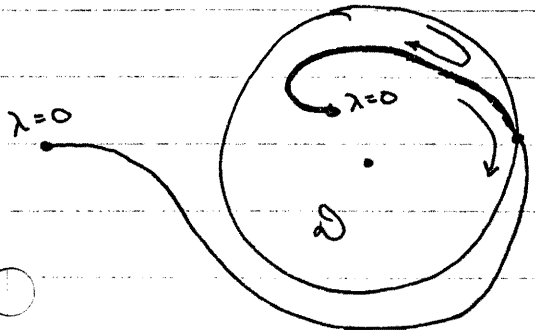


C is the arc



$\tau^2 < -1$

C is the circumference plus the line segment



C is the fallen question mark

The precise shape of C is immaterial, though we could show that, unless the circumference $|z| = 1$ is part of C, the arc C cuts the circumference just once at $z=1$, $\lambda = +\infty$.

Now we see that $\delta(z,\tau)$ is a holomorphic function of z inside D and continuous as z goes [redacted] to D's boundary. Therefore the maximum modulus theorem is applicable (Titchmarsh (1939) "The Theory of Functions" p 166-8), whence we conclude that the maximum of $|\delta(z,\tau)|$ over the disk $|z| \leq 1$, which is the same as the maximum over D, is achieved somewhere on D's boundary.

If the maximum is achieved on the circumference

$|z| = 1$ then, since the product of the roots of (*) is τ^2/z, we must have $|\delta|^2 \le |\tau|^2$, with equality when $z = 1$. If the maximum is achieved on C inside $|z| < 1$, we must still have $|\delta|^2 \le |\tau|^2 / |z|$, and hence the maximum of $|\delta|^2$ cannot exceed $|\tau|^2 \max_{z \in C} |1/z|$.

Now, on C

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

$$= (\xi + \sqrt{1+\xi^2})^2 \quad \text{for} \quad \xi = |\tau| / \sqrt{1+\lambda} \le |\tau|$$

Obviously $|1/z| \le (|\tau| + \sqrt{1+|\tau|^2})^2 \le (\sqrt{|\gamma|} + \sqrt{1+|\gamma|})^2$

so we find that

$$|\delta|^2 \le |\gamma| (\sqrt{|\gamma|} + \sqrt{1+|\gamma|})^2$$

with equality just when $\gamma \ge 0$ and $z = (\sqrt{\gamma} + \sqrt{1+\gamma})^{-2}$

In other words, the effect of $\gamma \ne 0$ upon the zeros of $x^2 - 2bx + c(1+\gamma)$ is to perturb them by a relative change δ, bounded by

$$|\delta| \le \sqrt{|\gamma|} (\sqrt{|\gamma|} + \sqrt{1+|\gamma|})$$

FIFTEENTH LECTURE, Tues Nov. 24, 1970

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

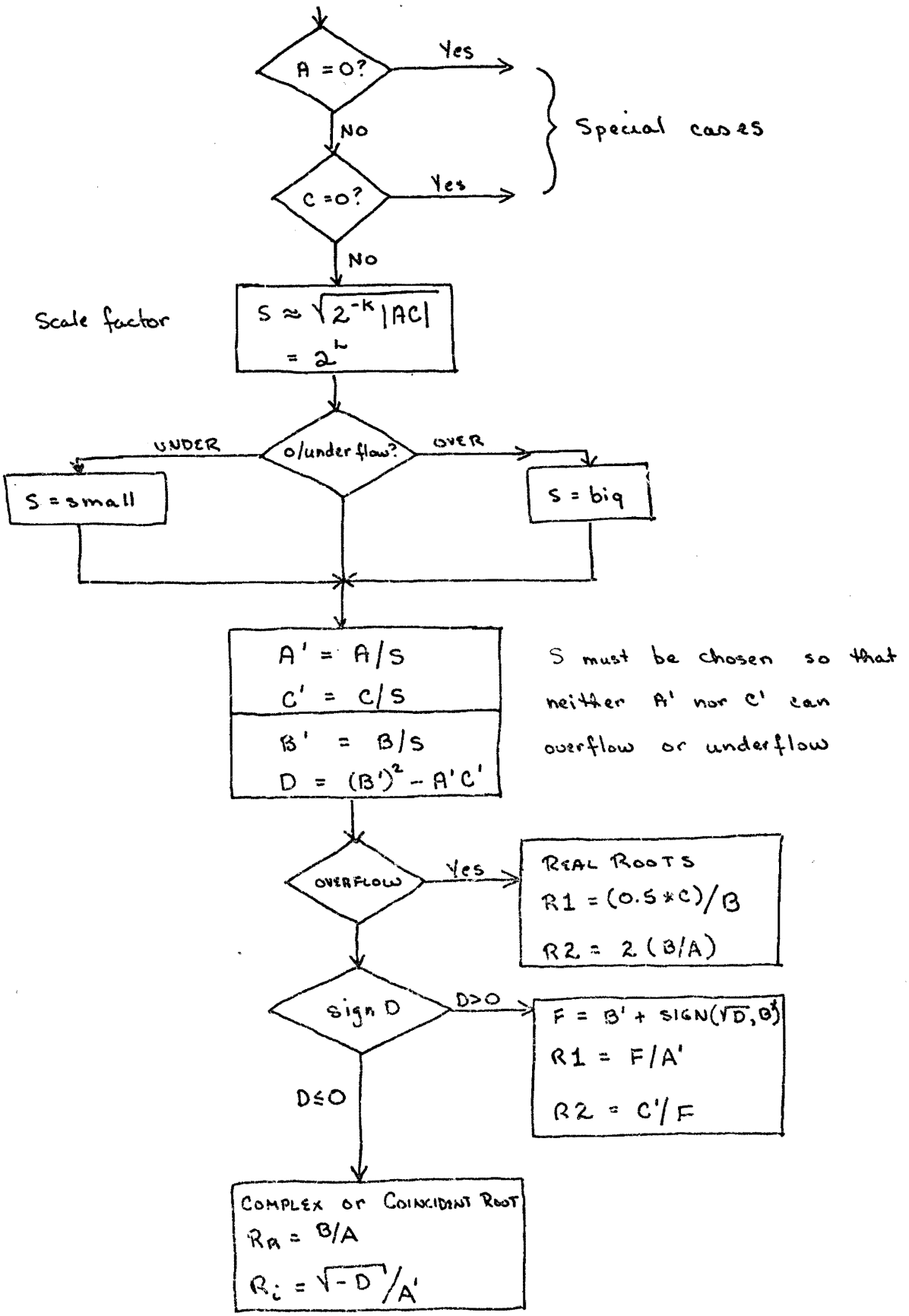
$$Ax^2 - 2Bx + C = 0.$$

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

The object here is to write relatively simple code to handle over/underflow, for most machines. Some choices indicated in the flowchart will have to be discussed with care, to see if choices can be made that follow the desired restraints. E.g., can an appropriate scale factor, S, be found so that A/s + C/s will never overflow nor underflow.

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

FLOWCHART: TO SOLVE $Ax^2 - 2Bx + C = 0$



SPECIAL CASES A=0 or C=0

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

Question: Wouldn't you look at the program a see if you were going to use A in multiplications or additions, to make your decision?
Answer: Yes, that is one rationale. But remember that that is of almost no consequence to the man who'll use your program, to solve a quadratic who doesn't care how it is done. A coefficient that is zero to you may very well not be zero to him.*

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

SCALE FACTOR S

Having accertained that neither A nor C is zero to the multiply unit, we can compute S. We will see later what value K must have, but for now simply

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

note that S is roughly the geometric mean of |AC|. The actual value chosen for S requires care. In the attempt to evaluate S, over/underflow may occur, but that actually is not serious. The object is to scale the whole equation by dividing thru by S in order to insure that the new A*C is a modest number close enough to 1 so that if the new $(B)^2$ overflows, we know that AC is negligible compared to B^2. On our machine, AC could be ~$10^{\pm 50}$ and that would be close enough.

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

OVER/UNDERFLOW IN S

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

---

* consider $A = 2^{1022+48}$, $C = 2^{1022+48}$: $AC = 2^{2044+96}$ overflow

if $S = 2^{600}$, $A' = 2^{422+48}$, $C' = 2^{422+48}$: $A'C' = 2^{844+96}$ in range

Question: It's not clear to me that a single S will do.
Answer: There is no single S. If S overflows or underflows, you don't choose S according to the formula.
Question: I meant a single S in any one situation.
Answer: That can be done. I'll indicate how to do it + leave the details to the students.

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

OVERFLOW IN B'= B/S

If B' overflows, then $(B')^2$ is so much larger than A'C', that we can neglect A'C' compared with B'. The roots then are relatively simple to compute: R1 = ½C/B, R2 = 2B/A

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

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

If $A'C' < 2^{1022+48-96}$, I cannot add it to a representable

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

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

The approximation when S over/underflows is crude because we cannot tell if AC overflowed by a little or a lot. A'C' could be $\sim 2^{800}$, but that is still in the acceptable range. If $(B')^2$ overflowed, A'C' can be thrown away without any more than a rounding error in double precision. Then the approximations R1 + R2 are correct to single precision.

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

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

You cannot use the primed values to compute R2 because B' may have overflowed.

COMPUTING S

I have to choose K in such a way that if |AC| is in range, the intrusion of the scale factor will not [illegible] louse things up. In getting to the point

where D didn't overflow, I must be sure that A'C' could not have overflowed or underflowed.

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

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

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

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

($2^{-1024+47}$ is zero to multiply box)

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

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

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

Now I'll take the squareroot & do something to it, & I'd better get 1.

I want a number bigger than $1 (2^0)$, so that when I take its SQRT & throw digits away, it'll be 1. I don't want a number bigger than 2 after taking the squareroot, so the original number must be less than 4, or less than 2^2.

$0 < 96 - 2 - K < 2$ (exponents)

So we have

$96 - 2 - K = 1$

or $K = 93$

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

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

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

DOES K=93 WORK FOR ALL OTHER NUMBERS

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

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

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

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

$$\sqrt{\ } = \sqrt{2(1-2^{-47})}\,(1+\epsilon) \approx \sqrt{2} \approx 1.4$$

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

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

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

When I take $\sqrt{\ }$ & throw away digits, nothing bad will happen.

By monotonicity, as long as $2^{1022+47} \le A \le 2^{1022+48}(1-2^{-47})$, nothing goes wrong.

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

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

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

TEST FOR SIGN OF D

D ≤ 0

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

$R_R = B/A$ or B'/A'

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

$R_I = \sqrt{-D}/A'$

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

Notice that if you had some decent way of turning off the spurious over/underflow responses, you could run in that mode until the test on D had been made. Then you could restore the mode wanted by the user before computing the actual roots, & if he wanted to be kicked off he would be. The only over/underflows that occur now are those that deserve to happen because the roots over/underflow.

D > 0

The roots are real and distinct.

First you have to compute

F = B' + SIGN(SQRT(D), B')

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

Now we compute the roots, & they could over/underflow.

R1 = F/A'

R2 = C'/F

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

ABOUT THE PROGRAM

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

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

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

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

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

The difficulty in their approach arises when you try to prove anything about a machine like ours which is capricious. If the program was reasonably simple & the number of rules was reasonably small, their formalization would appear to be quite successful. What I have done on the quadratic is essentially what they would do, stripped

of abstractions. It is possible to write down comments that enable you, at any point, to tell what the state of the machine is, subject to certain parameters. The parameters depend on the data. Every time you pass thru a decision you can give the new parameters in terms of the old. You could verify that certain relations remain satisfied by those parameters. But there is no systematic way to generate those relations, which depend on your objective. The men at Stanford have yet to prove the validity of any program half as complicated as the quadratic solver.

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

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

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

The example of the 29 incorrect squareroots was a time when I had the machine do some verification. But deciding what routine to use required ingenuity.

I don't think you can escape that for non-trivial programs. I think all you'd usually get from theorem proving was really just proof checking. But proof-checking could be tedious, & you may have trouble explaining to the machine that certain things are true (like properties of continuous functions).

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

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

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

BRIEF INTRODUCTION TO WHY BINARY IS BEST

To see why binary is best, you have to do a certain amount of error analysis, & then reflect on why you mean by an error of a unit in the last place.

In single precision on the CDC 6400, 1 unit in the last place means a relative error between 2^{-48} and 2^{-47}. Actually it is not quite as bad as that. Why is there this uncertainty? That is because the one ulp can be for either of these two numbers:

(a) 100 001

(b) 111 111

In (a), the error is 1 part in 2^{47}; in (b), it is 1 part in 2^{48}. So when I talk about an error of 1 ulp, there is an uncertainty about the bounds on the ϵ's; I want to write $(1+\epsilon)$, $|\epsilon| <$ something. If ϵ is at most 1 ulp, I have to say $|\epsilon| < 2^{-47}$, even though this bound is too big for numbers like (b) above. But 2^{-47} & 2^{-48} are not very different & we generally tend to ignore the difference.

But consider a hexadecimal machine like the 360, with 6 hex. digits. Then, 1 unit in the last place is between 16^{-5} & 16^{-6}

(a) 100001 one part in 16^5

(b) FFFFFF one part in 16^6 ($F_{16} = 15_{10}$)

When we now write $(1+\epsilon)$, $|\epsilon| < 16^{-5}$, the error bound may be too big by a factor of 16. That leads to certain difficulties.

Error propagates as relative error when you

multiply or divide. [redacted]

$$(1+\delta)(1+\delta) \approx 1+\epsilon+\delta$$

The bound on the sum is the sum of the bounds. On multiply/divide, the error propagates as the sum of the relative errors.

When you add or subtract, you really want to count error in ulps; you want to talk about an absolute error. Thus, given an operand with a bound on its relative error, you must convert that to a bound on its absolute error.

Consider C * X, where the operands are known to within 1 ulp. Clearly, their product should be know to 2 ulps. But on a hexadecimal machine, [redacted] it could be 31 ulps.

EXAMPLE IN DECIMAL

3.1 ± .1	3.2	3.0
3.2 ± .1	3.3	3.1
9.9|2	10.56	9.3|0

10.5

9.9

9.3

9.9 ± .6 not 9.9 ± .2

SIXTEENTH LECTURE, TUES. DEC. 1, 1970

Continuing the discussion of why binary is best. It is of course possible to live with a nonbinary machine (Knuth is prepared to write 7 volumes using MIX, which may be binary or decimal or something else); [redacted] my arguments are to show you that binary is better, not to persuade you to believe as I do that binary is best.

ERROR AND UNCERTAINTY

I started last time to show that the nature of error + uncertainty changes in our eyes when we consider different operations. For addition + subtraction, we think of the error & uncertainty as an absolute error & uncertainty because those are the things which add.

Consider $C = A + B$

$c = a + b$ in machine, plus error from adding

How uncertain is the sum c? in terms of the uncertainties in a and b? It is the absolute uncertainties that matter.

$|a - A| < \alpha$

$|b - B| < \beta$

$|c - C| < \gamma = \alpha + \beta$ (ignoring the error in addition)

[can consider $\alpha + \beta +$ {1 or 1/2 ulp in c from adding}

but then $c = a+b$ is not correct, may need $c = rnd(a+b)$

Consider $C = A * B$ Now the relative errors will add.

$c = a * b$

Now we have in mind something like

$$|\log {}^{A}/_{a}| < \alpha \quad (\text{or } |1 - {}^{A}/_{a}| < \alpha)$$
$$|\log {}^{B}/_{b}| < \beta$$
$$|\log {}^{C}/_{c}| < \gamma = \alpha + \beta$$
$$\text{or } \alpha + \beta + \{1 \text{ or } 1/2 \text{ ulp carried}\}$$

NUMBERS ANALOGOUS TO ERRORS

This way of looking at error is analogous to many a compiler writer's ways of looking at numbers. Some numbers are integers, to be used for indexing; some are real numbers to be used in floating point calculations. In a language like Algol, an attempt is made, in the syntax of the language to obscure the different functions the numbers may play; they are essentially different functions. Indexing is a very different way of looking at a number from looking at a number used in floating point operations. When you index, you are not really so interested in how big that number is (its cardinality), but rather in what you might call the ordinality aspect of the number; the numbers form a sequence, & you can use these to select entries in another sequence made to correspond to the integers. In most machines you expect a good deal of conversion between integers and floating point representations of integers, as the context in which the integers are to be used shifts.

Similarly in error analysis, you'd expect a good deal of conversion to take place from relative to absolute uncertainty, or vice-versa, as the context of the analysis changes.

WHY I PREFER BINARY

The main reason for my preferring binary machines is that in this conversion process as little uncertainty as possible is added on a binary machine. On other machines, the mere process of conversion adds an uncertainty. We saw, in an example, that on a decimal machine, what you mean by one unit in the last place interpreted as an absolute error must, when interpreted as a relative error, be a little uncertain by a factor near 10.

One ulp in 1000000 and one ulp in 9999999, as relative perturbations, vary by a factor near 10. On a binary machine, the variation is only a factor of 2. It is possible that a machine that used a number closer to 1 as its base would have even less variation, but nobody builds machines like that. On a logarithmic machine, this problem would not arise because one ulp would always be a relative error; there wouldn't be any other way to talk about it.

NAIVE BOUNDS

To illustrate that the difficulties are not really crippling but they are annoying, I'll show you a calculation in which the bounds, that a naive person might assign to uncertainty by reckoning only on ulps, could be disconcertingly wrong, if the base of the machine is big enough. Division yields more interesting examples than multiplication.*

* Multiplication will be left as an exercise for students to work out.

EXAMPLE

Suppose the following is part of a larger calculation. Compute $\frac{A}{B} - \frac{X}{Y} = D$, in circumstances when I know there will be lots of cancellation; if the difference is at all appreciable, though, I'd like to know it accurately.

Assume that I can find A, B, X, Y to within $\pm \frac{1}{2}$ ulp (using a conventional rounding machine, instead of a truncating machine – in this example the difference is merely a factor of 2).

Suppose I know that

$$.95 \leq \{ A/B, X/Y \} < 1$$

A large number of digits will disappear when I subtract. I'm not going to worry about that because I am interested in the absolute error. So I want to estimate the absolute error in D due to roundoff.

Figure the errors thusly:

error in A is at worst $\frac{1}{2}$ ulp

" " B " " " " " "

When I compute A/B (rounded), the errors are $\frac{1}{2}$ from numerator, $\frac{1}{2}$ from denominator, and $\frac{1}{2}$ from division

error in $\frac{A}{B}$ is $\frac{3}{2}$ ulp (thinking of the errors as relative errors, which is wrong)

similarly error in $\frac{X}{Y}$ is $\frac{3}{2}$ ulp

Now when we compute $D = A/B - X/Y$, there'll be no rounding error from the subtraction; the numbers will have the same characteristic, or if one number was rounded to 1, the

rounding after the operation will take care of that. The only error that D inherits is the sum of the errors in A/B and X/Y. So D is in error by at worst 3 ulps, of .999···99 (this number has the same characteristic as A/B and X/Y); this gives an upper bound on the error, roughly. How roughly?

Getting this estimate was painless. But on a machine whose base is bigger than 2, this estimate can be wrong by a correspondingly big factor. Actually, it is already wrong by a factor of 2 on a binary machine; on a hexadecimal machine, the factor is 16, and that factor of 16 is a little upsetting.

DECIMAL EXAMPLE

decimal machine with three significant digits

A = 104.49	a = 104
B = 104.51	b = 105
X = 104.51	c = 105
Y = 106.49	d = 106

a/b "=" .990 (rounded)	A/B = .999809
x/y "=" .991 (rounded)	X/Y = .98 1407
d = [redacted] -.001	D = .018402

There is an error of 19 units in the last place instead of 3. This is about as bad a case as you can get.

MORE RIGOROUS ERROR BOUND

We can explain incidents of this sort by getting a more rigorous bound on the error, and then see if this is a worst case. I think the discrepancy in the example is big enough to surprise you. If you were carrying more digits, you wouldn't be too upset by this. The fact that you'd lost 2 digits out of 14 instead of losing 1 is not seriously upsetting. Unless you are a person who must produce a subroutine for others to use & guarantee its accuracy to within 1 ulp. You have a severe problem if the last operation is a division; you have the numerator & denominator to ½ ulp, but the division may not give you a result good to 1½ ulp. On a hex machine, the error could be a good deal bigger & we'd like to find out how much bigger.

GENERAL ANALYSIS

Consider a general machine of base b that carries p digits ($p > 4$). Take two numbers x & y such that;

$$b^{p-1} + 1 \le x \le b^p \quad , \quad b^{p-1} + 1 \le y \le b^p$$

I chose that particular range, so that I could make statements like:

$x - \frac{1}{2} \le X \le x + \frac{1}{2}$ — x approximates X to within ½ ulp

$y - \frac{1}{2} \le Y \le y + \frac{1}{2}$ — X originally lay in the range b^{p-1} to b^p, and was rounded to x.

all cases are now covered; put decimal point at far right of word, & then we have all ways of approximating fl. pt. no. converted to integers & rounded to integers

Let q = rounded value of x/y

Now we can get bounds on x/y from the bounds on x and y.

$$\underbrace{b^{-1} + b^{-P}}_{\frac{\text{smallest } x}{\text{biggest } y}} \leq \frac{x}{y} \leq \underbrace{\frac{b^P}{b^P + 1}}_{\frac{\text{biggest } x}{\text{smallest } y}} = b - b^{2-P} + b^{3-2P} - \ldots$$

when you round, all digits past the second term get thrown away

$$b^{-1} + b^{-P} \leq q \leq b - b^{2-P}$$ all numbers are representable

There are two cases, because the characteristic of q may take on two different values. The range is roughly b^{-1} to b, which allows q to have 0 or +1 as a characteristic.

CASE 0

$x \leq y$ so $b^{-1} + b^{-P} \leq q \leq 1$

Now 1 ulp in q is b^{-P}, except for $q = 1$ (but that won't matter). We are interested in the error made when we round x/y to get q.

Rounding x/y to q creates at most $\frac{1}{2} b^{-P}$ of new error.

CASE 1

$x \geq y$ so that $1 \leq q \leq b - b^{2-P}$ and 1 ulp of q is b^{1-P}, and the error added is at worst $\frac{1}{2} b^{1-P}$.

So we have the extent of the rounding error in the quotient q. But that's not the uncertainty in q. All we really have is:

$$|q - x/y| \leq \tfrac{1}{2} b^{\gamma - P} \qquad \gamma = 0 \text{ or } 1$$

ERROR INHERITED BY q

We still have to find the error q inherits because we had to round $X * Y$ to $x * y$. So we now compute

$$\left|\frac{x}{y} - X/Y\right| = \left|x/y\right| \left|\frac{x-X}{x} + \frac{X}{x}\frac{Y-y}{Y}\right|$$

$$\leq \tfrac{1}{2}\left(\frac{x}{y}\right)\left(\frac{1}{x} + \frac{X}{x}\frac{1}{Y}\right) \qquad \text{upper bound on quotient}$$

since $x - X \leq 1/2$ and $Y - y \leq 1/2$

The upper bound for the error in the quotient is approachable arbitrarily closely by making the appropriate inequality signs nearly equality signs.

At this point we'll approximate; we have at least 4 digits of base b of precision in:

$$\left|\frac{x}{y} - \frac{X}{Y}\right| \approx \tfrac{1}{2} q \left(\frac{1}{x} + \frac{1}{y}\right) \qquad \text{since } \frac{X}{x} \approx 1,\ \frac{1}{Y} \approx \frac{1}{y}$$

(we do not need an error bound of higher precision than 1 ulp of number of digits being carried in the original calculation)

$$\left|\frac{x}{y} - \frac{X}{Y}\right| \doteqdot \tfrac{1}{2}\left(\frac{1}{y} + \frac{x}{y^2}\right) \qquad \text{equivalent expression to the above}$$

Case 0 will be the worst case

We know $x \leq y$, although they could be arbitrarily close, (to within 1 ulp).

$$\left|\frac{x}{y} - \frac{X}{Y}\right| \leq \tfrac{1}{2}\left(\frac{1}{y} + \frac{y}{y^2}\right) = \frac{1}{y} \qquad \text{(to within 1 ulp)}$$

that's how big the error could be

$$\approx b^{1-p} \qquad \text{(smallest } y\text{)}$$

All of the equality signs are simultaneously achievable. To make all inequality signs turn into the rough equality signs, the difference

between $x + y$ must be ½ ulp off the same sign as the difference between $x+y$, x must be approximately equal to y, & y must be one of its smallest possible values. (That tells you how I got those 3 dec. digit numbers). Now

$$\left| q - \frac{x}{y} \right| < (\tfrac{1}{2} + b)\, b^{-p} = (\tfrac{1}{2} + b) \text{ ulp in } q.$$

The error is not what we would have expected from the naive analysis. It it not ½ ulp but rather (½ + b) ulp. If $b = 2$, 5/2 is not so different from 3/2.

Case 1

$y < x < by$ (the range on x & y is less than a factor of b)

$$\left| \frac{x}{y} - \frac{X}{Y} \right| \lesssim \frac{1}{2y}(1+b) \qquad \text{saying } \frac{x}{y} \approx b$$

That equality is achievable, by making x at the top of the range and y at the bottom. So, with y as small as possible, we get

$$\left| \frac{x}{y} - \frac{X}{Y} \right| \lesssim \tfrac{1}{2}(1+b)\, b^{1-p}$$

Add this to the error committed in rounding the quotient.

$$\left| q - \frac{X}{Y} \right| < \tfrac{1}{2}(b+2)\, b^{1-p} = \tfrac{1}{2}(b+2) \text{ ulp of } q$$

I said Case 0 was really the worst. That is because if you measure in terms of ulps

$(\tfrac{1}{2} + b) > \tfrac{1}{2}(b+2)$ because $b > 1$

So in terms of ulps, case 0 is worse. this is relative error.

But if you measure in absolute error terms, you have

$$1/2 + b < 1/2\,(b+2)\,b.$$

Then case 1 is worse.

GETTING ALL THE INEQUALITIES TO BE EQUALITIES

You get your examples by choosing numbers indicated by these two cases. There is a small problem; when you choose numerical quantities to make the inequalities almost equalities, you can see how to make all the inequalities work except for

$$|q - x/y| \le \tfrac{1}{2} b^{x-p}$$

I've told you how to accomplish all the others, e.g., y is always near the bottom of the range. But having done all those things, you can't be sure of getting the above to be nearly equal. I'll show you how to achieve that as well.

CAN THE ERROR BOUND BE ACHIEVED?

The point is not so much to generate examples for amusement, but rather to see if a computed error bound is fairly close to one of the actual errors. An error bound is useless if it is very much larger than any error that could occur. So having found an error bound, you might wish to see if it is realistic. This is not necessary if the bound is small enough to ignore. But as long as we are quibbling about last digits, we'd like to know if the bound is achieved.

DIVISION EXAMPLE

$y = b^{p-1} + m + n$ m is a small positive integer

$x = b^{2p-1} + n$ $|n|$ " " " " "

$m + n > 0$

x + y are digit strings, starting with a 1, followed by some zeroes, + then some 1's at the bottom.

$$\frac{x}{y} = \frac{b^{p-1}+m}{b^{p-1}+m+n} = 1 - nb^{p-1} + n(m+n)b^{2-2p} - n(m+n)^2 b^{3-3p} + \dots$$

by looking at this we can see how to round $\frac{x}{y}$

this series converges rapidly, usually, because $m + n$ are small compared to b^p.

In doing the rounding, I must consider cases.

CASE 0 : ROUNDING

$n > 0$ because $x < y$

$\underbrace{1 - nb^{p-1}}_{\text{fit into 1 word}} + \underbrace{n(m+n)b^{2-2p}}_{\text{fit in double precision part of word}}$ + ignorable terms

: but it add, so it'll get chopped off and thrown away, unless $n(m+n)$ is huge.

We only run into interesting difficulties if

$$n(m+n)b^{2-2p} \doteq \tfrac{1}{2} b^{-p} \quad (\tfrac{1}{2} \text{ ulp of number less than } 1)$$

Then in rounding we'll commit the maximum possible error of $\pm \tfrac{1}{2}$ ulp.

$$\underbrace{2n(m+n)}_{\text{small integers}} \doteq \tfrac{1}{2} \underbrace{b^{p-2}}_{\text{big integer}}$$

We could have $n \sim b^{p/3}$

$m+n \sim b^{2p/3}$ (n is negligible)

You should check that the series will still converge with these estimates on n and m. Case 1 runs in a similar fashion.

Remember, the object of this analysis is merely to show that the error bound in the quotient of ½ ulp could be approached. You can do the same sort of analysis for two products that you know are very close.

MAIN REASON WHY BINARY IS BEST

In conclusion, we could have an error of $b^{+1/2}$ ulps instead of 3/2 ulps.

This is the main reason I have for preferring binary. In calculations for which I must give an error bound, it is easier to give that bound on a binary machine than on some other base machine, say hexadecimal, simply because of the conversions I have to do. The conversions introduce uncertainties, and while the uncertainties don't make the results worse necessarily, they just make it harder to say what has happened. The error bound for some method may be so pessimistic that you abandon that method for a better one, when you needn't have done so.

Tight error bounds are not necessary on all problem, but they are important for elementary function subroutines.

HOW MANY DIGITS SHOULD YOU HAVE

There is another analysis that not only attempts to say what base you should use but also how many digits to carry to optimize the design of some arithmetic unit.

When you ask how many digits, it is very hard to get an answer that is not based upon experience of the following kind.

I observe that people have been using k decimal digits for a long time perfectly happily. Therefore I guess k is enough. But in this argument you can always reduce k by 1 — if they were happy with k, presumably they'd still be happy if you took 1 away. By induction, you don't need any digits at all!

SLIDE RULE ACCURACY

A more serious fallacy is this: are people content with 3 digit accuracy on slide rules because real life only demands that much accuracy, or have they decided only to work on problems for which a slide rule is adequate, because any other computing device was awkward to get at. Have they learned to live with a slide rule? which is the case?

So it is hard to tell how much precision should be used, on the basis of experience of that kind.

DOUBLE PRECISION MAY BE ENOUGH

Other arguments lead to the following tentative conclusion. In certain kinds of error analysis, we often find a bit more than double precision is enough, to give a simple answer, if your data is good to a certain number of digits. If your data is floating point numbers, the error, in calculations done to double precision plus a little, is almost always due to uncertainties in the data, not to the rounding errors.

You'd like to think that if the data has 4 digits, carrying 6 would be enough. But in many problems, doing the naive, reasonable thing will not give you an answer that is as

good as it deserves to be.

An example is certain kinds of least squares problems like:

$$\min_x \sum_j \left(\sum_k a_{jk} x_k - b_j \right)^2 \qquad \text{given } a \text{ \& } b$$

$$= \min \| \underset{\sim}{b} - A \underset{\sim}{x} \|^2 \qquad \|\underset{\sim}{v}\| \equiv \sqrt{\underset{\sim}{v}^T \underset{\sim}{v}} \quad \text{(Euclidean norm)}$$

To solve this set of equations you write down:

$$A^T A \underset{\sim}{x} = A^T \underset{\sim}{b} \qquad \text{gives you the minimizing } \underset{\sim}{x}$$

This method has the difficulty that in computing $A^T A$ & $A^T \underset{\sim}{b}$ and going thru the process of linear equation solving, you ould carry twice as many digits and a bit as are valid in the data for A & $\underset{\sim}{b}$. Failure to do so can lead to answers more wrong due to rounding errors than to uncertainty in the data.

WORD LENGTH

Experience says double precision is enough. So look around at the data you plan to manipulate, represent it in some base as accurately as it deserves, double the number of digits & add a little bit for fun, and say that that is the normal word length for single precision calculations. My rule of thumb is very rough, but I'm trying to give some rationale for deciding how many digits to carry. This rationale makes the 24 bits on the 360 reasonable, until they decided to make it hexadecimal.

In industrial tolerances, 1% is rough but sometimes acceptable,

while 0.1% is considered fine work. For example, machining a 1" shaft to .001" is considered good. Slide rule experience also indicates that 1 part in 1000 is adequate accuracy. It seems that slide rule accuracy is enough for the overwhelming majority of calculations, when the man sees the numbers before him + he can see when something goes wrong. If he can't see the cancellations happening, you need to double the number of digits.

One part in 1000 is 10 bits; double that + you get 20; throw on a couple for good measure + you get 22 or 24 bits. In my opinion, 24 bits would serve for the majority, doing routine industrial + engineering calculations. This is born out by the fact that 6-place tables are as accurate as are generally available. In fact, 4-place table were + still are very popular.

Thinking in terms of decimal digits, one part in a thousand means 4 decimal digits, doubling means 8 or 9 decimal digits, but that takes more than 24 bits to represent. In hexadecimal arithmetic, things are even worse.

ARE LARGER BASES BETTER

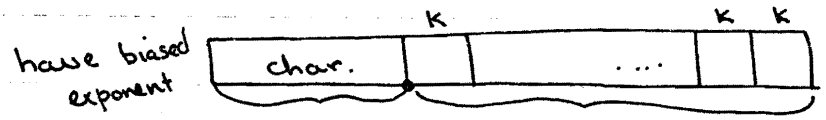
Maybe what happened on the 360 is that someone reasoned, maybe like I did above, that 24 bits was enough + also that hexadecimal would do. His argument for hex is one intended to optimize the cost of the machine on the assumption that shifts (normalization) are expensive. It has been noted that most

normalization shifts are small; if you can eliminate the small ones, you've saved something. With hexadecimal, you can eliminate the shifts because you allow 1, 2, or 3 leading zeroes.

Then people came up with spurious arguments that the larger the base, the better your useage of the digits that must represent the characteristic. In hex base, each digit of the characteristic represents a much wider range than in the case of binary; you have 16^k instead of 2^k; so relatively small integers in the characteristic can still give you a wide range. So people came up with the fallacious idea that for a given total number of digits, you got a wider range for a certain precision by using hexadecimal. I'll give you a rough idea of what an argument of this kind looks like.

CHOOSING A BASE

For a given number of digits, how should you choose the base to get the best useage out of those digits?



k binary digits to a clump; p such clumps

base = 2^k

Now, how do I choose k, p, & ℓ in order that for a total number of digits I use the register in the best possible way? Turn this around & say I wish to accomplish a certain range; i.e., I want all numbers in a certain range to be

representable. I also want to specify the precision of the numbers. Then, how many digits do I need?

The range for the exponent will be:

$$-2^l \le exp \le 2^l - 1$$

(The characteristic is l digits plus a bias digit. A sign digit was discarded because it wasted a bit pattern in representing a minus zero.)

$$\frac{\text{largest no.}}{\text{smallest no.}} = \frac{(2^k)^{2^l-1}}{(2^k)^{-2^l-1}} = (2^k)^{2^{l+1}-2}$$ the range

(can have a small no. almost unnormalized)

precision: relative uncertainty is $(2^k)^{1-p}$

(the worst relative error I can get)

cost: total bits $= p*k + l$

What happens when I fix the range + precision, + minimize the cost?

(Take $\log_2$ of all numbers)

fixed range $R = k(2^{l+1} - 2)$

fixed precision $Q = k(p-1)$

minimize $pk + l = (Q+k) + l$

$$= Q + l + R/(2^{l+1} - 2)$$

Now you have a function of 1 variable and you can use calculus.